MICROSOFT® PROGRAMMING SERIES

Microsoft®

Windows

k Included

D-ROM

Programming Microsoft® Outlook® and Microsoft Exchange

Second Edition

Includes Digital Dashboard Starter Kit

Build collaborative business solutions with Microsoft Outlook 2000, Exchange Server 5.5, and Exchange 2000

Thomas Rizzo

Microsoft®

Programming Microsoft® Outlook® and Microsoft Exchange

Second Edition

Thomas Rizzo

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Cataloging-in-Publication Data
Rizzo, Thomas, 1972-
Programming Microsoft Outlook and Microsoft Exchange / Thomas Rizzo.--2nd ed.
p. cm.
ISBN 0-7356-1019-3
1. Application software--Development. 2. Microsoft Outlook. 3. Microsoft Exchange.
I. Title.
QA76.76.A65 R59 2000
005.369--dc21 00-028169

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 QMQM 5 4 3 2 1 0

Distributed in Canada by Penguin Books Canada Limited.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at mspress.microsoft.com. Send comments to *mspinput@microsoft.com*.

Acquisitions Editor: Eric Stroo
Project Editor: Victoria Thulman
Technical Editor: Marzena Makuta
Manuscript Editor: Michelle Goodman

For Stacy, my family, and the Ecksteins, my new family.
I love you.

Contents at a Glance

Part IV Exchange Server 2000 Development

Table of Contents

Part II Building Outlook Applications

Acknowledgments

Writing a book is never a one-person job. It always involves help from other people, who are either reviewing your work or teaching you things you never knew before. The first people to thank are the folks at Microsoft Press, especially Eric Stroo, John Pierce, Victoria Thulman, Marzena Makuta, Ben Ryan, Michelle Goodman, Roger LeBlanc, Rob Nance, and Dan Latimer. Their hard work is greatly appreciated.

Thanks are owed to a number of people throughout Microsoft who helped in one way or another with this endeavor. First and foremost, thanks must go to the members of the Microsoft Exchange and Microsoft Outlook teams, who contributed to my knowledge of both Exchange and Outlook. These people include Jim Van Eaton and Bob Gering, who contributed their expertise of XML, XSL, and WebDAV as well as OWA to the Training application in the final chapters; and the Exchange Developer team, including Nat Ballou, Andrew Sinclair, Naveen Kachroo, Robert Brown, Alex Hopmann, Jim Reitz, Lisa Lippert, Chuck Daniel, Dana Birkby, Seth Cousins, Brent Jensen, Charles Eliot, and Denise Smith. They filled me in on the latest and greatest of the Exchange 2000 technologies. Thanks must also go to the Exchange Solutions Product Unit and the Exchange SDK team, including Keith McCall, Janine Harrison, Michael Patten, Steve Biondi, and Bruce Hamilton for providing a wealth of documentation and knowledge on Exchange development technologies. I owe a thank you to the Exchange Realtime Collaboration team, including Rick Ryan and Dalen Abraham. Another debt of gratitude must go to the Exchange management team, including Gord Mangione and Eric Lockard, because I stole the time of their valuable people to answer my questions.

On the Outlook team, I have to thank Ramez Naam, Chris Kimmel, and Don Mace for their support on all the Outlook technologies described in this book.

Thanks to Roman Lutz, who in the middle of a presentation I was giving on CDO, showed me some techniques on using CDO and Outlook together. You can find some of his techniques highlighted when I discuss the Outlook View control.

Thanks to Noam Topaz, who planted the idea and provided a kick start for the Site Server COM Add-in included in this book. His unique perspective on the types of applications you can build on the Exchange and Outlook platforms is refreshing. You can always depend on Noam to give you some unique ideas for applications you can build.

Thanks must also go to a number of other people who helped this process along by supporting it and understanding why I could never get together with them when chapters were due. This includes John, Kim, Michael and Katie Hand, Jen and Steve

Fowler, Dan Fay, Rajeev and Arpita Agarwal, Ed Yoon, Marie Maxwell, Jim and Vivian West, Vicky and Gary Halperin, Paul Miller, Ria Johnston, and Joanne Bromwell.

A special thanks goes to Randy Lehner, who provided the excellent security application that you'll find described in the Exchange 2000 chapters.

Finally, I have to thank the people I work with everyday for allowing me to complete this work. These people include Paul Gross, Russell Stockdale, Branch Hendrix, Stan Sorensen, and Gytis Barzdukas.

About the Companion CD

The companion CD contains sample applications discussed in this book, supplementary chapters and sample applications, and programming help files, as well as the electronic version of this book, additional information, and some software you might need to view the CD files.

You will find the sample applications in the Samples folder. You can browse these samples from the CD, or you can install them onto your hard disk. If you haven't disabled the autorun feature in your Microsoft Windows installation, a splash screen will appear when you insert the CD into your CD-ROM drive. You can then install the samples, help files, or other CD content. The instructions on your screen will guide you in the installation process. You can also access the splash screen by running StartCD from the CD root directory. To save your disk space, you might want to install only the selected applications.

Please refer to the appropriate Readme file for specific instructions on how to run the application.

To uninstall the sample files, open Add/Remove Programs in Control Panel, select the application you want to remove, and click Add/Remove. (In Microsoft Windows 2000, click Change/Remove.)

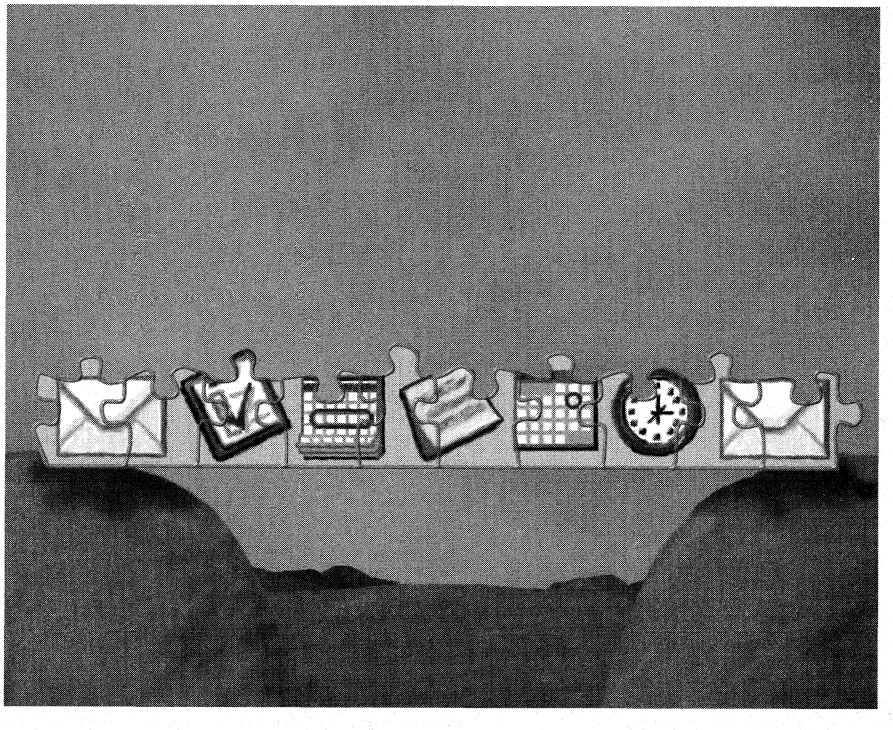
System Requirements

To run the sample applications, you will need to meet the following system requirements:

- Microsoft Windows 2000 Server, or Microsoft Windows NT 4.0 Service Pack 4 or later
- Microsoft Exchange Server 5.5 (or Microsoft Exchange Server 2000 for some sample applications)
- Microsoft Outlook 2000

Part I

Introduction to Collaborative Systems



Chapter 1

A Broader Definition of Collaboration

If you asked ten different people to define collaboration in a computer environment, you would receive ten different answers. Some would say collaboration is e-mail. Others would mention video teleconferencing or the World Wide Web. You might even hear Internet chat as an answer. People struggle to define collaboration because there are so many technologies and its definition today is broad. Really, all of these answers are correct. Collaboration—at least in part—is the integration of many different technologies into a single application or environment to facilitate information sharing and information management.

Integrated technology, however, is only one aspect of collaboration as we're defining it. Timing is another. We're all familiar with real-time collaboration in which you work with others at the same moment, taking turns communicating ideas. But new technology offers you an entirely different way to collaborate—asynchronous collaboration—in which you don't have to be present to participate. Asynchronous collaboration allows you, at your convenience, to collaborate with other people, at their convenience. E-mail, public databases, the Internet, and intranets are all forms of asynchronous communication.

Collaborative technology provides these key benefits to businesses:

- *Extensive, secure communication.* Collaborative technologies enable extensive communication through many different mediums and secure communication through encryption and digital signature technology, which is critical as businesses increase their use of the Internet.

- *Storage of information in a central location.* Information is placed in a central repository, or database, so that individuals inside and outside a corporation can access it. If shown in a threaded view, the history of the information is accessible and new information can be added to it.
- *Ability to extend existing technologies with new functionality and bridge islands of information.* Collaborative systems connect disparate systems and facilitate finding and sharing information stored in existing technologies. Essentially, they bridge islands of information.

How does a collaborative system provide these benefits to corporations? In terms of its architecture, a collaborative system must have several characteristics. First, it must have a robust, replicated object database that can store many different types of information such as web pages, office documents, and e-mail messages, and it must support replication both from server to server and from server to client. This replication allows geographically dispersed individuals to access diverse information. To work with the data, the database needs to allow many different clients, ranging from web browsers to e-mail clients.

Second, it must support the Internet and industry standards. The days of stovepipe computing are over. New technologies are connecting disparate networks to form one global, cohesive network. A collaborative system must be able to interoperate with these networks over the Internet, and it must follow industry standards to allow openness to a large number of external systems as well as guarantee the integrity of the data.

Third, a collaborative system must offer powerful, easy-to-use development tools and technologies. The environment must be open so that developers can use any tool to develop solutions and users can access and customize the user interface.

TOOLS FOR BUILDING COLLABORATIVE SYSTEMS

Microsoft offers a number of products and tools that are designed to help you leverage a company's current technology investments and extend them with new functionality. These tools, which fall under three key product types, are listed here:

- *Client products.* Tools include Microsoft Outlook and Microsoft Internet Explorer.
- *Server products.* Tools include Microsoft Exchange Server, Microsoft SQL Server, Microsoft Internet Information Services and Microsoft Site Server.
- *Development products.* Tools include Microsoft Visual Studio, which encompasses Microsoft Visual Basic and Microsoft Visual InterDev.

The two main tools you will want to learn are Microsoft Outlook and Microsoft Exchange Server. Both provide a robust infrastructure with which corporations can run mission-critical services. Combine this infrastructure with the rich development tools provided by both products and you have a powerful platform on which you can write solutions. The type and complexity of these solutions can range from simple forms to complex applications. The next few sections briefly describe some products and tools available from Microsoft for building collaborative solutions.

Microsoft Outlook

Outlook supports the ability to manage information (e-mail messages, appointments, contacts, and tasks) and share it throughout an organization. Outlook also includes a development environment that allows you to write collaborative applications quickly. Part II of this book is dedicated to Outlook and its development environment. Chapters 8 and 10 in particular discuss and illustrate the features of Outlook 2000 that will enable you to further extend Outlook.

Microsoft Internet Explorer

With the ubiquity of the Internet, browser technology is becoming increasingly important for user collaboration. Internet Explorer, with its support for dynamic HTML, scripting, and security, is an ideal client interface for your applications. In Chapters 12, 13, and 14, you will see how to take advantage of Internet Explorer using both Outlook and Exchange Server.

Microsoft Exchange Server

Exchange Server, which is part of the Microsoft BackOffice suite of products, is a linchpin for any collaborative system because it supports communication, information sharing, and workflow services that use Internet standards and protocols. Chapter 2 of this book provides an introduction to Exchange Server.

Microsoft SQL Server

SQL Server is a relational database system that offers easy storage and retrieval of information. Its built-in data replication, powerful management tools, Internet integration, and open system architecture allow you to integrate SQL Server into existing environments cost-effectively.

Microsoft Internet Information Services

Internet Information Services (IIS) is a free web server available for Microsoft Windows 2000 Server. It provides an easy way to publish and share information securely over corporate intranets and the Internet through HTML documents. The power of IIS is

demonstrated when web applications are written using its built-in server-side script technology called Microsoft Active Server Pages (ASP). ASP allows developers to write applications by using any ActiveX scripting language, such as JScript or Microsoft Visual Basic Scripting Edition (VBScript). These scripts execute on IIS and can access different data such as that provided by Exchange Server or SQL Server. The information returned from the server-side script is in HTML, making these applications compatible with any standard HTML web browser such as Internet Explorer. Chapter 7 introduces Active Server Pages and its programming model.

Microsoft Site Server

Site Server is a web publishing, analysis, and search tool. Because Site Server is integrated with Windows 2000 Server and IIS, you can easily set up and deploy intranets. Site Server helps corporations get the most from their intranets by implementing best practices for publishing and staging intranet content.

Site Server can also implement content tagging. Content tagging is a structured, site vocabulary that authors use to classify the web content they create. When used in conjunction with Site Server's integrated search and knowledge management capabilities, these tags enable users to more easily find information. Plus, Site Server integrates and manages the information from other BackOffice products through full-text indexing of these different data sources.

Microsoft Visual Studio

Visual Studio is an integrated and comprehensive suite of development tools for building web-based or Microsoft Windows–based applications. You can quickly build collaborative solutions that take advantage of the BackOffice family of products because Visual Studio and BackOffice are integrated. Throughout this book, you will see examples of collaborative solutions that use Visual Studio tools.

Microsoft Visual Basic

Visual Basic, a component of Visual Studio, is an effective and easy-to-use tool for creating high-performance windows applications. It includes a rapid development environment with graphical layout tools and great performance because of native code compilation. Visual Basic also creates open, industry-standard ActiveX components. These components can provide functionality to other applications whether they are web-based or Windows-based.

Microsoft Visual InterDev

Visual InterDev, a component of Visual Studio, empowers web application developers to rapidly build fully interactive, dynamic web sites. With visual development features and powerful database tools, Visual InterDev provides the most complete and technically

advanced development system for building both intranet and Internet applications. Through the use of Visual InterDev Design–time controls and wizards, you can add collaborative technologies to your web applications.

EXAMPLES OF COLLABORATIVE SOLUTIONS

Now that you have a better understanding of collaboration and collaborative technologies, let's look briefly at the systems you can create. With Exchange Server, you can build many different types of open and extensible applications, all of which can take advantage of information stored inside and outside of Exchange Server. You can leverage other data sources in your organization, such as SQL Server databases. This openness to other data sources allows you to pick the best database for storing the application's information without compromising the user interface consistency.

The types of applications you can build can be broken down into five categories: messaging, tracking, workflow, real-time, and knowledge management. None of these application categories are mutually exclusive—for example, a workflow application can take advantage of messaging services. Rather, these categories define the primary function of a particular application. Throughout this book, we'll explore sample applications that fall into these five categories.

Messaging Applications

Messaging applications use primarily the messaging infrastructure of Exchange Server. E-mail is the best known of these, but you can build many other types, such as discussion group applications. Exchange Server supports threaded discussions; you can make any folder in Exchange Server a threaded discussion folder by changing the view of the messages inside that folder. These discussions can be replicated to and from Internet newsgroups and can be moderated for the appropriate content.

Another example of a messaging-based application is a mailbox agent. A mailbox agent can perform many different types of functions based on how it is programmed. For example, suppose a sales force needs the ability to run certain queries against a database of sales information. Although you could write a Microsoft Access application that queries the database and returns the results, a salesperson wouldn't be able to work on other items until the database processed the request and returned the data set in the Access user interface. This means that users would have to check the Access application continually to see whether the data was available. However, with a mailbox agent, a salesperson could use a form to specify the type of information she needed and then e-mail the form to the agent. The agent would process the form and run the query on her behalf. Once the database was finished processing the query, the agent would e-mail the data set to the salesperson. Eventually, she receives notification containing the requested data set.

A mailing list server is a messaging agent that forwards all mail it receives to its registered recipients and allows users to add and remove themselves from the list of recipients via e-mail.

> **NOTE** To see a mailing list application in action, sign up for the Microsoft Exchange Server mailing list at *http://www.msexchange.org.*

A document library is another example of a messaging application. Users can submit documents to a library by dragging and dropping them, e-mailing them, or sending them through a web browser. Because these libraries are stored in a central location, many users have access to the documents. Intelligence can be added to the library by creating a mailbox agent that notifies users when new documents are available. Custom views are available on a folder so that users can quickly find desired documents. Comments about the documents can be placed in the folder, and interested users can gauge their relative value.

One popular example of a document library is a library of web favorites. A user can set up a document library to store a corporation's favorites in a central location. By dragging and dropping Internet shortcuts into this library, a user's personal favorites become corporate favorites. Plus, users get the benefit of being able to create custom fields and views that describe and categorize the favorites in the folder.

Tracking Applications

Tracking applications manage and track information, such as a list of contacts, from its creation to its deletion or "completion." Tracking applications usually require the integration of many different data sources because the information needing to be tracked typically resides in more than one location.

One example of a tracking application is a job candidate tracking application, which enables a human resource department and other employees to track a prospective employee from the moment he submits a resume through the interview process and finally to the decision to hire or reject. The candidate's status is always available for review. Figure 1-1 shows a hypothetical example of a job candidate tracking application that uses Outlook and Exchange Server to track prospective candidates.

You could also create an account tracking application, which includes tracking for contacts, revenue, and tasks. Figure 1-2 shows the Account Tracking application we'll build in Chapter 6. In Chapter 10, we'll enhance this application for Outlook 2000.

Helpdesks are also tracking applications. In a helpdesk application, trouble tickets are submitted to the helpdesk by users specifying technical problems. Problems are assigned to technicians based on the ticket type. Audit trails are established for each ticket so that the technicians have historical information that helps them work on the problems. After fixing a problem, the technician adds the ticket and its resolution to a log of frequently asked questions, which users can query.

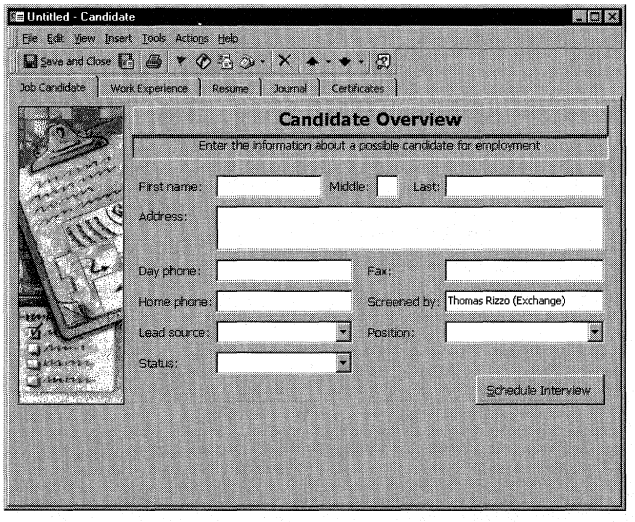


Figure 1-1. *A hypothetical job candidate tracking application in Outlook.*

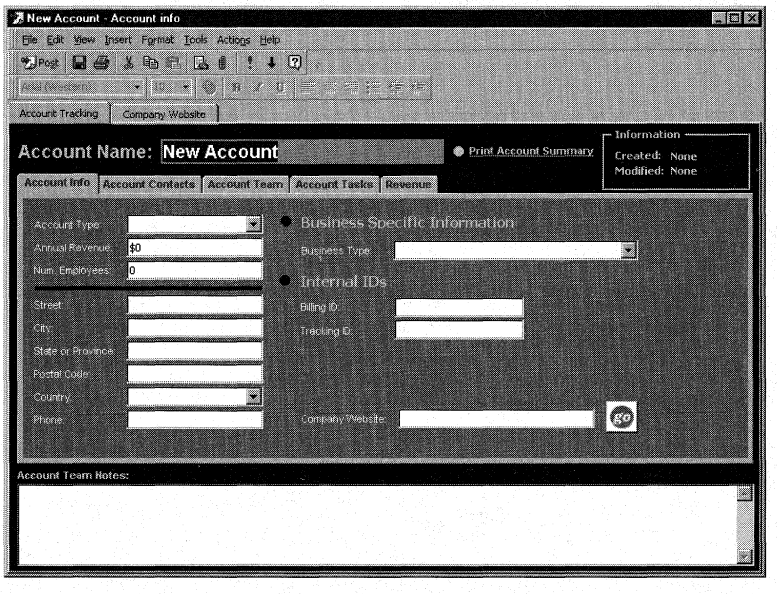


Figure 1-2. *The Account Tracking application from Chapter 6.*

A helpdesk might include other tracking applications as well, such as inventory management. For example, if the technician had to request a new machine for the user, an inventory management program informs the technician whether a new machine is in stock. By adding a workflow application to the helpdesk application, the technician

could obtain approval for the machine from the user's manager and the help desk manager. Figure 1-3 shows the Helpdesk application we will build in Chapter 12.

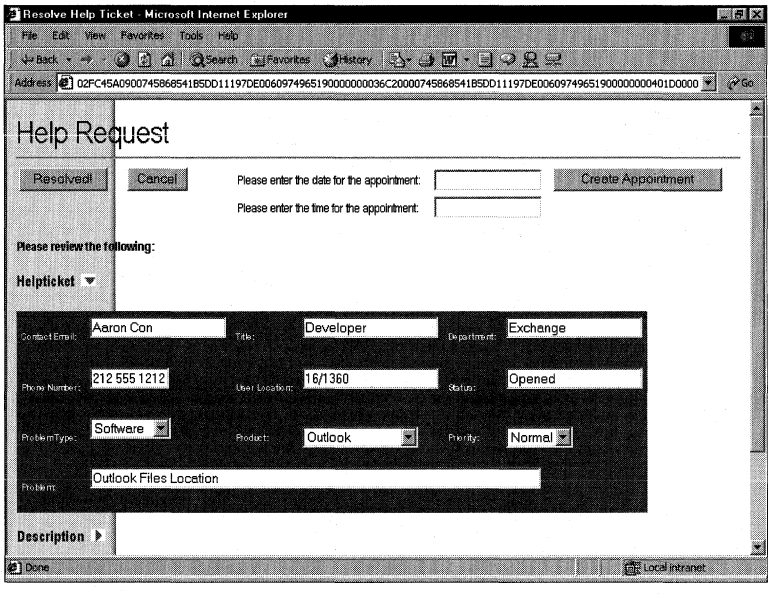


Figure 1-3. *The web-based Helpdesk application from Chapter 12.*

One last example of a tracking application you might build is a class registration application, which tracks information about a class and its participants. It informs users when desired classes become available, reminds them of which classes they are registered for at least one day in advance, and notifies them of any updated materials made available by the teacher. When the class is completed, class notes and a survey can be distributed to class members.

Workflow Applications

Workflow applications are primarily constructed around three concepts, which are known as the three Rs—Roles, Routes, and Rules:

- *Roles.* A role is the logical representation of a person or an application in a workflow process—for example, expense report approver Roles can change dynamically depending on who is involved in the particular workflow process. They allow you to easily abstract the different functions people perform in a workflow process.
- *Routes.* A route defines what information will route and who will receive it. Routes can be sequential, parallel, conditional, or any combination of these. Figure 1-4 illustrates three types of routes.

- *Rules.* A rule is conditional logic that assesses the status of the workflow process and determines the next steps. Here's an example of a rule: *if the manager approves the expense report, route the report to accounting, or else send the expense report back to the submitter.* A rule can be based on the properties of a message or on some other data source.

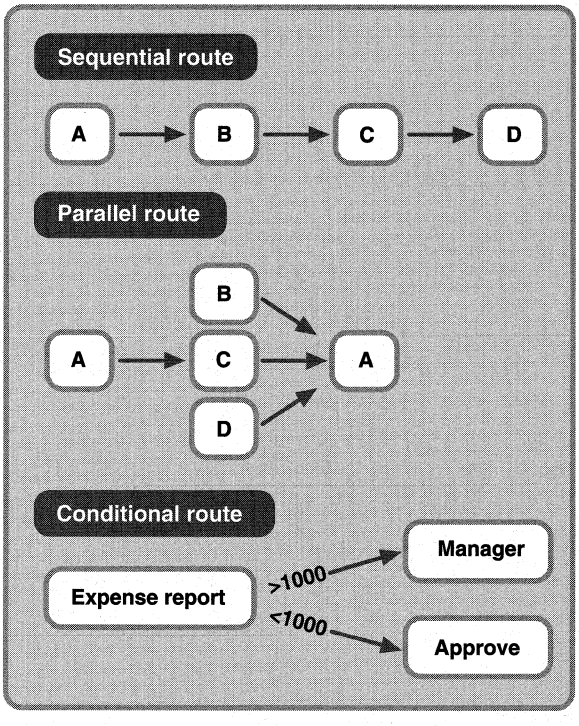


Figure 1-4. *Sequential, parallel, and conditional routing types.*

Let's take a brief look at a few workflow samples. The Expense Report application, which is discussed in detail in Chapters 13 and 14, is one example of a workflow application that you can build with Exchange Server. Here's how such a workflow application might function: a user submits expense reports from a web application, and based on the total amount showing in the expense reports, a particular workflow process is started. If the expense is under $1,000, the expense report is automatically approved; if the expense is at or over $1,000, the report is routed to the user's manager for approval. The manager either approves or rejects the expense report, and based on his decision, another workflow process is initiated to either pay the expense report or inform the user that the expense report has been denied. Finally, if the manager does not approve or reject the expense report in a certain period of time, the workflow application reroutes the expense report to the manager's manager for approval. Figure 1-5 shows an example of the Expense Report application in action.

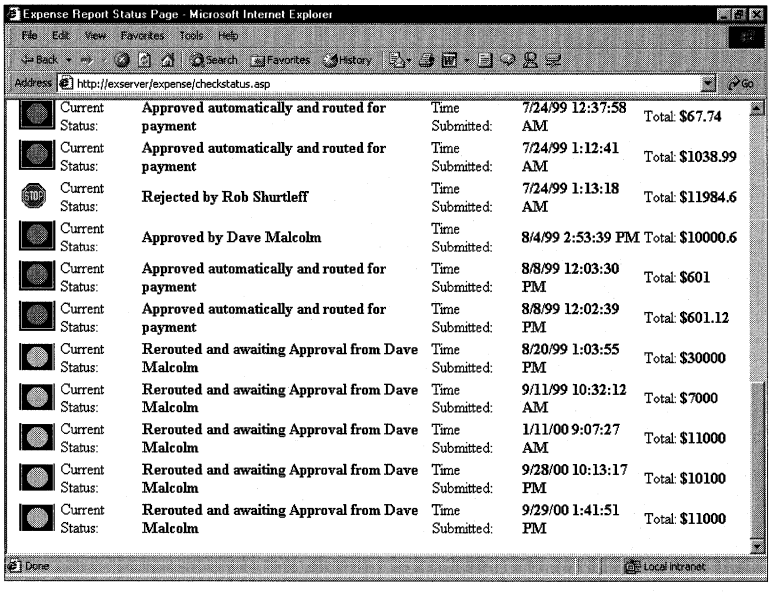


Figure 1-5. *The web-based Exchange Server Expense Report application from Chapters 13 and 14.*

Another example of a workflow application is a document routing application, in which a document to be reviewed is routed to users in parallel, user feedback is collected within a certain period of time and consolidated into a single message, and the consolidated message is sent to the originator of the workflow application. Chapter 14 will show you how to create an application like this using Microsoft Exchange Server Routing Objects.

Real-Time Applications

Real-time collaborative applications are the newest category of Exchange Server applications. Real-time applications have the potential to enable instantaneous collaboration (as compared to the "delayed" collaboration of messaging-based applications). The challenge, of course, is to connect geographically dispersed users in real time. When you combine real-time and messaging technologies, you can build applications that leverage the strengths of both.

One example of a real-time application that you can build with Exchange Server is a class registration system that schedules virtual classes by sending Microsoft NetMeeting requests. NetMeeting allows individuals to collaborate over the Internet using video, audio, whiteboards, and application-sharing technology, as shown in Figure 1-6. In Chapter 6, you'll examine an Account Tracking sample that demonstrates how to integrate NetMeeting into your own application.

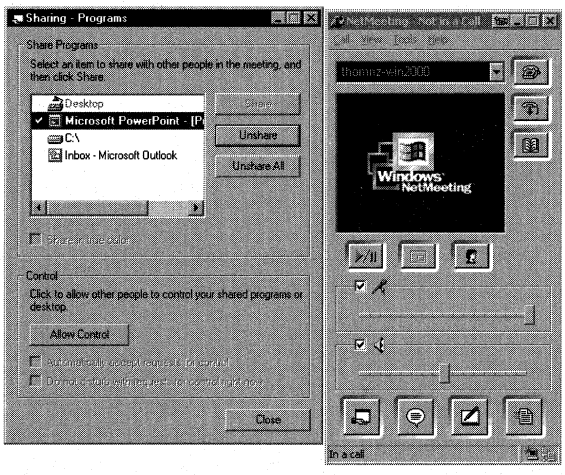


Figure 1-6. *NetMeeting allows you to collaborate with other people in real time.*

Another real-time application that will interest you as an Exchange Server developer is a new technology called Instant Messaging. Instant Messaging allows users to monitor when other users are online so that they can collaborate with one another. It allows two different organizations to create virtual "buddies," or business partners. Instant Messaging is like a virtual water cooler!

CHAT ENABLES REAL-TIME COLLABORATION

Chat—a popular service on the Internet today—is one example of a real-time application. Chat enables real-time conversation by allowing a participant to type in messages that appear instantly on another participant's computer. When added to collaborative applications, chat can greatly enhance functionality for users. For example, you can extend a help desk application with chat services so that help-desk technicians can hold "office" hours during which they conduct real-time question-and-answer sessions. Those chat transcripts can be posted to a discussion group so that other users can troubleshoot questions based on the transcript.

Knowledge Management Applications

"Knowledge management" refers to the use of collaborative technology to implement structured processes for finding and gathering information—in other words, it is a strategy for moving information from the individual to the larger group or corporation. At a time when corporations want to leverage the information that their intellectual as-

sets—people—possess, knowledge management applications are critical. They make available all kinds of information, from individual experiences to best practices to detailed technical data. The effective sharing of knowledge brings to a company three primary advantages: more effective use of existing intellectual assets; competitive advantage through the pooling of resources and greater accessibility of important information; and new opportunities and more focused innovation. Although knowledge management is a new term and a new strategy for mining and sharing information, it uses technology that has been available since Exchange Server first shipped. Applications based on this strategy are called knowledge management applications.

Implementing a Strategy to Manage Knowledge

You can use collaborative technology such as Exchange Server to employ a knowledge management strategy, but collaborative technology is not synonymous with knowledge management. That is, corporations must establish processes that will not only collaborate but also gather and make accessible information that is current, relevant, and tested.

You might be wondering what types of applications you can build with Exchange Server to implement the concept of knowledge management. One type is a search application in which you can search discussion groups and contacts in Exchange Server, as well as search in SQL databases and web sites. This search capability is a very powerful tool. One important benefit of universal search engines is that users do not have to change the way they collaborate because the search engine crawls the necessary data sources to retrieve the relevant information.

Another type of application that facilitates knowledge management is a knowledge base. By developing knowledge bases with Exchange Server, you can enhance conventional collaborative methods. Typically, knowledge bases are used by corporate users who post free-form, unmoderated messages to a common folder. Users who want specific information, such as text in a message, query the knowledge base in a general way and then cull all the returned information that meets their criteria. Because of the general nature of the queries and unstructured way information is posted, many of the results are irrelevant or invalid.

Imagine how a more structured method of entering and searching for information in a knowledge base could facilitate collaboration and knowledge management. Suppose users who were posting information to a knowledge base had to fill out a form that asked them to categorize their information, indicate how long it would be valid, and rate its usefulness if it originated from external sources. Users would be able to query on categories and ratings and receive only current and relevant information. By supplying just a little extra information, users make the data stored in the knowledge base infinitely more useful. And if you added smart agent technology to the application, you could program the knowledge base to e-mail links to relevant information that meets users' predefined criteria.

Chapter 2

Exchange Server as a Platform for Collaboration

A builder is only as good as his tools: This adage still holds true for developers building successful software applications. As a developer, you require solid tools and technologies, and Microsoft Exchange Server is one of those tools. It provides a number of core capabilities—such as robust messaging functionality, an industrial-strength object database, Internet protocols, and an open directory structure—that make it an ideal platform for your collaborative solutions.

Robust Messaging Infrastructure

Exchange Server provides an infrastructure with certain core services that enable you to focus on building value-added services rather than on re-creating existing services. This infrastructure complements current network topologies and protocols and, as you will see, guarantees that every message gets through to its destination. The following sections discuss some of the advantages of the Exchange Server messaging infrastructure.

Least-Cost Routing, Load Balancing, and Failover

Exchange Server provides technologies in its messaging engine that allow organizations to define different routes of communications between Exchange Servers. Costs can be assigned to these different routes, and the least costly route is always attempted first by the Exchange Server. If this route is unavailable, the Exchange Server will failover to the next least costly route. If you assign the same cost to two different routes, the Exchange Server will distribute the communications traffic evenly over both routes, thereby load balancing the connections.

Let's look at an example. Imagine there are three routes between an Exchange Server in New York and an Exchange Server in California, and the routes consist of one route over the Wide Area Network (WAN), another over a dial-up 28.8 modem, and the third over a satellite link. The administrator of the Exchange Server system can assign costs to each of these routes: the WAN route is assigned a cost of 20, the modem route is assigned a cost of 50, and the satellite route is assigned a cost of 70. Based on the cost of the routes, for communications, the Exchange Server would always attempt the WAN route first. If this route was down, the Exchange Server would failover to the next least costly route (the modem), and if that route was unavailable, it would attempt to connect over the satellite.

Now this is a simple example, but Exchange Server supports the building of very complex routing tables with associated costs that it automatically calculates. For example, consider a message that has to be routed through seven different Exchange Servers until it reaches its final destination. Each Exchange Server has three unique routes to the next server. Exchange Server would automatically find the least costly route of all of the supplied routes.

Delivery and Read Receipts

Exchange Server supports both delivery and read receipts when delivering information through the Exchange Server system. Delivery receipts are returned to an individual user or an application when an item has been delivered to its final destination. This destination can be another Exchange Server or messaging server over the Internet. A delivery receipt also reports the time and date that an item was received by a particular system. You can take advantage of delivery receipts in your application by using them to trigger events when they are returned. For example, a workflow application can consolidate delivery receipts to track the status of message delivery to workflow participants. Figure 2-1 shows an example of a delivery receipt.

Read receipts are similar to delivery receipts, except that read receipts are sent to a user or application when the recipient actually opens the item, and delivery receipts are sent as soon as the item is delivered to the destination server. You might want to use read receipts in your application for time-sensitive items sent through the Exchange Server system. The application could track when the item is read, and

if no action is taken after a certain amount of time, it could reroute the item to a different user or application. Figure 2-2 shows an example of a read receipt.

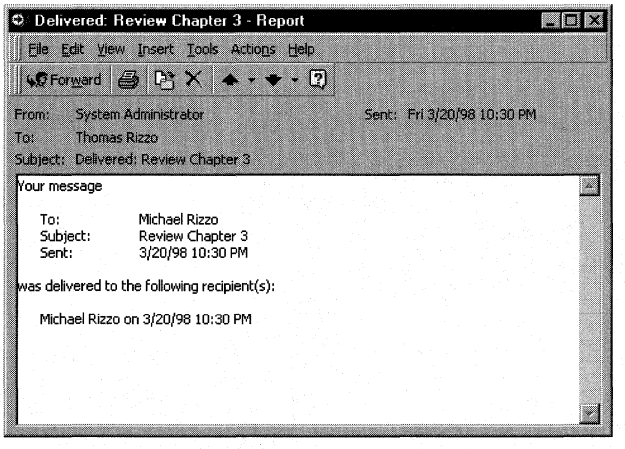


Figure 2-1. *A delivery receipt sent back to a user looks like this. Applications can also send back delivery receipts.*

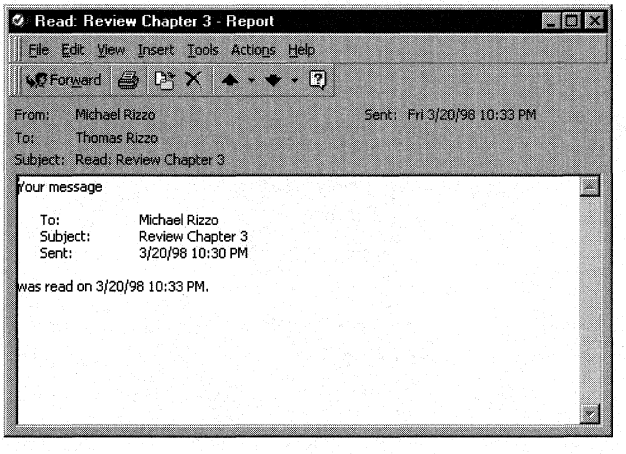


Figure 2-2. *Applications can use read receipts like this to track when users open items sent by the application.*

Message Tracking

Exchange Server supports more than delivery and read receipts. When message tracking is enabled, Exchange Server keeps logs of the items that have entered the Exchange Server system from other systems. Exchange Server also logs where items were routed to, which Exchange Server components routed them, and when the items were delivered to their final destinations. Message tracking enables you to find an item's

route based on specific criteria such as the sender of the item, the intended recipient, or even the component of Exchange Server that handled the message. This powerful tool allows you to trace any item in your application and determine whether or not it reached its destination. Figure 2-3 shows an example of tracing an item in the Exchange Server system.

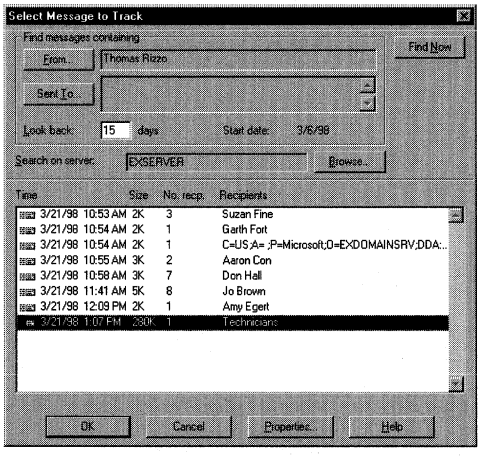


Figure 2-3. *Tracking items from Thomas Rizzo across the Exchange Server system.*

INDUSTRIAL-STRENGTH OBJECT DATABASE

At its core, Exchange Server is an object database. This object database is highly scalable, replicated, built for 24-hour availability 7 days a week, and can hold many different types of objects, including messages, Microsoft Office documents, video files, voice mail, faxes, hyperlinks, text documents, custom forms, and applications such as executable files. You can store all of your application's data in the database while replicating the information to other locations, so the application and its relevant information are available anytime, anywhere. Core features of the Exchange Server database are discussed in the following sections.

Huge Storage Capacity

Many collaborative applications require large amounts of data to be available anytime. Exchange Server makes an excellent repository for this data because it can handle large amounts of information and ensure the reliability and availability of that information. Exchange Server supports very large databases—up to 16 terabytes (16,000 gigabytes) of information. That's pretty big considering that if you compiled every Wall Street transaction in history, you'd have only a little more than 1 terabyte. Really, the only factor limiting the size of your database is the hardware you run

Exchange Server on. The database can run continuously because it has online defragmentation and allows backup programs to work with the database, even when users are logged on.

Exchange Server can store many different types of objects and their associated data in the same database. These objects can even be in the same table, or folder (as tables are called in Exchange Server). Users simply drag and drop different types of objects into these folders, and Exchange Server adds them to the database. This flexibility gives you a distinct advantage when developing applications. Figure 2-4 shows a folder in Microsoft Outlook with many different types of objects.

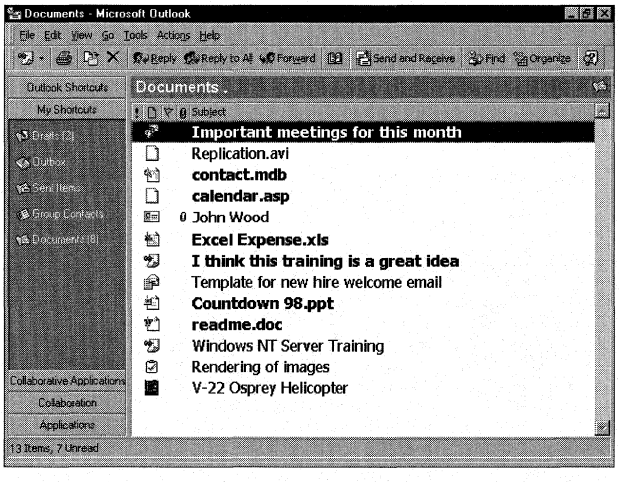


Figure 2-4. *The Exchange Server database is an object database and can hold many types of objects in a single folder.*

Multiple Views

The Exchange Server database not only supports multiple objects in a single folder but multiple views of those objects. You can customize views of the objects stored in the folder by sorting, grouping, and filtering the objects using any combination of their properties. For example, you can customize the view of an Exchange Server folder containing Office documents by specifying Office properties, as shown in Figure 2-5. Even custom properties can be columns in a view, and you can use them to sort and group items in the view.

Exchange Server also supports "per-user" views that allow individual users to create custom views. Exchange Server actually maintains for each user the initial view of the folder, the status of read and unread items, and whether a particular grouping is expanded in the view. Figure 2-6 shows a single view of an Exchange Server folder but for different users. Notice how different the views look. Views can also be replicated offline by using Exchange Server's built-in replication features.

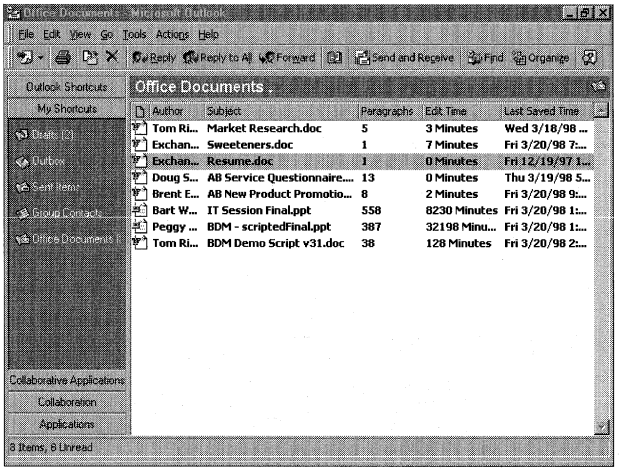


Figure 2-5. *Views support using properties from Office documents. Your applications can use these properties for sorting, grouping, and filtering.*

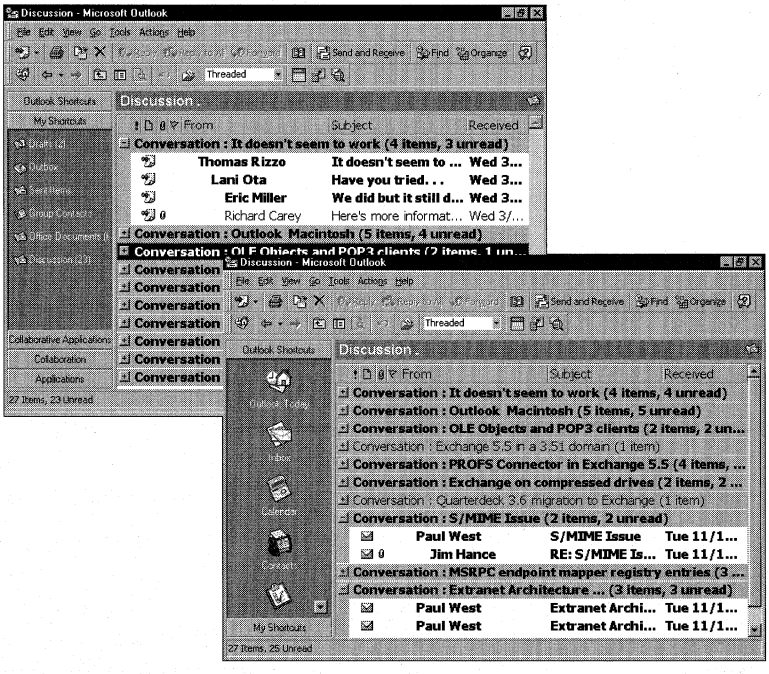


Figure 2-6. *The initial view of the same discussion folder for two different users. Notice that certain groups are expanded and certain items are marked as read.*

Built-In Replication

The Exchange Server database is a replicated database, enabling replication from Exchange Server to Exchange Server and from Exchange Server to Outlook on the client machine. Exchange Server even supports filtered replication between the server and the client.

Replication in Exchange is not the same as simple duplication. Exchange Server replication is more similar to the concept of synchronization in that only the changes are sent to replicas in the system. Sending only the changes, as opposed to copying the entire folder for each replication cycle, saves not only time but network bandwidth.

Setting up server-to-server replication with Exchange Server is easy. All the administrator has to do is select the folder to be replicated and then select the server to replicate the folder to. The settings that enable server-to-server replication in the Microsoft Exchange Administrator program are shown in Figure 2-7. Once these settings are in place, the actual replication messages are sent over the Exchange Server messaging infrastructure. This allows the replication messages to leverage Exchange Server's load balancing, least-cost routing, and failover capabilities. Exchange Server also supports setting the time and size limits of the replication messages.

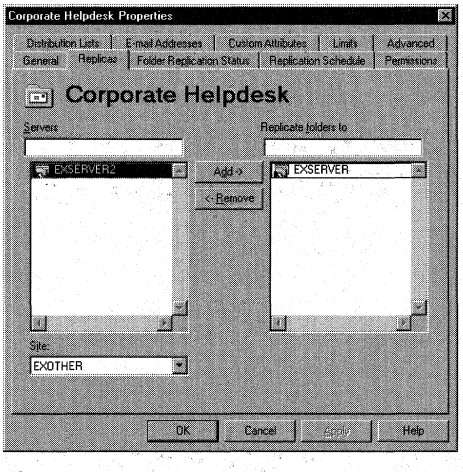


Figure 2-7. *Setting up server-to-server replication for your applications in Exchange Server is as easy as pointing and clicking in the Exchange Administrator program.*

The Exchange Server replication feature has built-in conflict management capabilities that enable users to edit the same information at the same time in the same folder or even in replicas of a folder in different locations. To determine which item to accept as the newest, Exchange Server implements "last saved wins," the process

of querying the time an item was saved and retaining the most recently saved item. You can also set an option that alerts users via e-mail when items are in conflict. Both versions of the item are sent to these users, and they can decide which item is the most up-to-date. Exchange Server will keep the item they select.

For server-to-client replication, Exchange Server and Outlook support bidirectional synchronization of changes to information in Exchange Server folders. This synchronization occurs in Outlook as a background process, so users can continue working in Outlook. The synchronization can be scheduled so that it happens at certain intervals. For example, a user can configure Outlook replication so that every 30 minutes the Outlook client synchronizes its local database with new information from the Exchange Server.

Outlook also supports filtered replication, in which only a subset of information is synchronized to the local database. Filtered replication is most useful to users when large amounts of data are available in the Exchange Server database but users want to take only a subset of that data offline. For example, imagine an Exchange Server folder with 50,000 sales contacts. A typical user wouldn't be able to accommodate the entire folder on her local hard disk, so she could set the replication criterion to only those contacts for whom she is the sales representative. Instead of 50,000 contacts, the filtered subset is 1,000 contacts. Figure 2-8 shows the interface in Outlook where users can set the criteria for filtered replication.

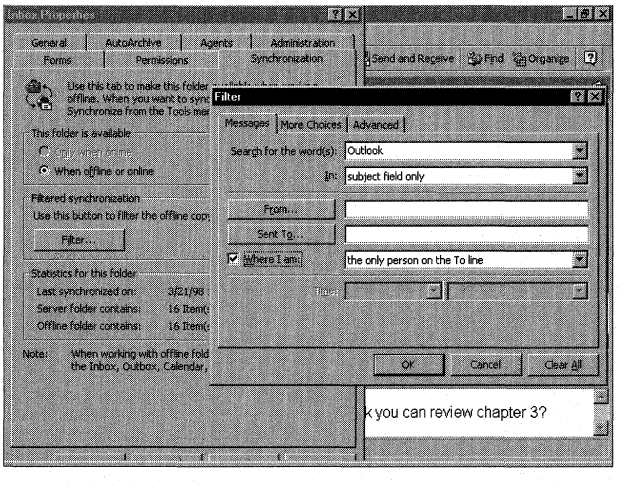


Figure 2-8. *Setting up filtered replication in Outlook is easy for users of your application.*

Schema Flexibility

Typically, when you begin work on an application that deals with a database, you are forced to plan your schema for the database before you start writing your application. If the application requirements change and a new field has to be added to the database, the schema might not be flexible enough to support the addition, and you might have to drop the present database and create a new one.

With the Exchange Server database, however, you can add new fields at any point in development, which allows you to accommodate the changing requirements of an application. New fields are automatically available to users, so users can create custom views using them.

Transaction Logging

A transaction is a unit of work, such as adding an item to an Exchange Server database. Before any item is committed to the Exchange Server database, the transaction is written to a transaction log file and then to the database. This process is called write-ahead transaction logging, and it guarantees that no item will be lost.

Transaction logs allow the Exchange Server to recover the database after some form of failure, such as a power loss. In this type of scenario, after power is restored and the server is rebooted, the Exchange Server automatically recovers the database. Using checkpoints in the transaction logs, the Exchange Server replays any transactions that were not committed to the database before the power failure.

The transaction log is an inherent feature of the Exchange Server database, so any application you develop on Exchange Server can take advantage of it. Any items your application sends or stores in the Exchange Server system will be delivered or committed, even in the event of certain failures in the computer system or network.

EXCHANGE SERVER DIRECTORY

To collaborate effectively, users must be able to find other users and information easily. Exchange Server provides a hierarchical directory for this purpose. This directory holds the critical information of an organization, and it can meet the needs of both large and small organizations because it's scalable and easy to manage. Some of the most important features of the Exchange Server directory are described in the following sections.

Reliable Database Engine

The Exchange Server directory is implemented using the same database technology as the Exchange Server messaging infrastructure, so the database engine's reliability is high. This reliability guarantees that the directory will always be available to your applications.

Multimaster and Replication Capabilities

The Exchange Server directory is a multimaster, replicated directory. A multimaster directory allows an administrator to make changes to it on any Exchange Server in the organization, changes that Exchange Server then propagates to other servers through replication. Directory replication is implemented over the messaging infrastructure of Exchange Server, so directory-replicated messages can take advantage of the least-cost routing, failover, and load-balancing features of Exchange Server.

Directory replication in the Exchange Server system is not limited only to server-to-server replication. Exchange Server also supports server-to-client directory replication. By using a feature called the Offline Address Book, Outlook can replicate the Exchange Server directory, or a subset of it, to a user's local machine. This allows a user of your application to address items to other users and to look up detailed directory information, even when the user is working offline.

Customizable Attributes and "White Pages"

Exchange Server exposes a number of attributes in the directory that you can customize and replicate. For example, you could customize the Exchange Server directory with a field named "cost center," and set up a supplies requisition program that dynamically queried the directory for users ordering supplies. Based on what information users entered in the cost center field, the application would update an accounting system so that the cost of supplies are automatically deducted from the cost center. Figure 2-9 shows where you can customize the Exchange Server directory.

The directory has some additional built-in features that you can take advantage of, such as its ability to store all types of information about an organization, including users' office locations, phone numbers, department names, titles—even a user's manager and direct reports. Exchange Server is an ideal "white pages."

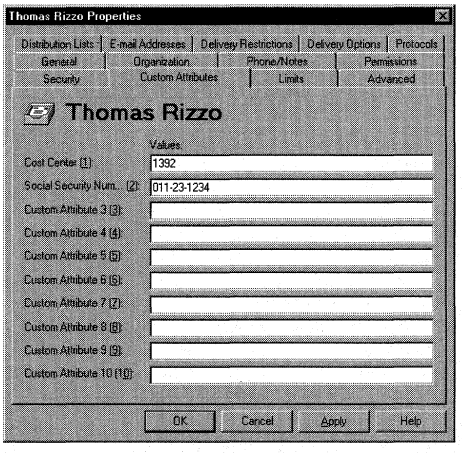


Figure 2-9. *Customizing attributes in the directory. Your applications can take advantage of these customized attributes.*

For workflow applications, a central, hierarchical directory of this kind is crucial. Workflow applications must be able to route items based on an organization's staff structure, which is dynamic. If names of individuals were hard-coded in an application, staffing changes would require the application to be rewritten. With the Exchange Server directory, you can query and dynamically generate employee information.

Extensibility and Security

The Exchange Server directory is not limited to storing information for only one organization. Through the use of custom recipients, the Exchange Server directory can also hold address and organizational information for users from other organizations. The Exchange Server directory exposes the same functionality to these types of directory objects as it does to the standard directory objects. Figure 2-10 shows an example of a custom recipient in the Exchange Server directory.

Any directory object in the Exchange Server system can be secured by using access permissions, which determine who can see particular objects in the directory. For example, an administrator can set the access permissions on the business partner directory entries so that certain workers are denied access. These permissions can be set either per user or per group.

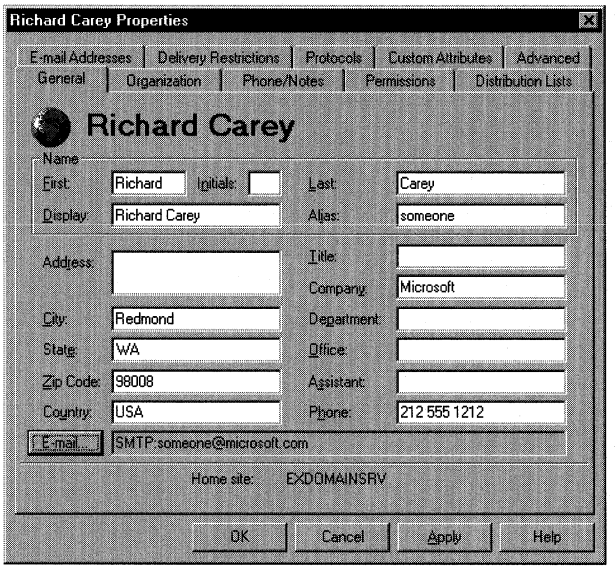


Figure 2-10. *A custom recipient in the Exchange Server directory. Recipients can hold organizational information for users outside your current organization.*

Internet and Industry Standards Support

The Exchange Server directory supports Internet standards such as LDAP version 3. LDAP, which stands for Lightweight Directory Access Protocol, is an adapted subset of the X.500 standard that specifies a common protocol for directory access over TCP/IP. The key benefit of LDAP support in Exchange Server is that any LDAP-compliant client or application can query the Exchange Server directory. LDAP version 3 as implemented in Exchange Server enables you to chain directories together through a feature called referrals. Referrals tell the Exchange Server directory where to look for information that a user is querying for when the directory does not currently possess it. For an application, referrals are crucial since one directory might not contain all the needed information about users and services. Rather, many different directories, which could be hosted on servers in different locations and even in different organizations, might contain pieces of this information.

The Exchange Server directory supports ADSI (Active Directory Services Interface). ADSI is an application programming interface that enables you to modify many different directories using standard protocols. The different directories that ADSI supports are the Active Directory in Microsoft Windows 2000, the Microsoft Windows NT version 4 domain-based directory, any LDAP-compliant directory such as Exchange Server directory, Novell NetWare's NDS Directory, and Novell NetWare Bindery. The ADSI interface abstracts the low-level functions of these directories and exposes a

number of objects with which you can write applications. Because ADSI provides COM interfaces that give every directory element a common set of properties, the application can use the same programming interface to connect to directory elements in several directory services. Figure 2-11 shows a diagram of ADSI and the directory services it can access. ADSI is an important technology to learn since it ties all of these disparate directories together with a common programming model, and it is Microsoft's strategic directory programming interface. Chapter 15 demonstrates how to program to an Exchange Server directory using ADSI.

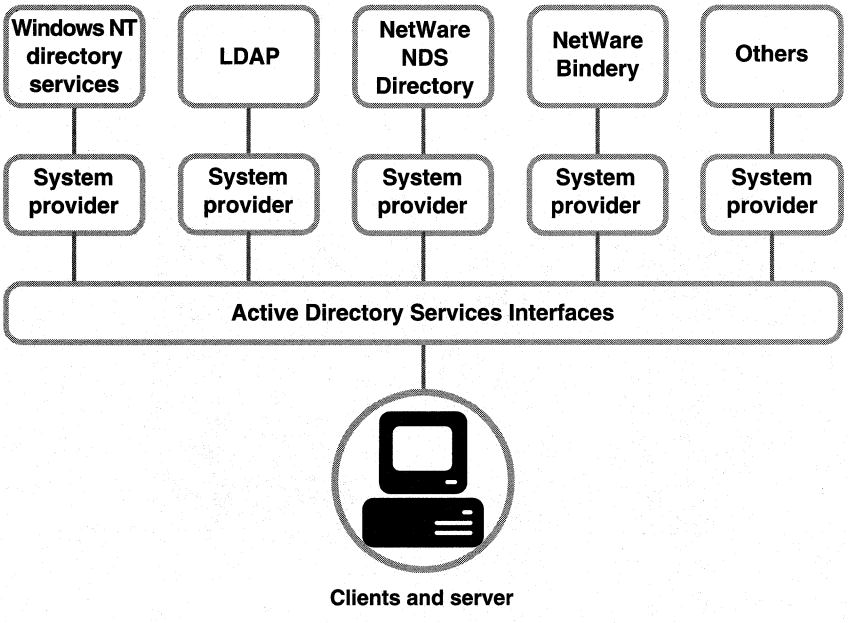


Figure 2-11. *ADSI allows you to talk to many different directories, including Exchange Server, using the same interfaces. This access is provided through the different system providers (SPs) in ADSI.*

PUBLIC FOLDERS

The core of Exchange Server's collaborative technologies is a feature called public folders. Public folders are repositories for all kinds of information users will share. They can be accessed by many types of clients, using various protocols to communicate. Public folders can contain custom forms for contributing or reviewing information in the folder, and users can create custom views for organizing and filtering the information in the folder. The main features of public folders for developers are described in the following sections.

Public folders, just like the Exchange Server directory, are built on the Exchange Server object database, so they take advantage of its architecture and enjoy the same benefits:

- Flexible database schema
- Server-to-server, server-to-client, and filtered replication

Folder and Application Accessibility

Public folders in Outlook are arranged in a hierarchical tree view, as shown in Figure 2-12. As you can see, this arrangement makes it easy for users to scroll and find information. This hierarchy is actually a virtual view of public folder replicas in the Exchange Server system; users don't have to know on which server the public folders actually exist.

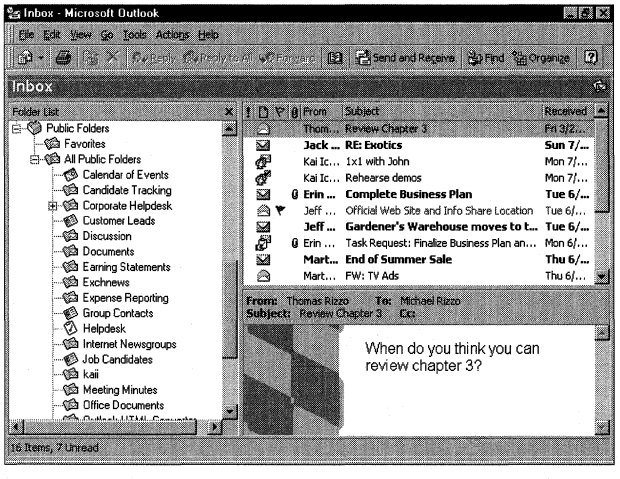


Figure 2-12. *The hierarchical tree view of the Public Folder allows users to quickly find information.*

Exchange Server can assign costs to different sites so that users connecting to a remote site with a public folder follow the least costly route to that site. This assignment of costs to remote connections for public folders is called public folder affinity. Figure 2-13 shows the administration interface for public folder affinity.

Public folders are not limited to holding only the data for an application; they also can hold any custom forms associated with the application. The availability of these forms makes your application easier to use by allowing users to go directly to a public folder to select the associated form rather than search for the form in a global forms list.

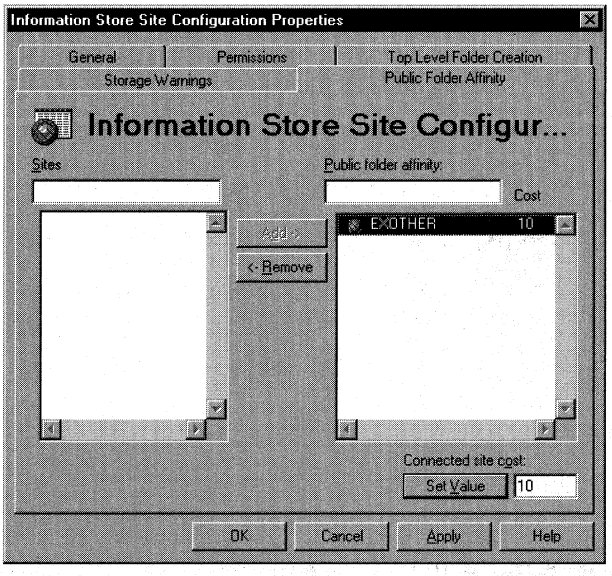


Figure 2-13. *By setting the Public Folder Affinity option, users of your application will access one replica of the Public Folder database over another depending on their location on the network.*

Security and Content Control

Inherent in public folders is security control. Public folder permissions can be set on three different scales:

- *Global.* Default permissions for everyone in the organization; default permissions for anonymous users
- *Group.* Permissions for a specific list of users
- *Per user.* Individual permissions for a particular folder

All of these permission levels can be combined for a particular folder or set of users. Assigning permissions is easy, as shown in Figure 2-14. Notice that the permissions tab supports predefined roles for users, which you can use to quickly set permissions for a folder.

In addition to having roles-based permissions, public folders have built-in moderation capabilities. A moderated public folder allows you to control what content is posted to a folder and who has permission to approve this content. Before any item is posted to a folder, the item is mailed to the selected moderators, who approve the content. You can quickly set up a moderated public folder in the folder properties.

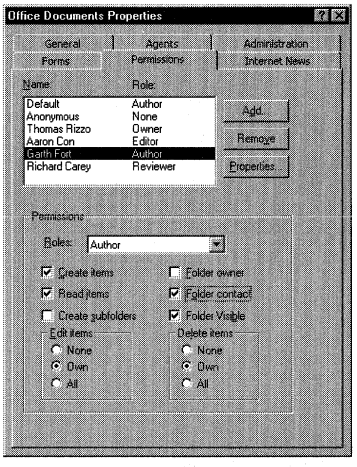


Figure 2-14. *Setting permissions for a public folder is easy.*

Public folders support the e-mailing of items into a public folder, which makes information available to many users, cuts down on e-mail traffic, and saves disk space. Mailing-list server applications and distribution list applications can really take advantage of public folders. By default, the e-mail address of the public folder is hidden from the address book, but the public folder can be exposed in the address book so that users can browse for its e-mail address.

Internet Standards Support

As you've seen, significant economies can be achieved when information is stored in a central location rather than in individual mailboxes. By using Exchange Server public folders as the central location, organizations can expose information to any standard Internet client that supports the Network News Transfer Protocol (NNTP), the Internet Mail Access Protocol version 4 (IMAP4), or the Hypertext Transfer Protocol (HTTP). These Internet clients can post and read information securely from a public folder. More importantly, these protocols allow users who do not have Outlook on their machines to take advantage of the functionality of an Exchange Server public folder. For example, an organization can set up a customer service public folder that enables internal users to employ Outlook to view folder information and allows external users to choose from several clients, including Outlook, an NNTP newsreader such as Outlook Express, a standard web browser, or even an IMAP4 client such as Netscape Communicator. Exchange Server exposes its collaborative functionality to all of these clients. Let's take a look at some specific information on the Internet protocols supported in Exchange Server.

NNTP

NNTP is an internet standard that defines server-to-server replication of data in the form of articles. These articles exist in a hierarchy of newsgroups, which are similar to discussion folders in Exchange Server. Users can replicate the articles offline, plus the articles are presented in a threaded view so that users can view their history. Exchange Server supports both NNTP server-to-server replication and the ability of any standard NNTP client to read information in Exchange Server public folders. This allows any public folder in Exchange Server to be replicated to another NNTP server or read by an NNTP client. Organizations can use this feature to expose public folders and their information to their customers. Figure 2-15 shows an example of a newsreader using NNTP to access an Exchange Server public folder.

IMAP4

IMAP4 is an Internet standard that defines a way for clients to access messaging information on a server. Exchange Server is an IMAP4-compliant messaging server, so any standard IMAP4 client can access the messaging services of Exchange Server. Some of these services include sending and receiving e-mail, synchronizing e-mail to offline storage, and accessing public folders. Accessing public folders with IMAP4 extends the power of public folders to any standard IMAP4 client.

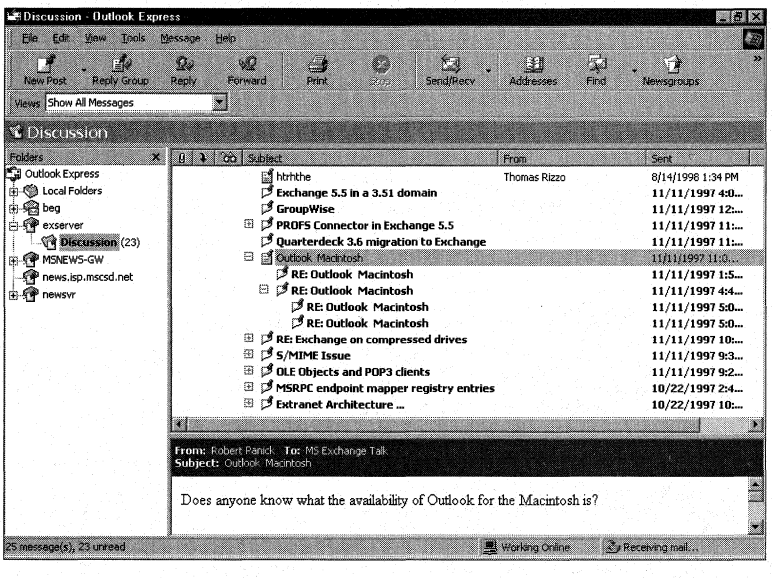


Figure 2-15. *A public folder being viewed by Outlook Express, an NNTP newsreader. Exchange Server folders have built-in support for NNTP.*

HTTP

HTTP is the primary protocol used to distribute information on the World Wide Web, that is, to transmit graphics and documents from a web server to a web browser. HTTP is a client-to-server protocol, meaning that a client running on the user's machine sends to a server a request for data, and the server receives the request and sends the relevant information back to the client. HTTP servers can do more than just send back simple data—scripts that access other back-end services on the network can run on the HTTP server. These services can be databases, collaboration servers such as Exchange Server, or custom-built applications.

One example of an application that uses the HTTP protocol to access Exchange Server information is Microsoft Outlook Web Access. Outlook Web Access, which we discuss in Chapter 7, is an application that allows any standard web browser to access the information stored inside an Exchange Server. Outlook Web Access is built using Microsoft Collaboration Data Objects (CDO). CDO, which we look at in detail in Chapter 12, is a set of COM objects that exposes the services of Exchange Server to any COM-based development tool. CDO is the object library for Outlook Web Access, and Microsoft Internet Information Services (IIS)—especially Microsoft Active Server Pages (ASP), VBScript, and JScript—are the development tools. Figure 2-16 shows the architecture for Outlook Web Access.

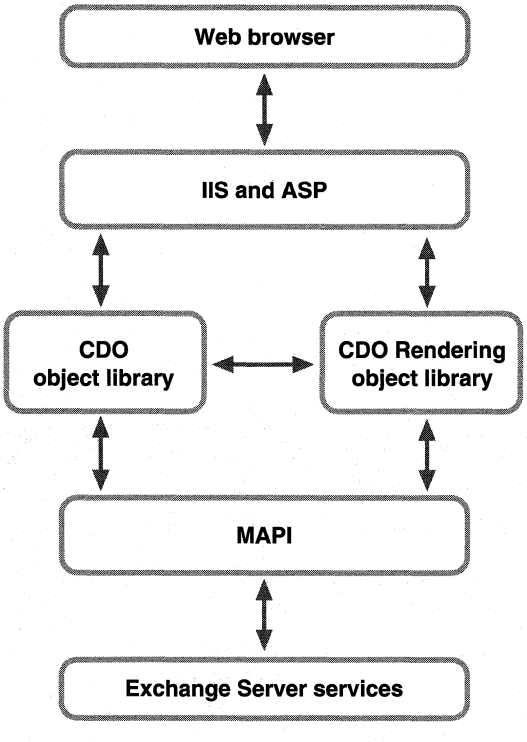


Figure 2-16. *The Microsoft Outlook Web Access architecture.*

Active Server Pages allows you to write scripts using any standard ActiveX scripting language. With these scripts, which execute on IIS and return HTML to the web browser, you can build dynamic web applications that take advantage of objects on the web server. The following code listing shows a simple Active Server Pages application:

```
<HTML>
<HEAD>
<TITLE>Hello World!</TITLE>
</HEAD>
<BODY>
<H1> Hello, I was created on <%= Now() %>.<P>
<% Set BrowserControl = Server.CreateObject("MSWC.BrowserType") %>
You're using <I>
<%= BrowserControl.browser & " " & BrowserControl.Version & " " %> </I>
as your web browser.
<% set BrowserControl = Nothing %> </H1>
</BODY>
</HTML>
```

If you browsed this web page using Microsoft Internet Explorer version 5, you would see the image in Figure 2-17. Notice that none of the script is sent back to the client, only the text, and that the date in the text is generated dynamically from the system date on the web server. The browser is also detected by using a component on the web server. Chapter 7 discusses the ASP object model and programming environment in more detail.

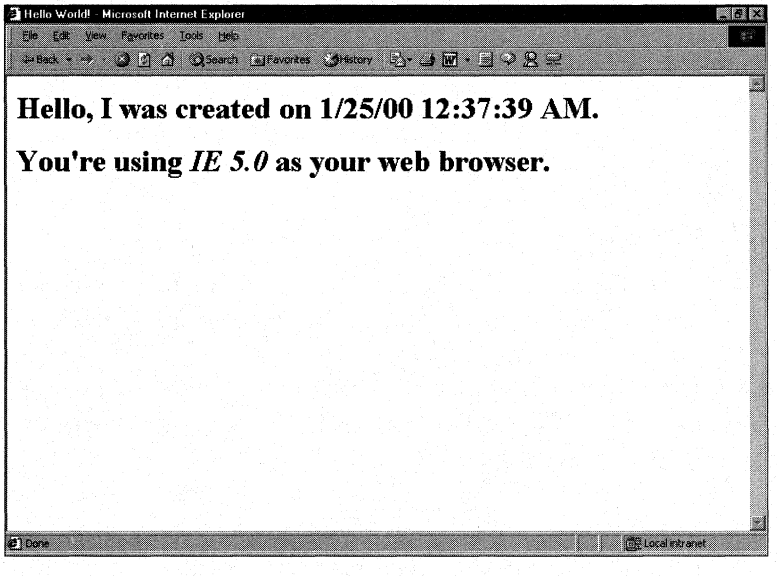


Figure 2-17. *Results of browsing the ASP page using Internet Explorer.*

By using Active Server Pages, Outlook Web Access can dynamically and securely create web pages based on the information and services in Exchange Server, such as a user's mailbox, calendar, and contacts; and messaging and calendaring services. You can access the same features in your application since the enabling technology for Outlook Web Access is the CDO library.

INTEGRATED, INTERNET STANDARDS-BASED SECURITY

With so many corporations connecting their systems to the Internet and exposing their networks to millions of Internet users, security has become a large concern. While most users on the Internet are not lurking and waiting to break into corporate networks, some "bad apples" on the Internet are. Exchange Server prevents these users from accessing privileged information by implementing Internet standards-based security in an integrated way.

Windows NT Security

Exchange Server integrates with Windows NT security in two ways. First, users have to be authenticated using a Windows NT account before gaining access to any Exchange Server resource that requires authenticated access. Administrators can set up a Windows NT security infrastructure, and Exchange Server will use that infrastructure for its own security and access permissions. This enables users to log on only once to access both the network and Exchange Server services.

Second, Exchange Server uses the built-in auditing capabilities of Microsoft Windows NT. This integration allows an administrator to detect security breaches by tracking events, across Windows NT and Exchange Server, which occur within a system. All the events can be viewed in one window using the Windows NT event log.

Secure Messaging

Many corporations today use the Internet as a backbone for their corporate communications system. While this is cheaper than leasing lines between servers, it opens a world of security concerns. Exchange Server alleviates these concerns by implementing some key features that allow corporations to securely use the Internet as a communications network backbone. For securely sending messages between servers, Exchange Server supports Secure Socket Layers (SSL) in combination with the Simple Mail Transfer Protocol (SMTP). SMTP is the primary way that different mail systems talk over the Internet. SSL allows systems to encrypt data sent from one system to the other. By implementing SSL with SMTP, an organization can encrypt its data from one Exchange Server to another when sending the data over the Internet.

Secure Applications

SSL is not only supported with use of SMTP, but it is also used with other Internet protocols that Exchange Server supports. By using SSL, Outlook Web Access can encrypt any traffic between a user's web browser and web server. This secures any HTML documents that Outlook Web Access is sending to the user. You can take advantage of SSL when using custom forms in the web forms library of Outlook Web Access.

S/MIME Support

Exchange Server supports encryption and digital signatures by using Secure Multipurpose Internet Mail Extensions, or S/MIME for short. An Internet standard, S/MIME is a method of digitally signing and encrypting messages between users on the same vendor's system or users on different vendors' systems.

S/MIME is built on X.509 version 3 certificates. These certificates are generated by a certificate authority such as VeriSign or Certificate Server included with the Microsoft Windows 2000 Server. Since Exchange Server supports X.509 version 3 certificates, it can accept the certificates from other certificate authorities. Similarly, clients can trust certificates from other authorities through the use of Certificate Trust Lists.

Exchange Server also supports the revocation of security certificates. Revoking certificates is useful when a user feels that her security has been compromised and someone else is signing messages on her behalf. Likewise, when a user leaves an organization, you might want to revoke the user's certificate to make sure that all messages sent by this user are marked invalid. When an administrator revokes the certificates for a user, any encrypted messages previously sent by that user will notify other users, upon opening of the messages, that the certificate is invalid. After revoking a certificate, the administrator can issue a new certificate to the user.

As a developer, you can take advantage of the advanced security features of Exchange Server. By building your applications based on the standard Outlook e-mail message, you automatically inherit the advanced security functionality in Outlook. This allows you to digitally sign and encrypt your custom forms before the user sends or posts forms.

Multitiered, Replicated, Secure Forms Library

Locating new applications in an organization can be a hard thing to do because they exist in so many places. For example, an application can exist on one of many possible file servers. Although the emergence of intranets has enhanced the ability to find

applications, you still have to find the site with explicit links to the information you want. And if you do find the web server that has the application, you might have to connect to a server halfway around the globe, making connection speeds to that application very slow.

Exchange Server's multitiered, replicated, secure forms library makes it easier to locate applications. The Exchange Server forms library is divided into four main components: an Organizational Forms Library, the folder forms libraries, a Personal Forms Library, and a web forms library. Some of these libraries can be synchronized offline, so users can work with the applications, even when the users are disconnected from the network. You can choose which of these is best for your application.

Organizational Forms Library

The Organizational Forms Library contains, most often, forms that everyone in an organization needs access to, such as vacation requests, business cards, and travel expense reports. The Organizational Forms Library is contained on the Exchange Server and can be replicated to servers throughout your network, so access to these forms is fast. It lists all the available applications throughout an organization. Figure 2-18 shows an Organizational Forms Library in Microsoft Outlook 98.

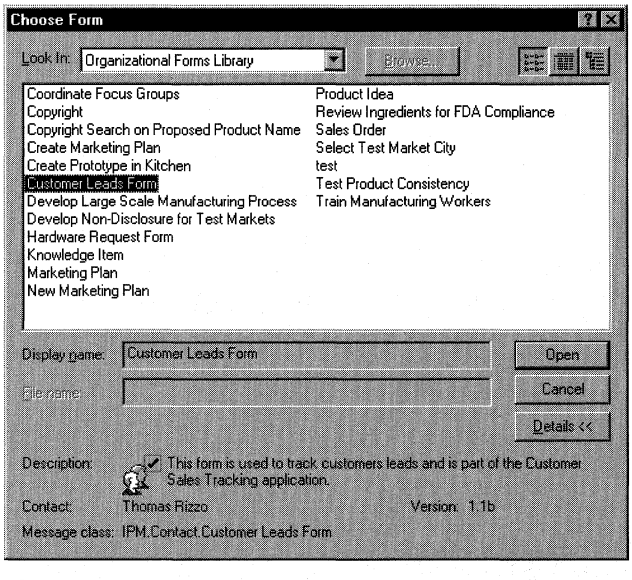


Figure 2-18. *You can publish your application in the Organizational Forms Library, and it will automatically be available to your users.*

The Organizational Forms Library is secure, so administrators can set which users have permissions to publish or edit information in the forms library. It is also multilingual; the Exchange Server presents the server-based forms library that corresponds to the language of the client program accessing the forms library. For example, when a Japanese client requests a list of forms in the Organizational Forms Library, the Exchange Server displays all the corresponding Japanese forms. This multilingual capability allows you to customize and deploy your applications to the correct client without writing any code.

Folder Forms Library

A folder forms library is for folder-specific forms. The folder forms library is more secure than the Organizational Forms Library. You would post forms you do not want to share globally in the folder forms library. The forms stored in a *personal* folder forms library are shared only with the users to whom you give access. The forms stored in a *public* folder forms library can be shared with any user who has the correct permission on that public folder. Using the synchronization capabilities of Exchange Server, users can replicate public folders (including their data and forms) offline.

Personal Forms Library

The Personal Forms Library is the most restrictive in terms of sharing its forms with other users; this library "belongs" to a particular user and cannot be shared with any other user in the organization. All forms in the Personal Forms Library can be used both on and off the network. Users can test forms in the Personal Forms Library before publishing them to the Organizational Forms Library or folder forms library.

Web Forms Library

The web forms library is a hierarchy of folders stored in the Windows NT file system where your web server, IIS, runs Outlook Web Access. Exchange Server supports HTML forms as a development environment, so Outlook Web Access has an easy and automatic way for web developers to publish custom forms in the web forms library. To create an HTML-based application, you need only to create a subdirectory in the file system where Outlook Web Access is stored and copy your HTML files to it. The new form will appear in the Launch Custom Forms window of Outlook Web Access. Users can then start working with the application from the web forms library. Figure 2-19 on the following page shows forms in the web forms library.

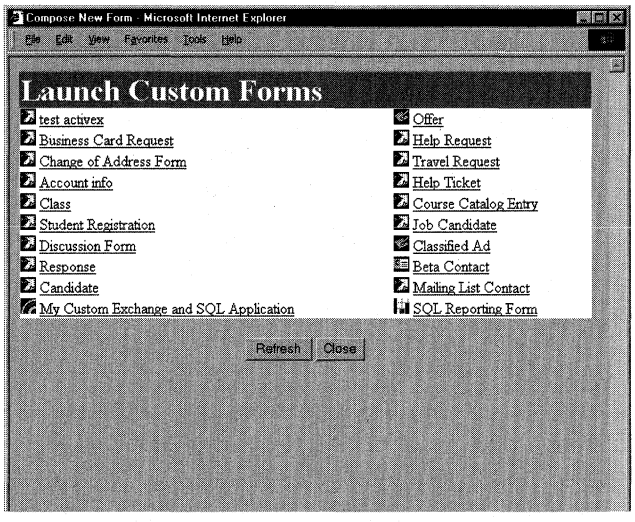


Figure 2-19. *The web forms library holds HTML applications that you develop for your organization.*

BUILT-IN INFORMATION MANAGEMENT TOOLS

Managing information when building applications can be one of the most tedious tasks for a developer. But public folders, with their built-in and configurable services, handle these tasks automatically for you. They allow you to set the expiration time for information, which prevents public folders from becoming inundated with megabytes of outdated and useless information. Their conflict management features prevent two users from unintentionally saving two versions of the same document. If two users edit the same document stored in a public folder and then try to save their changes, Exchange Server sends a conflict message to both users and to any folder contacts defined on the public folder. The users then have the choice to keep one of the two items or both. Figure 2-20 shows the Conflict Message dialog box.

Rules

To manage the massive amount of information received by an organization, Exchange Server supports rules. Although many other collaborative systems also have this functionality in some form, in most cases a user must be logged on to the system before the rules can be processed. Also, other systems don't allow rules in folders other than a user's personal folders. With Exchange Server, rules are supported for both personal folders such as an Inbox and for public folders.

By setting rules in your public folder application, you can to some extent control the flow of information into and out of it. Public folder rules are configurable by

the owner of the public folder and are server-based, which means no client has to be logged on for the rules to fire. Instead, the server fires the rules.

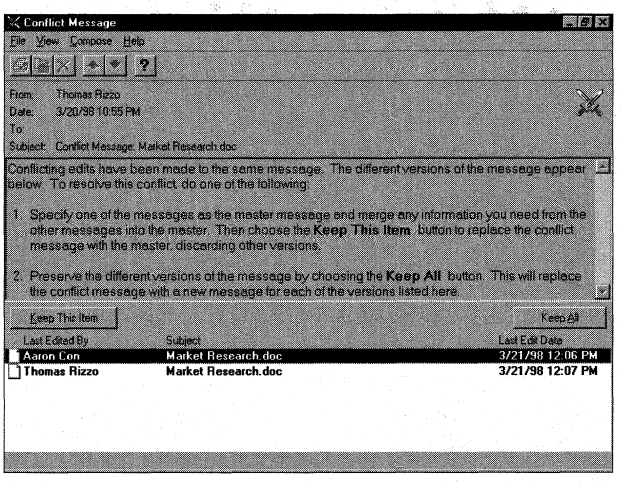


Figure 2-20. *Exchange Server automatically detects when conflicts of information occur in your applications. The server will then send a notification to the folder owner and to the users who generated the conflict.*

The types of rules you can create range from simple rules, such as "send a thank you e-mail to anyone who sends a message to the public folder," to very complex rules. Complex rules can entail checking multiple fields on an item and taking a specific action based on those fields.

Event Scripting Agent

Sometimes rules are not the best strategy for controlling the flow of information in your application—for example, they might be too constrictive. In these situations, a feature in Microsoft Exchange Server version 5.5, called the Microsoft Exchange Event Scripting Agent, will help you greatly. The Event Scripting Agent (discussed in more detail in Chapter 13) allows you to write custom event handlers for the most common Exchange Server folder events by using standard development tools that support COM. For example, you can write a script event handler that calls COM objects that you create. Plus, as its name implies, the Event Scripting Agent ships with a scripting engine that understands any ActiveX scripting language, so you can write your custom agents using a scripting language such as VBScript or JScript. You choose which development tool you use to write these agents.

Once you write your custom agent, you can place it in an Exchange Server folder, such as your server-based Inbox or a public folder. Your agent can handle four distinct events, listed on the next page.

- *OnMessageCreated.* This event fires when any type of new item is posted to the folder, such as an e-mail message, a calendar appointment, or a Microsoft Word document. This event can be generated from any type of client, such as Outlook or a web browser client. An expense report agent might use this event to look up the manager of a person who submits an expense report and then route the report to the manager for approval.
- *OnChanged.* This event fires when any type of item is edited and saved back into the folder. An example agent for this type of event is a resource scheduling agent, which monitors a public folder calendar for conference rooms. When a meeting time is changed, the agent notifies all the attendees and any catering services.
- *OnMessageDeleted.* This event fires whenever an item is deleted from the folder. It's useful when you want to synchronize the contents of a folder with another data source. By writing a custom agent for this event, you could delete from other folders or databases items that are related to the deleted item.
- *OnTimer.* This event fires based on a time limit you specify, which can be weekly, daily, hourly, or on a more granular time. For example, you can customize an event so that it fires every 15 minutes starting at 6:00 P.M. and ending at 3:00 A.M., or set the event to fire only on certain days. An example of an agent using this event is another expense report agent that works in conjunction with the sample expense report agent we just discussed. Suppose the manager does not approve the expense report forwarded by the first expense agent in the specified amount of time—an hour, let's say. A scheduled event, created to check pending expense reports every hour, determines that the expense report needs to be escalated to the manager's manager. The agent could look up that individual using the Exchange Server directory and route the expense report to her.

As you can see, by creating custom agents, you can implement custom functionality that would otherwise be unavailable in public folder rules.

CONNECTIVITY AND MIGRATION TOOLS

Information in a corporation is stored in various places. For employees, business partners, and customers to collaborate effectively, these "islands" of information must become connected, and information must be accessible. To enable this, Microsoft Exchange Server has a number of built-in migration and coexistence tools.

A series of connectors enables an Exchange Server to coexist with other types of collaborative systems. These connectors ensure the reliable delivery of messages between Exchange Server and these other systems, but the connector's capabilities do not stop there. The connector can also provide directory synchronization between the two systems. Directory synchronization gives clients on both systems the ability to seamlessly query the directory for users on another system. This global, unified directory in the Exchange Server system makes building collaborative applications easier because it centralizes information. The systems that Microsoft Exchange Server can connect to, send messages from, and synchronize directories with include Microsoft Mail, Lotus cc:Mail, and Lotus Notes. Exchange Server can transfer messages with host-based systems, such as OfficeVision VM (PROFS), and System Network Architecture Distribution Services (SNADS) systems, such as IBM OfficeVision/MVS and Fisher TAO.

Sometimes, corporations find it more cost-effective to have only one collaborative system rather than several. To help organizations move to Exchange Server, migration tools for a large number of systems are included in the product. These tools make it easier for organizations to transfer their users and information into the Exchange Server system. The products supported by these migration tools are Microsoft Mail, Lotus cc:Mail, Novell Groupwise, Collabra Share, and Lotus Notes.

CLIENT OPTIONS

The Microsoft Outlook family of clients provides users with a choice of clients to use with Exchange Server. These clients support multiple platforms and provide varying levels of functionality, depending on the needs of the user. In addition, all the clients in the family support a consistent user interface, so moving from one client to another is easy. The following sections describe these clients.

Pocket Outlook

Microsoft Pocket Outlook runs on any handheld device that supports the Microsoft Windows CE version 2 operating system. Pocket Outlook enables communication and enhances collaborative work by supporting e-mail, contacts, tasks, and scheduling. These services can be synchronized with Outlook 2000 by using the built-in ActiveSync technology in Microsoft Windows CE.

Outlook Express

Microsoft Outlook Express is a POP3, IMAP4, and NNTP client that ships for free with Microsoft Internet Explorer version 5. Outlook Express provides basic e-mail and newsgroup functionality that can be customized to meet the needs of the user. When used in conjunction with Exchange Server, Outlook Express has simple calendaring functionality.

Outlook Web Access

Outlook Web Access is a browser-based view of information stored in Exchange Server and is covered in Chapter 7. Outlook Web Access supports e-mail, calendars, public folders, custom views, and directory functions, all from a standard web browser. The technology behind it comprises ASP and CDO.

Outlook for Microsoft Windows Versions 3.x and the Macintosh

For companies supporting employees who use 16-bit Windows and the Macintosh, Microsoft Outlook offers consistent versions for both platforms. Both provide the same user interface as the other members of the Outlook family and include e-mail, personal calendaring, task lists, group scheduling, HTML-based custom forms, and an easy migration path from current Microsoft e-mail clients.

Microsoft Outlook

Outlook is Microsoft's premier e-mail and collaboration client. With Outlook, users can manage many types of information including their e-mail, personal calendar, contacts, and tasks. Group scheduling, task management, and journal capabilities improve user productivity through better information management. Tight integration with Office and Internet Explorer provide major benefits to users of these applications because Microsoft Outlook extends them with enhanced functionality.

CHOOSING A CLIENT

As a developer preparing to build a messaging, tracking, workflow, real-time, or knowledge management application for Exchange Server, you must first ask: "Which client interface should I use—Microsoft Outlook or a web browser?" The answer depends on a number of factors.

You need to ask yourself a few questions. For example, does the application need offline support? If the answer is yes, consider using Outlook, which has built-in support for offline forms, and which automatically synchronizes any offline changes to Exchange Server. Compare this functionality to the web browser client, which has limited support for offline forms. Web browsers can cache web pages for offline viewing, but they cannot process server-side scripts such as Active Server Pages without a web server on the local machine. The typical user does not have a web server on a local machine.

Support for non-Win32 clients, such as Windows 3.1, Microsoft Windows for Workgroups, Macintosh, and UNIX, is another factor to look at when designing applications. The ubiquity of web browsers for multiple operating systems has enabled

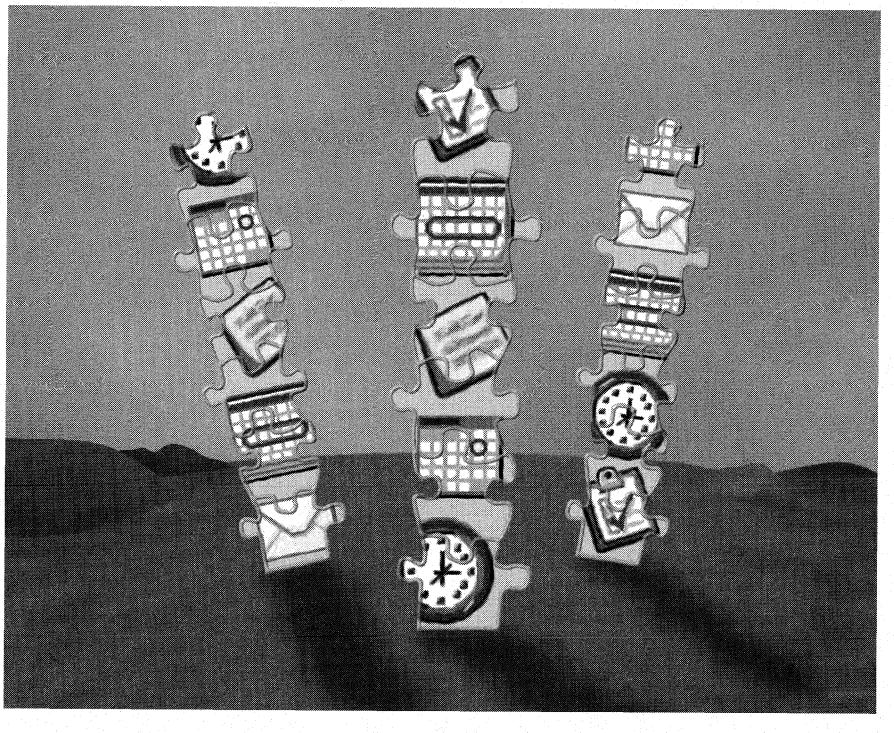
web-based applications to provide cross-platform client support. Although Microsoft Outlook runs only on Windows 95, Windows 98, Windows NT, and Windows 2000, new technology enables Outlook forms to be converted to web-based applications. This technology, called the Outlook HTML Forms Converter, is discussed in detail in Chapter 7.

Your skill as a developer is another factor to consider. Microsoft Outlook offers a very approachable development environment, even for the novice developer. It also provides built-in capabilities that allow power users or developers to customize an application without writing any code—very appealing to those who want to meet a specific need quickly and avoid creating an application from scratch.

If you are more familiar with other development tools, consider that Exchange Server exposes its services through a rich API that can be called from any development environment that supports COM. Some examples of these development environments include Microsoft Visual Basic, Microsoft Office (through Microsoft Visual Basic for Applications), Microsoft Visual C++, and Microsoft Visual J++. This environment flexibility allows you to leverage the tools and skills you currently have.

Part II

Building Outlook Applications



Chapter 3

Folders, Fields, and Views

The first step in developing any application is planning it. Without proper planning, you might dive too quickly into development, only to realize that you need more resources than you expectede or the application does not meet the requirements of your users. Planning an application begins with assessing why the application is needed. Figure out the business purpose of the application. This step sounds obvious, but it helps you focus your development efforts and define how complex or simple the application should be.

After deciding why to build the application, you need to answer the "who" question: Who are the users of this application? If the users are technically savvy, for example, you might want to include advanced functionality. If you are developing an expense report application that everyone in the organization will use, you will want to keep the design of the application simple to accommodate diverse users and technical skills.

In addition to considering the technical skills of your users, you have to think about the hardware on which the application will run. If laptop users need to use your application while traveling and will be disconnected from the network, you need to plan for offline support. If a remote user is the principal user of your application, you should make the application small and fast, since these users have low bandwidth connectivity.

To develop applications, you need software building blocks. In the same way that brick, wood, and concrete are the materials that carpenters need to build a house, software building blocks are the materials you need to build an application. Microsoft

Outlook provides five key building blocks for developing collaborative applications: folders, fields, views, forms, and actions. This chapter is dedicated to showing you how to take advantage of the first three. (In the next chapter, you will learn how to use forms and actions.) Specifically, this chapter will cover how to do the following:

- Create folders and set properties for a folder, including setting the permissions on a folder, setting the replication properties of a folder, and creating custom rules in a folder.
- Create custom fields, such as combination and formula fields, all of which allow your application to hold custom data. Plus, you will learn how to use these fields in setting your properties for filtered replication in Outlook.
- Create custom views by using the five default view types in Outlook, custom fields, and Microsoft Office document properties.

OUTLOOK DEVELOPMENT TIPS

Here are a few tips for developing applications with Outlook. As you read through this chapter and Chapters 4 and 5, keep these issues in mind:

If possible, develop and test your application in a personal folder before deploying it in a public folder.

If you have to develop your application in a public folder, restrict access. Personal folders do lack some public folder functionality such as permissions or rules, so if your application requires complex permissions or rules, you might want to build your application in a public folder. To limit access to this folder while constructing the application, set an option in the folder to restrict access to only owners of the folder. We'll talk more about this feature later in this chapter.

Always save a backup copy of a custom form before testing it. Certain logic errors on your form can freeze Outlook and force you to kill the Outlook process. For example, a simple oversight in your VBScript code could cause an infinite loop in your application. The only way to end the loop would be to kill the Outlook process. If you did not save a backup copy of the form, you would lose all the changes since the last backup.

As obvious as it sounds, you should test your application thoroughly before deploying it. This should involve trying all the permissions, views, rules, forms, actions, and custom code in the application. If you deploy an application and later realize you need to make changes, make a backup copy of the original application in your personal folders, modify the application backup in your personal folders, and retest and deploy the new application. This method provides the least disruption to current users of the application.

FOLDERS

Folders are the focal point for any Outlook application. They hold data; views for that data; and forms, agents, and rules. They provide users with a storage location for information and a hierarchy structure that makes finding information easy.

In Outlook, you can create folders in three places: in your mailbox stored on the Microsoft Exchange Server, in your personal folders stored on your computer's hard disk, and in public folders. Each of these locations has advantages and disadvantages. For example, if you create a new folder in your personal folders, you cannot easily share it with other users in your organization. In addition, you cannot set permissions on it. (In this book, we use public folders for storing application data.) Many of the properties you can set on public folders are applicable to the other two types of folders.

> **NOTE** Some of the steps and figures in this chapter are based on a user having permissions to create public folders. If you can right-click on a public folder and choose New Folder, you have permission to create a subfolder. If you are unable to create a public folder, contact your Exchange Server administrator to see whether a public folder is available to you that will allow you to create folders. If no public folder is available to you, ask your Exchange Server administrator for the proper permission.

Creating Public Folders

To help you work through the rest of the chapter, we are going to look at three simple applications that use the different building blocks of Microsoft Outlook: a threaded discussion application, an account tracking system, and a document library application. Each of these applications needs its own separate public folder to store its data.

To create a public folder for each application, follow these steps:

1. From the File menu in Outlook, select New and then Folder. The Create New Folder dialog box appears.
2. In the Name box, type a name for the folder. Start with the threaded discussion application, and type *Outlook Discussion Group*.
3. The drop-down list named Folder Contains shows possible items. Keep the default, which is Mail Items.

   > **NOTE** Outlook allows you to set the default type of item contained in the folder. If you were creating a public folder of task items, you would select Task Items from the drop-down menu. The folder can hold other types of items besides the default item you select.

4. In the Select Where To Place The Folder box, expand the Public Folders tree, select All Public Folders, and then click OK.

5. Outlook might prompt you about whether you want to add a shortcut to this folder to your Outlook bar. Click No.
6. Repeat these steps to create an Account Tracking public folder and a Document Library public folder.

Customizing Folder Properties

After creating the folders, you need to customize their properties for your application. Outlook automatically creates and sets certain properties of the folder for you. For example, Outlook creates common views for a folder based on the default type of folder you select. For a calendar folder, Outlook creates default calendar views such as day/week/month and active appointments; for a contacts folder, Outlook creates default contact views. You can change the default properties for a folder in the folder's Properties dialog box: right-click on a folder in the folder list, and select Properties from the context menu. The properties for the folder appear. Figure 3-1 shows the Properties dialog box for a Job Candidates application.

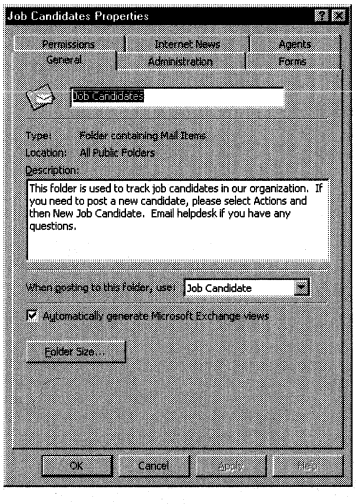


Figure 3-1. *The Properties dialog box for a Job Candidates application.*

General Tab

The General tab allows you to modify the general properties of a folder. In addition to specifying the folder name and describing the folder, you can do the following tasks:

- *Specify the default form for posting items to the folder.* You can set which default or custom form a user should use when submitting an item to the folder. As you will see with our sample applications, you'll want to modify this property after you develop custom forms for the folder.
- *Automatically generate Outlook views for users of the Exchange client.* When the Automatically Generate Microsoft Exchange Views check box is checked, Outlook automatically generates all views for the folder so that users on the Exchange client can use them. This property must be set if you want your custom Outlook views to be available in the Outlook Web Access client or to your Collaborative Data Objects (CDO) applications. By default, Outlook enables this property.
- *Check the size of the folder.* Click the Folder Size button to check how much space the folder is using to store its items and any subfolders. This option can help you figure out which folders are being used most frequently by users.

Administration Tab

The folder's Administration tab enables you to perform common administrative tasks. The following sections describe them.

- *Set the initial view for the folder.* The initial view can be either a built-in Outlook view or a custom view. Outlook Web Access respects this initial view property; when a user browses this folder in Outlook Web Access, the view you set will be the initial view.
- *Set how Outlook formats items dragged into your folder.* The Drag/Drop Posting drop-down list has two settings: Move/Copy and Forward. Move/Copy specifies that when an item is dragged into the folder, the item appears exactly as it appears in its original location. The user who drags the item into the folder is not indicated, and the person who originally posted the item is retained as the owner of the item. The Forward setting, in contrast, identifies the user who dragged the item into the folder as the user who forwarded the item. Outlook modifies the original text of the item to indicate that the item was forwarded.

- *Save the folder address to your personal address book.* Use the Add Folder Address To Personal Address Book button to save a folder's address so that you can later preaddress any custom forms that you want Outlook to automatically send to the folder. The administrator can also expose the folder in the Global Address List. Once this is done, the folder appears as just another recipient, which you can select in the address fields on your form.
- *Set the current availability of the folder.* By default, the option This Folder Is Available To is set to All Users With Access Permissions. While designing your application in a folder, you can set this property to Owners Only so that users cannot access the folder. This property affects only the current folder, so users still can access and continue working with subfolders under the parent folder. When a user tries to submit items to the parent folder while you have it disabled, Outlook returns the items with a note explaining that the folder and its contents are available only to owners at this time. After the application is done, you can reset this option so that all users can access the folders as long as they have proper permission.
- *Create rules for the folder.* The Folder Assistant button allows you to set rules for the folder. Because these rules can control information flow in a public folder and check specific properties of items as they are submitted into the folder, the Folder Assistant is important to designers of applications. For more information on designing rules, see the section titled "Creating Public Folder Rules" later in this chapter.
- *Moderate folder content.* The Moderated Folder button gives you access to settings that automatically moderate all the content in a folder before a user can post information. You can enable moderation on any public folder. For information on how to set up a moderated folder, see the section titled "Setting Up Moderated Folders" later in this chapter.
- *Show the folder path.* The Folder Path text box shows the location of the folder in the public folder hierarchy. Remember this property when you are designing an application, because it enables users to quickly open a folder without having to search through the public folder tree.

Forms Tab

On the Forms tab, you can specify which forms are associated with a folder. You can also restrict which forms users can post to the folder. Clicking the Manage button displays the Forms Manager dialog box, as shown in Figure 3-2. The Forms Manager allows you to copy custom forms from other folders or forms libraries into the current folder. You can also update or delete forms.

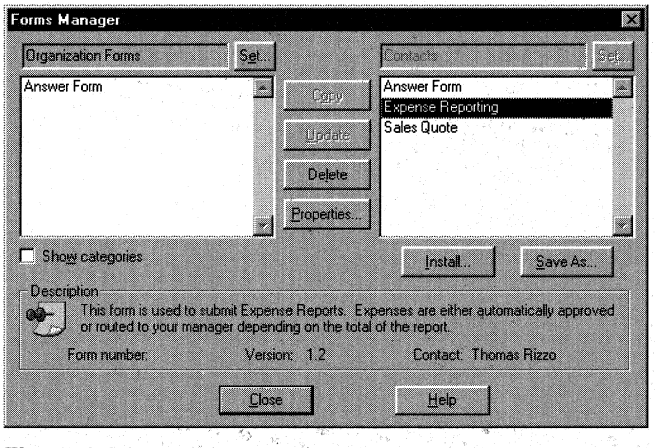


Figure 3-2. *The Forms Manager dialog box allows you to modify forms associated with the current folder.*

Permissions Tab

The Permissions tab, shown in Figure 3-3 on the next page, allows you to set user and group permissions for your folder and its items so that only those features you want your users to access are exposed. To modify these permissions, you need to be an owner of the folder. By default, when you create a folder, Outlook gives you owner rights. This means you have the full range of permissions to create, edit, or delete items in the folder. You can also change the permissions of other users in the folder.

When you first open the tab, you see that the default role for users is set to Author. This role corresponds to a set of permissions on the folder: users have the ability to view the folder, create and open items in it, and delete and edit their own items.

To learn how to set permissions for our Document Library and Account Tracking applications, follow the next set of steps. We'll limit who can create and edit documents in the folder to only users in our division, but we'll enable all users to at least read the information in our Document Library.

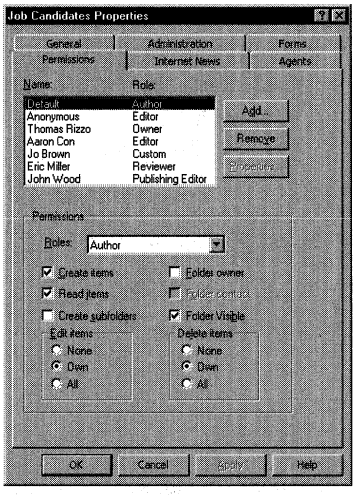


Figure 3-3. *On the Permissions tab of the Properties dialog box, you can add, delete, or modify the permissions that users have on the current folder.*

1. In the folder list, right-click on the Document Library folder you created earlier and select Properties.
2. Click on the Permissions tab.
3. In the Name box, select Default. In the Permissions area, select Reviewer from the Roles drop-down list.
4. Click Add, and select several coworkers from the address list in the Add Users dialog box. (Outlook also allows you to select and assign permissions to distribution lists. This capability makes it easier to set permissions for a large number of users.) When finished, click OK.
5. In the Name box, select one of the names you added in the preceding step. In the Roles drop-down list, select Publishing Author. This role will allow your coworkers to create, read, and edit their own items in the folder. Your Permissions tab should look similar to Figure 3-4.

Follow the same steps for the Account Tracking application, with these exceptions:

- Set the default permissions to None since we do not want anyone in our organization besides sales representatives accessing the application.
- Hide the folder from Default users by unchecking the Folder Visible check box. Remember to give your salespeople permission on the folder or they won't be able to see it either!

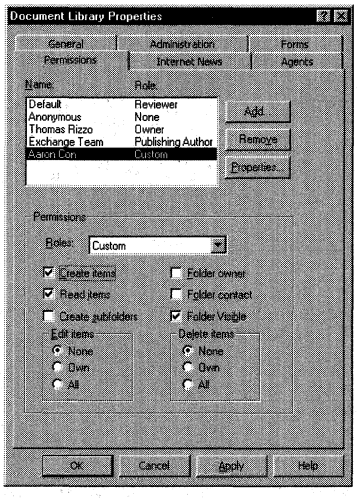


Figure 3-4. *Permissions for the Document Library application.*

Figure 3-5 on the next page shows an Outlook user browsing the public folder hierarchy. Notice that the Account Tracking folder is not visible to this user because he does not have the Folder Visible permission.

SELECTING INDIVIDUAL PERMISSIONS VS. SELECTING ROLES

Outlook provides roles with associated permissions so that you do not have to select each permission individually. If you wanted to create a custom role, you would select the permissions individually, and Outlook would apply these permissions to any type of item in the folder. For example, try dragging and dropping some Microsoft Word documents into the Document Library folder. Log into Outlook as a different user. This user is assigned the default permissions for the folder, meaning that all documents in the folder are read-only. Now double-click on one of the Word documents. You should see Word open but with Read-Only at the top of the document. This is Outlook maintaining the permissions you set on the items in the folder, even though the Word document is not a default Outlook item type.

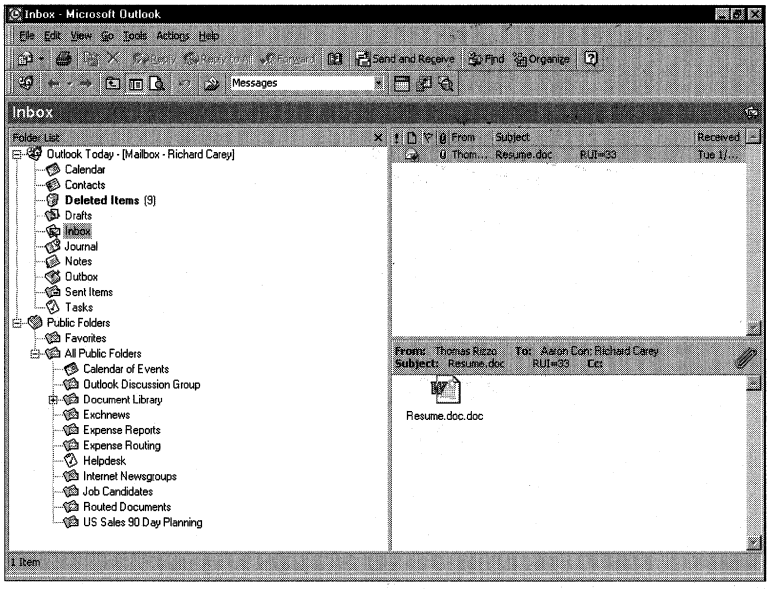


Figure 3-5. *A user browsing the public folder hierarchy. Since the user does not have permissions to view the Account Tracking folder, the folder does not appear in the hierarchy.*

Internet News Tab

On this tab, you can view the Internet newsgroup name of the public folder. Exchange Server supports exposing public folders as part of an Internet newsgroup hierarchy. For example, we can publish our Outlook Discussion Group as an Internet newsgroup named *Comp.MyCompany.Discussions*. By doing this, other corporations can receive, as a newsfeed, our threaded discussions in the public folder. On this tab, you can also set whether the public folder should be visible to newsreader clients.

Synchronization Tab

Outlook supports synchronizing folders and forms for offline use. Now let's set up two of our applications to handle offline synchronization:

1. Enable Outlook for offline access. From the Tools menu, select Options. Click on the Mail Services tab. Check the Enable Offline Access check box, and click OK.

2. To enable offline synchronization for public folders, add the folders to your public folder favorites. Open the folder list in Outlook. Expand the Public Folders tree to display Favorites and All Public Folders. Drag and drop the Document Library folder and the Account Tracking folder into the Favorites folder. Both folders should appear in your Favorites folder.

3. Open the Favorites folder. Right-click on the Document Library folder and select Properties.
4. Click on the Synchronization tab, and select the When Offline Or Online option.
5. Click OK.
6. Repeat Steps 3 through 5 for the Account Tracking folder. Now both of these folders are set for offline synchronization. Later in this chapter, we will use the Filter option in Outlook to select the items to synchronize, based on specific criteria, from the server to our client.

Setting Up Moderated Folders

One of the most requested features of an application that distributes information to many users is the ability to moderate content before it is posted. Moderation allows folder owners to decide which content is appropriate for the application and to select a group of people who can approve the content, and it discourages people from posting random information to the application. By using public folders, you can supply this functionality to your users without having to write any code yourself. The ability to moderate content is a built-in feature of public folders. To show you how moderated public folders work, let's enable moderation for the Outlook Discussion Group application. Take a look at Figure 3-6 on the next page as you follow these steps:

1. Find the Outlook Discussion Group folder you created in the public folder list, right-click on it, and select Properties.
2. In the Properties dialog box, click on the Administration tab.
3. Click the Moderated Folder button to open the Moderated Folder dialog box.
4. Check the Set Folder Up As A Moderated Folder check box to make the discussion folder a moderated folder.
5. In the Forward New Items To box, either type the names of people who are moderators or enter the address of another public folder to which Outlook should forward the items.
6. Check the Reply To New Items With check box. By enabling this option, every user who mails or posts items in the folder will receive a reply note from Outlook.

7. Choose Standard Response as the response type. Users automatically will receive an e-mail in their Inbox thanking them for their submission and explaining that there might be a delay before the item is available in the folder due to a pending review by other users.

   NOTE You can also send a custom response.

8. In the Moderators area, click the Add button. Select users or distribution lists to be moderators of the content placed in the folder. Figure 3-6 shows a sample Moderated Folder dialog box. When finished, click OK.

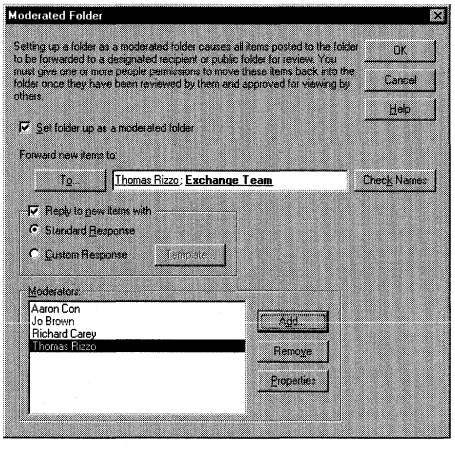


Figure 3-6. *The Moderated Folder dialog box.*

MORE ABOUT MODERATORS AND FORWARDING ITEMS

Moderators are individual users or distribution lists that are allowed to approve content. When a moderator posts an item to a folder, the item is not forwarded for review. Instead, the item is left in the folder. If the owner of the folder is not listed as a moderator, the item she posts to a folder will be forwarded for review. The owner cannot drag and drop the item back into the folder; Outlook automatically forwards the item for review again until a *moderator* drags and drops the item back into the folder. If you are going to use a moderated folder, add the folder owners as moderators.

Creating Public Folder Rules

Sometimes the built-in moderation features don't provide you with enough control over the information flowing into your application. So instead of using moderated public folders, you can place custom rules into your application. These rules automatically process new items as they arrive.

Rules consist of conditions and actions. As you would guess, if the conditions of a rule are met by an item, the associated action occurs. Outlook provides an easy way to create rules through the Folder Assistant. The Folder Assistant, shown in Figure 3-7, allows you to create, edit, delete, enable, disable, and order rules. We will step through an example later in this section.

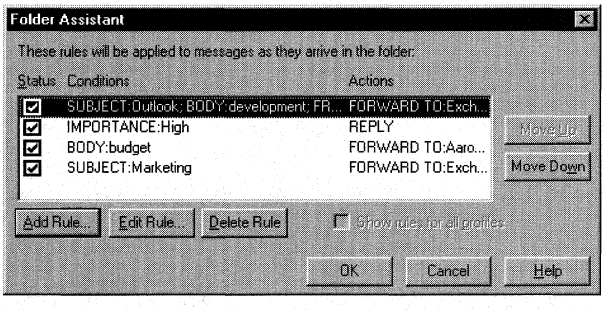


Figure 3-7. *The Outlook Folder Assistant helps you create custom rules for your applications.*

Setting the Conditions for a Rule

The conditions for a rule can range from very simple, such as checking who the item is from, to very complex, such as checking who the item is from and also searching the subject and text for specific phrases or text strings.

The Folder Assistant allows you to specify multiple conditions as well as multiple arguments within a single condition. Multiple arguments in a condition are separated with semicolons. When processing incoming items for a rule, Exchange Server ORs the arguments together. If the item meets one of the arguments, the associated action occurs. One example is to create a single rule that checks whether an incoming item is from any of the specified people. To do this, you use the From condition and separate each name with a semicolon, such as *FROM:Michael Rizzo; Jo Brown*. If the item is from either Michael Rizzo or Jo Brown, the action for the rule will occur.

If you specify multiple conditions on different items within a rule, Exchange Server will AND the conditions. All conditions must return true for the action to occur. For example, if you specify the From condition to be *FROM:Jo Brown* and the Subject condition to be *SUBJECT:New sales quote*, the item must both be from Jo Brown and have a subject of *New sales quote* for the action to occur.

You can combine the two techniques to make more complex conditions with multiple arguments. For example, suppose in a discussion database, you set the message Body condition to be *BODY:help;problem* and the From condition to be *FROM:CEO;CIO*. If a message is submitted to the folder from either the CEO or CIO and has either *help* or *problem* in the message body, your rule's action will occur. My recommendation for the action for this rule is to forward it to the help desk as a high-priority message!

In addition to allowing you to specify simple conditions such as the subject, name of the sender, and name of the intended recipient, the Folder Assistant allows you to set up what are called advanced conditions. Some examples of advanced conditions include size of the item, date ranges, and the presence of attachments. You can even specify advanced conditions that check user-defined fields on forms, folders, and custom office document properties.

One other advanced feature is the ability to create rules that fire when the conditions you specify are *not* met. For example, you might create a rule that fires for items that are from anyone except John Hand. To do this, you would specify *John Hand* in the From condition and then specify to process the rule only if the conditions are *not* met. This type of rule comes in handy when an inclusive condition, such as every user in an address book, is impractical to specify.

Finally, you can set an option in the Folder Assistant that will stop the rules engine from processing any subsequent rules after the current rule fires. You should use this condition when you have multiple rules in your folder and you want the current rule to be the last one applied.

Setting the Actions for a Rule

If the conditions of a rule are met, Exchange Server applies the rule's corresponding action to the item. There are four actions you can use in a rule, as shown in Figure 3-8.

Following is a description of these actions:

- *Return To Sender.* This action sends any item e-mailed into a folder back to the sender. Outlook does not allow the user to post the item. Instead, it returns notification that the user does not have permission to add this item into the folder.
- *Delete.* This action deletes the item immediately. By setting this action, you disable other possible actions in the rule, such as Return To Sender, and you automatically enable the Do Not Process Subsequent Rules condition.

- *Reply With.* This action automatically replies to the sender. You can customize the reply message by clicking the Template button, which opens a new message form. You can add recipients to the reply, enter custom message text, or insert any attachments that you want to include for the user. To save and close your reply template, choose Save & Close from the File menu.
- *Forward.* This action forwards all messages not marked as private to a specified recipient. You can specify the method that Outlook uses to forward the item. The options for this are Standard, Leave Message Intact, or Insert Message As An Attachment.

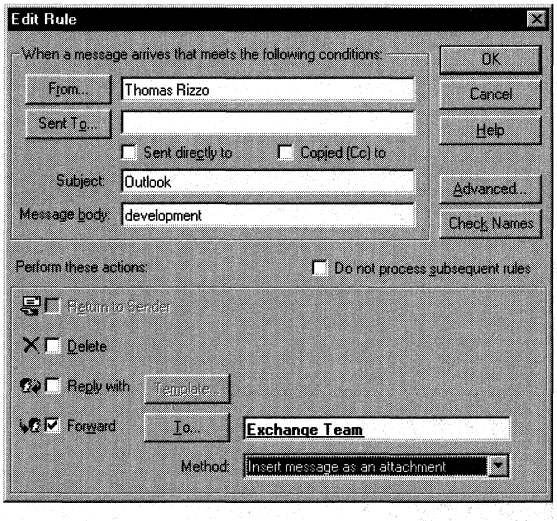


Figure 3-8. *The Edit Rule dialog box. Notice the four key actions that you can set for your rules.*

Applying Rules

Exchange Server will process multiple rules in the order that they appear in the Folder Assistant, which is from top to bottom. To change the order in which rules are applied, use the Move Up and Move Down buttons to move a rule higher or lower in the list, respectively.

Implementing Public Folder Rules

To help you understand how to implement public folder rules, we are going to customize the Account Tracking and Document Library applications with rules we create. For the Account Tracking application, we're going to add a custom reply for the user who submits an item. This reply will state that the folder has received the new item. Follow the first set of steps on the next page.

1. Find the Account Tracking folder in the folder list, right-click on it, and select Properties from the context menu.
2. On the Administration tab, click the Folder Assistant button.
3. Click the Add Rule button. The Edit Rule dialog box appears.
4. Check the Reply With check box.
5. Click the Template button to display the reply template.
6. In the reply template, type this in the Subject field: *Your item has been received.* In the message body, enter *Thank you for submitting your item to the Account Tracking application.* Your item should be available immediately for other people in the organization to use.
7. From the File menu, select Save & Close.
8. Click OK in the Edit Rule dialog box. Outlook prompts you that this rule will fire for all incoming messages. Click Yes.
9. Click OK in the Folder Assistant dialog box. (If a message box is displayed indicating that you do not have Send As permission, check with your Exchange Server administrator to ensure that you have Send As permission on the public Account Tracking folder. See Knowledge Base article Q152113 for more information.)

You should see your new rule in the Folder Assistant. Try posting a new message to the Account Tracking application to test your rule.

For the Document Library application, we're going to add an advanced custom rule that will check the Author property of the Microsoft Office document. If the author is not a member of our team, the item will be returned to the sender. To add this rule, follow these steps:

1. Follow steps 1 through 3 from the preceding procedure for the Document Library folder.
2. In the Edit Rule dialog box, click the Advanced button. In the Show Properties Of area, select the Document option.

   NOTE On some configurations, the properties do not display when you select the Document option.

3. Enable the Author property in the Show Properties Of section. For the values, type the names of people on your team; separate the names with semicolons.
4. Enable the Only Items That Do Not Match These Conditions check box. Click OK.

5. Enable the Return To Sender check box.
6. Click OK three times.

FIELDS

Fields are named variables where Outlook stores the data for your application. A number of built-in fields store default information. These built-in fields are associated with folders and their default content type. For example, in a Contacts folder, built-in fields include First Name, Last Name, Mailing Address, and Primary Phone. In your Inbox, built-in fields include From, To, Subject, and Message. Outlook also supports Office document properties as fields. For more information on using Office document properties as fields, see the section titled "Extending Functionality with Office Document Forms" in Chapter 4.

Outlook provides an extensive amount of built-in fields, but there will be many times when you need to add custom fields for your application. Outlook fully supports this capability and allows you to add custom fields to any folder. Your custom fields can range from a simple data type, such as a text field, to a complex data type, such as a formula field that includes a formula to calculate the value of the field from other Outlook fields.

Creating Custom Fields

The easiest way to create and delete custom fields in Outlook is to use the Field Chooser. The Field Chooser allows you to see both the built-in Outlook fields and your custom fields. The easiest way to access the Field Chooser is to select a table view in your folder, such as any of the default Outlook views that begin with the word *By*. For example, in your calendar, you can switch your view to the By Category view. After selecting a table view, right-click on the column headings and select Field Chooser from the context menu, as shown in Figure 3-9 on page 65.

To create the new field, click the New button, enter a name and data type for the field, and select the appropriate format. The following is a list of possible data types for fields in Outlook and the type of formatting these fields support:

- *Text.* This field type can hold text strings or a combination of text strings and numbers, such as a mailing address. It can be up to 255 characters long.
- *Number.* Use this type of field for numeric data (except numbers that represent currency) and for mathematical calculations. You can customize the format of this field with nine different formats, such as a scientific notation format (125.3E+03).

- *Percent.* Store numeric data that is a percentage here. You can choose from four different formats. For example, you can set how many decimal places to show in the percentage, such as show only one decimal place (10.9%).
- *Currency.* Store numeric data represented as currency here. You can format this data type to either show or hide the cents portion of the currency. For example, you can have this data type show either $5,232 or $5,232.10.
- *Yes/No.* This field stores data that holds only one of two values for the following pairs of values: Yes/No, True/False, On/Off, or a checked or unchecked check box.
- *Date/Time.* Store date and time data here. You can format this field with a number of standard formats, such as Monday, May 05, 1998 7:00 AM; 5/5/98 7:00 AM; May 5, 1998; or Mon 5/5/1998.
- *Duration.* This field is for numeric time data represented as an amount of time elapsed. You can expose the data in this field in several formats, such as 12h or 12 hours. This field automatically calculates when the data in the field should be displayed as days, hours, or minutes. For example, if you set the format for this field as "12 hours" and you enter *.25*, Outlook automatically displays 15 minutes. You can also set the format so that Outlook takes into account only working hours. (By default, that means an 8-hour day, but you can customize the default.)
- *Keywords.* This data type is used to hold multiple text values (which are separated by commas) and is similar to the Categories field used in Outlook. When creating custom views, keywords can be used to identify items. Examples of keyword field values are *small, medium,* and *large.*
- *Combination.* This data type holds a combination of fields and literal text. You can show each field or only the first nonempty field. The fields created with this data type are read-only in Outlook. For more information on creating combination fields, see the next section, titled "Creating Combination Fields."
- *Formula.* This data type holds the results of formulas you create. You can use the Microsoft Visual Basic expression service that is built into Outlook to create functions and operators for your formula. The fields created with this data type are read-only in Outlook. For more information on creating formula fields, see the section titled "Creating Formula Fields" later in this chapter.
- *Integer.* This data type holds nondecimal numeric information. You can customize the format to be only numbers, such as 3,332, or to be "computer" numbers formatted as kilobytes, megabytes, and gigabytes.

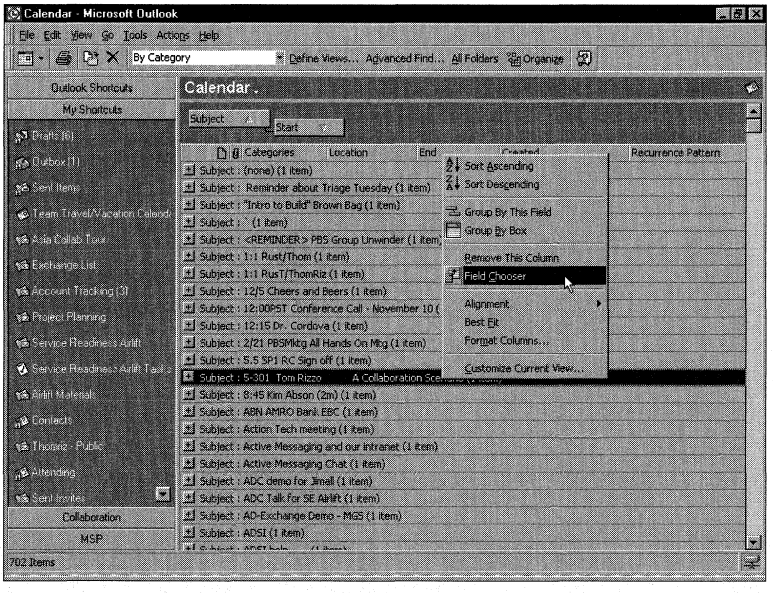


Figure 3-9. *Select the Field Chooser from the context menu for a table view.*

Creating Combination Fields

You can combine values from other fields with literal strings to create a new field, called a combination field. Combination fields are useful when you have many different types of fields and want to create a single field that combines them all. You can also use a combination field when you have multiple fields that hold conflicting data and you want to display only one of the fields in your form. Here's how you would create a combination field from two fields and a text fragment:

1. In your Inbox, open the Field Chooser and click New.
2. Enter a name for the field, such as *My Follow-Up Field*, and select Combination as the type of field.
3. Click Edit.
4. You have the choice to either join fields and text fragments together to create a combination field or show only the first nonempty field and ignore all the subsequent fields. Select the Joining Fields And Any Text Fragments To Each Other option.
5. Type *Need* in the Formula box.
6. Click the Field button, point to Frequently-Used Fields, and then click Follow Up Flag.

7. Type *By* in the Formula box.
8. Click the Field button, point to Frequently-Used Fields, and then click Due By.
9. Click OK twice.
10. Drag and drop this new field from the Field Chooser onto your Inbox column headings. You should see something similar to Figure 3-10.

Use a combination field when you want to expose a primary value for a specific field but also need the option to expose a field that holds a secondary value if the primary value is not available. If the primary value is available, you do not want to display the secondary value. For example, in our Document Library application, users can drag and drop Office documents into the public folder. Outlook can expose the properties of these office documents as fields. In the Outlook view, we want to expose the document author, but because users are not required to fill this property in when designing or saving the document, the Author property could be left blank. If the Author property is left blank, we want to use the From field in Outlook to display the name of the user who dragged and dropped the document into the folder. Follow the steps on the facing page to create the combination field that shows the first nonempty field.

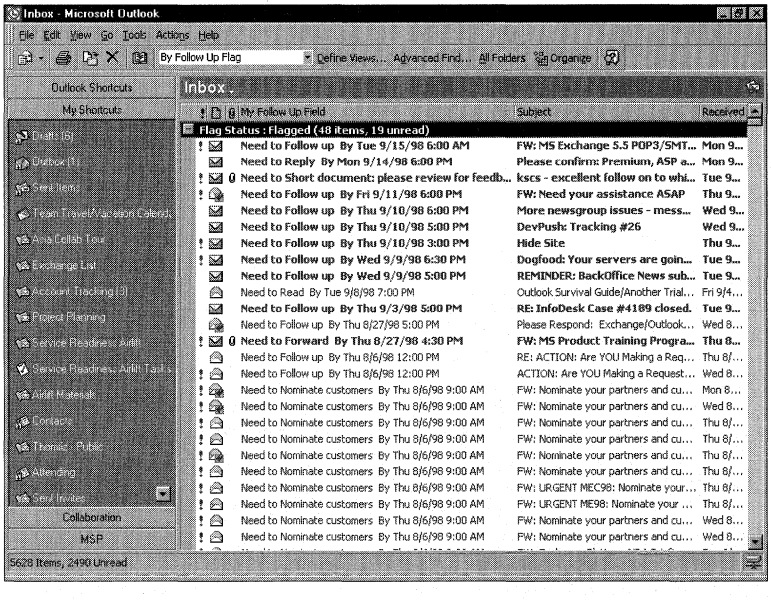


Figure 3-10. *Two different fields are combined to create a single combination field.*

1. Go to the Document Library folder you created, and open the Field Chooser.
2. Click New.
3. Type a name for the field, such as *Document Author*.
4. Select Combination as the field type.
5. Click Edit.
6. Select the option named Showing Only The First Non-Empty Field, Ignoring Subsequent Ones.
7. Click the Field button, point to All Document Fields, and click Author.
8. Click the Field button, point to All Mail Fields, and click From.
9. Click OK twice.
10. Drag and drop your new field from the Field Chooser onto the view column headings. You should see a view similar to the one shown in Figure 3-11.

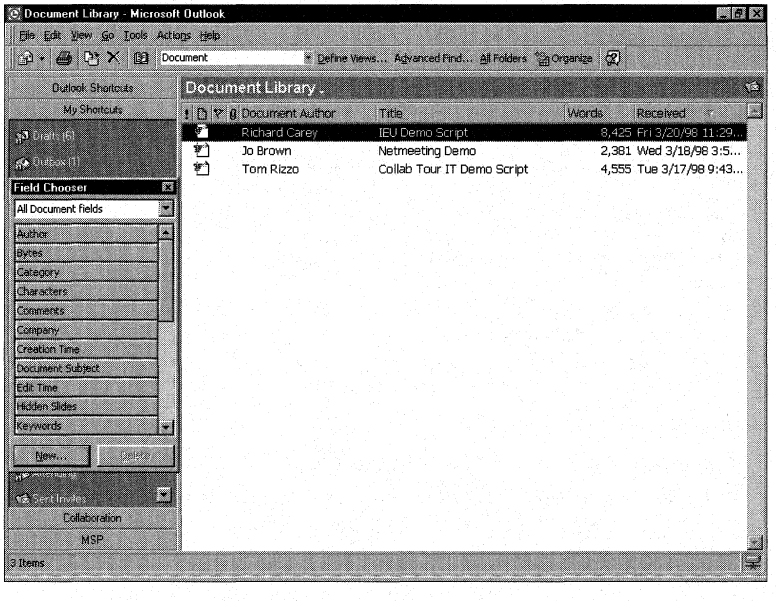


Figure 3-11. *You can use combination fields to show the first nonempty value from multiple Outlook fields.*

Creating Formula Fields

Formula fields allow you to use functions to calculate values from both standard and custom fields. These calculated values are stored inside of the formula field. Use formula fields when you need to calculate the value of one field based on other fields in your application. For example, you can use a formula field to calculate the total of an expense report or a person's wage based on the amount of time they worked multiplied by their hourly rate. Outlook makes creating formula fields quite easy by offering a simple interface for field selection and by displaying a list of all possible formulas and required inputs. You can use the provided Field and Function buttons to quickly select the fields and functions you want to use in your formula. Outlook will automatically parse your formula and check it for syntactical errors.

Formula fields do have some performance implications. First, Outlook has to process formula fields whenever values change in the application. The more complex you make your formulas, the longer Outlook will take to process them. Second, Outlook automatically recalculates formula fields whenever the current view changes. Third, Outlook does not allow you to sort, group, or filter views by using formula fields.

Follow these steps to create a formula field that displays the amount of time elapsed since an item was received. Figure 3-12 shows a custom formula field.

1. In the Field Chooser, click New.
2. Type a name for your new field, such as *Days since received.*
3. Select Formula from the Type drop-down list.
4. Click the Edit button.
5. In the Formula box, type this:

```
IIF(DateDiff("d",[Received],Now())>=7,
DateDiff("w",[Received],Now()) ? " week(s) ago",
DateDiff("d",[Received],Now()) ? " day(s) ago")
```

6. Click OK twice.
7. Drag and drop the new field onto your view column headings. You should see the field automatically calculate. If the item was received within one week, the field displays the amount in days. If the item was received more than a week ago, the field displays the amount in weeks.

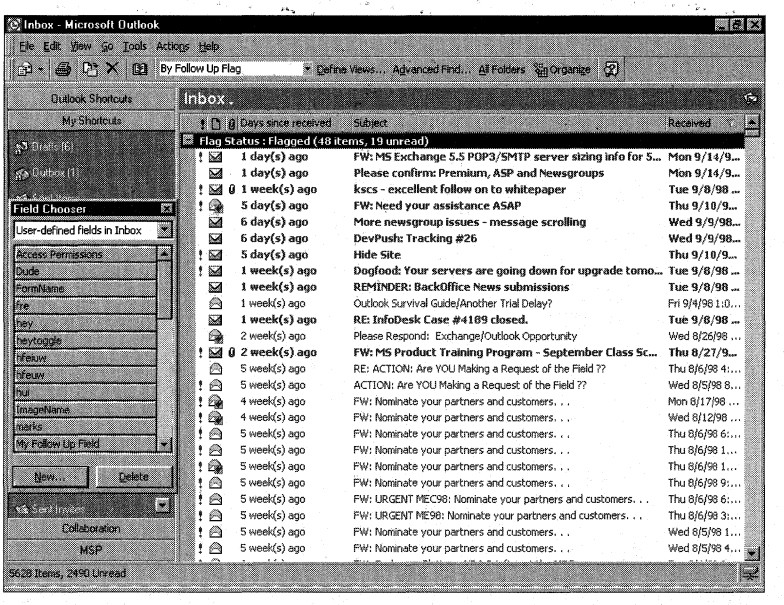


Figure 3-12. *A custom formula field in an Outlook view.*

Using Custom Fields in Filtered Replication

As you learned in Part I, Outlook and Exchange Server support synchronizing items offline. By default, Outlook synchronizes all the items from the server to your offline database. But what if you don't want to synchronize *all* the server items offline? Outlook offers the capability to synchronize subsets of information, called filtered replication. You select the parameters—either built-in or custom fields—that Outlook will use to filter the synchronized items. To set up filtered replication, follow these steps:

1. Before setting filtered replication, you should add a custom property to the Account Tracking application by selecting the previously created Account Tracking folder, which is nested in the Favorites folder.
2. In the Field Chooser, click New.
3. Type *txtAccountSalesRep* in the Name box, and select Text as the type.
4. Click OK.

5. From the File menu, point to the Folder option, and then select Properties For Account Tracking.
6. Click on the Synchronization tab. In the This Folder Is Available area, make sure that the When Offline Or Online option is selected.
7. Click the Filter button.
8. In the Filter dialog box, click on the Advanced tab.
9. Click the Field button, point to User-Defined Fields In Folder, and click txtAccountSalesRep.
10. From the Condition drop-down list, select Is (Exactly). In the Value text box, type the name of a user of the Account Tracking application.
11. Click the Add To List button to add your criteria. Your screen should look like Figure 3-13.
12. Click OK, click Yes, and then click OK twice. Now only the items meeting your criteria will be synchronized to your offline database.

Figure 3-14 shows the folder before setting filtered replication and after setting filtered replication. As you can see, a subset of the information in the folder is available to the client offline.

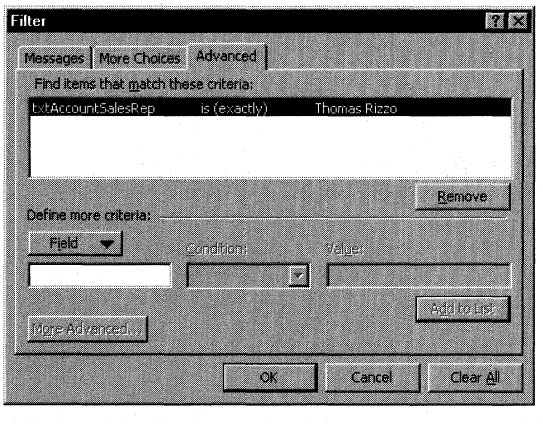


Figure 3-13. *The Filter dialog box, showing synchronization information for the Account Tracking application.*

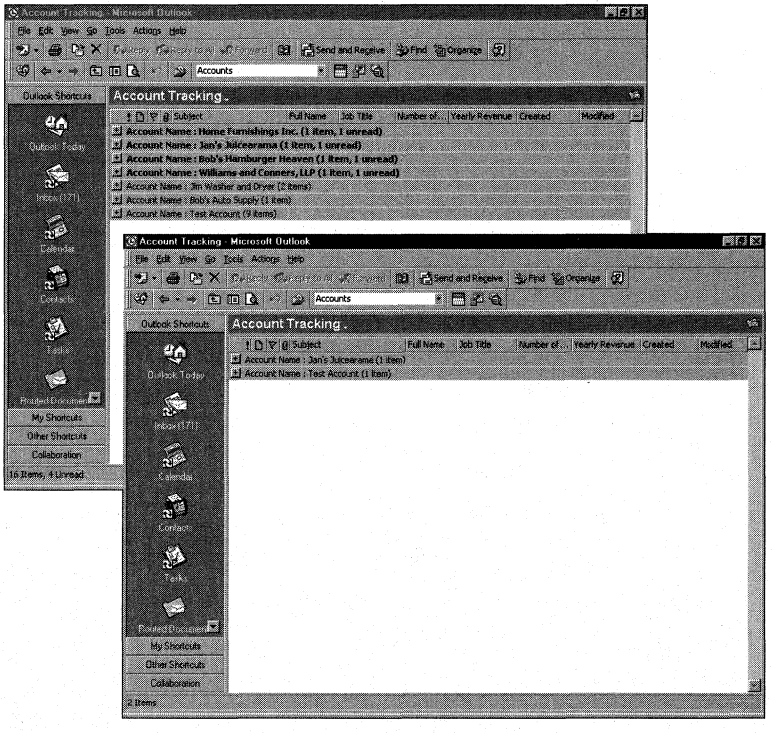


Figure 3-14. *The Account Tracking folder before filtered replication, and the Account Tracking folder after filtered replication. Notice how a subset of items are synchronized offline.*

VIEWS

Outlook supports a variety of folder views, including custom views, to give you and users flexibility in the presentation and organization of information. These views can be used in any type of Outlook folder. Outlook allows you to set the initial view for a folder, and it remembers the state of the view for each user. Outlook supports five types of views:

- *Table view.* The most commonly used Outlook view, the table view consists of rows and columns that expose the information from the folder.
- *Timeline view.* This view shows, as icons, the chronological order of the items in the folder.
- *Card view.* The card view shows the items in the folder as individual cards, similar to a business card file.

- *Day/Week/Month view.* In this view, items are arranged in a calendar. This view is best used for applications that have a date and time field as one of the application's primary fields.
- *Icon view.* The icon view presents all items in the folder as individual icons on an invisible grid. The icon view is best used for items where seeing the details of the item are not important.

Creating New Views

Outlook provides two options for creating new views: defining new views with the Define Views dialog box or adding the Current View box to your Outlook toolbar. This second option is the easiest and is the one you will probably use more often. To add the Current View box to your Outlook toolbar, follow these steps:

1. Right-click on the Outlook toolbar, and select Customize.
2. On the Commands tab, select Advanced from the Categories list.
3. On the Commands list, scroll down until you find the Current View drop-down list.
4. Drag and drop the Current View drop-down list onto your toolbar. You should see Outlook fill in the Current View drop-down list with the name of the current view of the folder. You can now use the Current View drop-down list to create new views by typing over the name in the box.

When you attempt to save your views, Outlook will prompt you to indicate which view will be used. There are three primary ways you can apply your views in Outlook:

- *This Folder, Visible To Everyone.* This option enables the view to be used on the current folder and to be visible to everyone. Any person with permissions to open the folder will be able to select this view from the drop-down list of current views. As a developer, you can create the views for your Outlook application and then save them so that all your users can use them.
- *This Folder, Visible Only To Me.* When you select the private view option, the view is for the current folder but is only visible to the current user. You could use this view if you wanted to show specialized information (like debugging information) to only certain users, not all users of your application.
- *All Folders.* This option enables the view to be used in all folders that have the same item type as the current folder. This allows you to share your favorite views across folders of the same type.

So far, our Document Library application is only a folder where users can drag and drop documents to share with other users. By using views, we can transform our simple Document Library application into a more powerful and useful application for our users. The first view we are going to create is an icon view so that our application looks more like a network file share than an Outlook folder. This will make it easier for our users to navigate among the files in the folder. To create the icon view for the Document Library application, follow these steps:

1. Select the Document Library folder in Outlook.
2. From the View menu, point to Current View, and then click Define Views.
3. Click the New button, and type a name for your view, such as *As Icon*.
4. Select Icon from the Type Of View box.
5. Select This Folder, Visible To Everyone in the Can Be Used On area.
6. Click OK. The View Settings dialog box is displayed.
7. Click the Other Settings button, and select the type of icon you want to use: Large Icon, Small Icon, or Icon List.
8. Click OK twice.
9. Click Apply View. We now have an icon view of our Document Library application. Your view should be similar to the view shown in Figure 3-15.

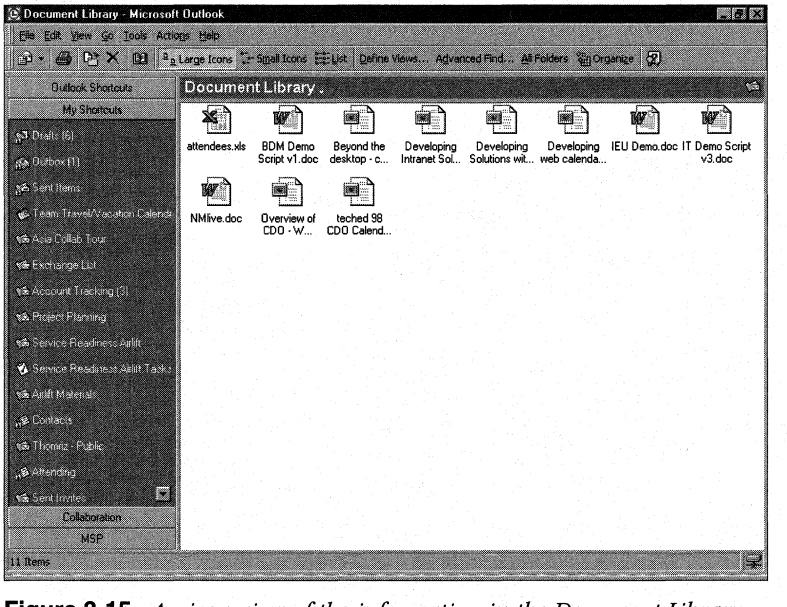


Figure 3-15. *An icon view of the information in the Document Library application.*

The second view we are going to create is a timeline view. This view will enable our users to quickly see the last time a document was saved and how much time has elapsed between the time the document was created and the time it was last edited and saved. By implementing this feature, users can quickly discard older versions of the document to ensure they are using the most recent version. As you will see in Chapter 4, you can customize your views by using custom properties directly from Office documents.

To create the timeline view for the Document Library application, follow these steps:

1. Select the Document Library folder in Outlook.
2. From the View menu, point to Current View, and then click Define Views.
3. Click the New button, and type a name for your view, such as *Document Timeline*.
4. Select Timeline from the Type Of View box.
5. Select This Folder, Visible To Everyone in the Can Be Used On area.
6. Click OK. The View Settings dialog box is displayed.
7. Click the Fields button. The Date/Time Fields dialog box is displayed.
8. Select the Created field in the Available Date/Time Fields list as the starting time for the document in the view, and then click the Start button.
9. Select All Document Fields from the Select Available Fields From drop-down list.
10. Select Last Saved Time from the Available Date/Time Fields list, and then click the End button so that the last-saved time is the ending time for the document in the view.
11. Click OK twice. Click Apply View. The view of your documents folder should be similar to Figure 3-16.

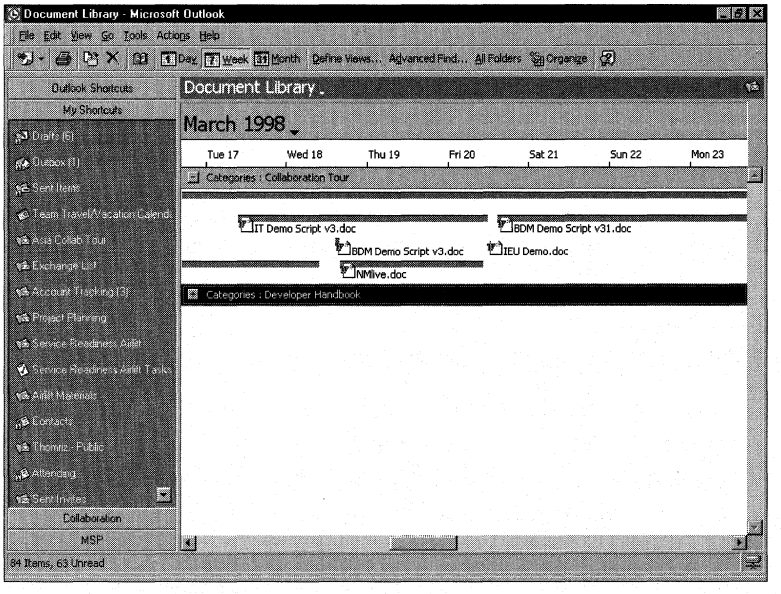


Figure 3-16. *A timeline view of the Document Library application. Notice how Outlook automatically draws a line indicating the amount of time that has elapsed between the creation time and the last-saved time of the document.*

Customizing the Current View

You can customize the current view by using the Field Chooser to drag and drop new columns. After you add the new column, the view is automatically updated using values taken from the items in the folder. You can also add complex data types to the view, such as combination or formula fields. Many people find it easier to customize a view by using the drag-and-drop capabilities of the Field Chooser as opposed to selecting available columns from a drop-down list.

To add new columns to the Document Library view, follow these steps:

1. In the Document Library application, create a new table view and name it *Document Properties*.
2. Right-click on any column heading in the Document Properties view, and select Field Chooser.
3. Select All Document Fields from the drop-down list.

4. Drag and drop Author, Revision Number, and Last Saved Time from the Field Chooser to an area next to one of the columns in the view. Outlook presents red arrows to indicate where the field will be inserted.
5. To remove a column, select the column heading and then drag and drop it off the column heading row.

   NOTE You can remove columns easily from your view by dragging the columns until a large X appears. Once you release the mouse button, the column disappears from the view.

Formatting the Columns in a View

Notice that when you drag and drop columns from the Field Chooser, Outlook, by default, gives the column heading the same name as the field on which the column is based. Also notice that Outlook applies default formatting for the columns. For example, Outlook automatically formats the Last Saved Time column with the day, the date, the year, and the time the document was last saved. Most users won't need this much detailed information about the last-saved time for the document. To make views more intuitive to your users, Outlook allows you to change the name of the column heading without changing the name of the underlying field. You can also change the default format of values for a specific column in the view. For example, you can change the format of the Last Saved Time column heading so that it only displays the date the document was last saved rather than the date and the time, as we saw earlier. Please note, however, that changing the format of the column does not modify the format of the field on which the column is based. To modify the format of the field, you must use the Field Chooser. To change the format of a column in the Document Library application, follow these steps:

1. Right-click on the Last Saved Time column heading in the Document Properties view, and select Format Columns from the context menu.
2. In the Format drop-down list, select the option that shows only the month, the day, and the year, such as April 07, 2000.
3. In the Label box, enter *Last Edit Time*.
4. Click OK. Your date/time column should look similar to the one shown in Figure 3-17.

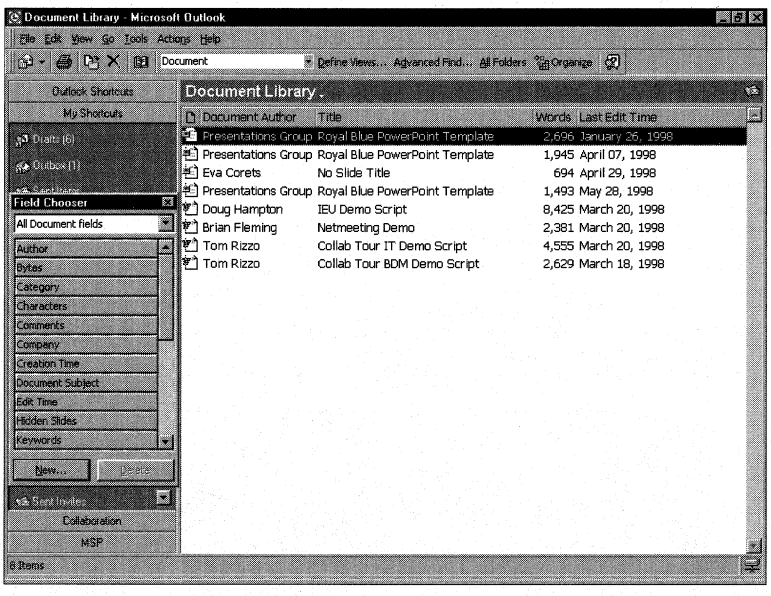


Figure 3-17. *The new Document Properties view after changing the format and label of the date/time column.*

Grouping Items in a View

Grouping items in an Outlook view makes it easy for users to find items that are related. Outlook supports up to four levels of grouping in a single view. You can group items in a view in one of two ways:

- *Using the Group By box.* This is the easiest method because it allows you to use drag-and-drop functionality to select the column as the grouping. If you drag and drop more than one field into the Group By box, Outlook graphically draws the relationship between the fields as primary groups and subgroups.
- *Customizing the Current View option.* This method gives you a few more options when setting the grouping for a view, but it isn't as easy as using the Group By box.

To group items by Author using the Group By box, follow these steps:

1. Right-click on a column for the Document Library application, and select Group By Box to display the Group By box above the column headings.
2. Drag and drop the Author column into the Group By box, or drag items from the Field Chooser into the Group By box.
3. If you want to group by more than one field, drag and drop a second field into the Group By box. For our purposes, drag the Categories field from the Field Chooser to the Group By box. Notice how Outlook draws a line from the Author field to the Categories field to indicate that the view is grouped first by author and then by category. Take a look at Figure 3-18.

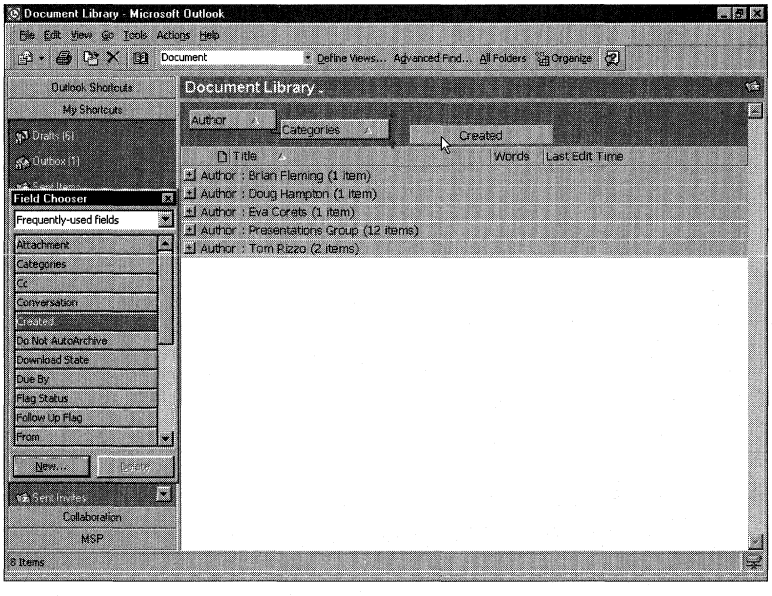


Figure 3-18. *The line connecting the Author field to the Categories field indicates how the items are grouped—in this case, by Author first, and then by Categories.*

4. Close the Group By box by right-clicking on a column heading and then selecting Group By box.

To create the same grouping using the Current View option, follow these steps:

1. From the View menu, point to Current View, and then click Customize Current View.
2. Click the Group By button to display the Group By dialog box.
3. In the Select Available Fields From drop-down list, select All Documents Fields.

4. In the Group Items By area, select the Author field. You can also show the field in the view by checking the Show Field In View check box.
5. In the Select Available Fields From drop-down list, select Frequently-Used Fields. In the first Then By area, select the Categories field. Figure 3-19 shows the completed Group By dialog box.
6. Click OK twice.

You now have created the same view using both methods. The only difference between the two is that in the Group By dialog box, you can select whether the groups are expanded or collapsed by default.

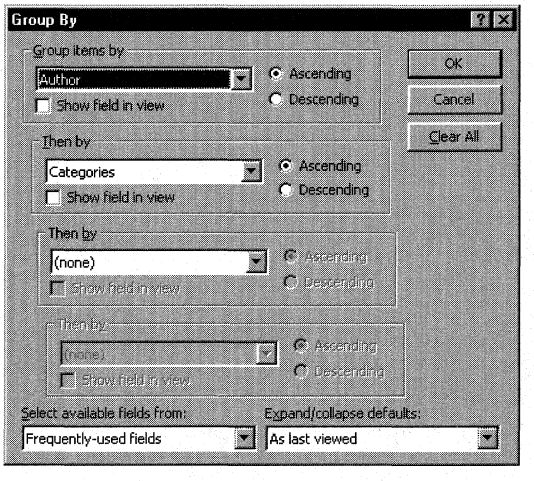


Figure 3-19. *The Group By dialog box.*

Sorting Items in a View

Outlook also supports the ability to sort items in a view in either ascending or descending order. When you combine sorting with grouping, you get the best combination of features for making your information available to users in a view. For example, instead of just grouping our Document Library items by author, we can also sort the items so that the most recently saved documents appear at the top of the grouping.

To create a sorted list, you can click on the column heading or use the Sort dialog box. To create a sorted view by using the Sort dialog box, follow these steps:

1. From the View menu, point to Current View, and then click Customize Current View.
2. Click the Sort button to display the Sort dialog box, shown in Figure 3-20. In the Select Available Fields From drop-down list, select the category, or location, of the field you want to use as your sort criterion.

3. In the Sort Items By area, select the field that you want to use as your sort criterion.
4. To select further sorting subsets, select the next field you want to sort by in the Then By area.
5. Click OK twice.

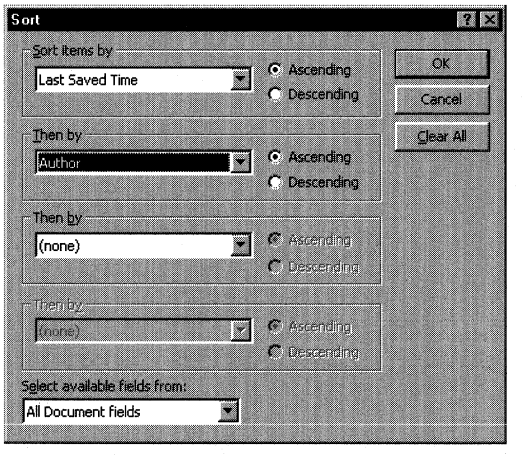


Figure 3-20. *The Sort dialog box in Microsoft Outlook.*

Filtering Information in Views

Filtering allows you to create views in which only certain information is visible to users. The criteria you set can be built-in Outlook fields or custom fields. Filters can have only one or two conditions or they can be more complex, using multiple conditions or the advanced filtering features. When you set multiple conditions on a filter, Outlook ANDs them together. When you set multiple arguments in a single condition, Outlook ORs these arguments so that if only some meet the condition, the item appears in the view. To create a simple filter for the Document Library application, follow these steps:

1. From the View menu, point to Current View, and then click Customize Current View.
2. Click the Filter button.
3. Click on the More Choices tab.
4. In the Categories box, type *Outlook; Exchange*, as shown in Figure 3-21.
5. Click OK twice.

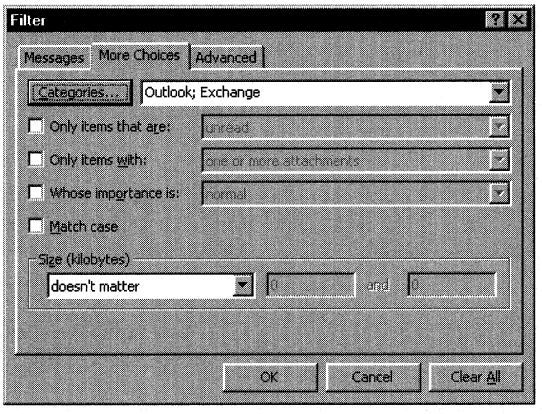


Figure 3-21. *The Categories filter for the Document Library application.*

Here's how to create a complex filter for the Document Library application:

1. From the View menu, point to Current View, and then click Customize Current View.
2. Click the Filter button.
3. Click on the Advanced tab.
4. Click Field, point to All Document Fields, and then click the Last Saved Time field.
5. In the Condition drop-down list, select Last Month.
6. Click Add To List.
7. Click Field again, point to All Document Fields, and then click the Author field.
8. In the Condition drop-down list, select Is Not Empty. Click Add To List.
9. Click Field again, point to All Mail Fields, and then click the Message Class field.
10. In the Condition drop-down list, select Contains and type this text in the Value box: *IPM.Document.Word.Document.* Click Add To List.
11. Click OK twice.

Editing View Settings

You can customize the formatting of your views at a more detailed level by using the Other Settings dialog box. The type of view you create determines which settings are available for you to edit. For example, if you are editing a table view, you can set the font size, enable in-cell editing, enable autopreview, create gridlines, and enable

the preview pane. If you are editing an icon view, you can set the view type, such as Large Icons, Small Icons, or Icon List, and you can specify whether Outlook should automatically arrange and sort the icons in your view.

IN-CELL EDITING

The in-cell editing option for customizing a table view allows users to quickly add new items to a folder or change the properties of current items in the folder without opening a form. All of the changes to the item can be typed directly into the view. This capability helps speed up applications that require a lot of data entry, such as customer contact lists or surveys.

If you enable in-cell editing in a folder that contains Office documents, you cannot modify the properties of the Office documents directly in the view. These properties are read-only inside of Outlook. You must modify these properties using the Office program that originally created the document.

Conditional Formatting

If your custom view is a table view, you can use the conditional formatting capabilities of Outlook. Conditional formatting enables items that meet certain conditions to use your custom formatting. For example, you can set a condition in the Document Library application so that all Word documents appear in a 12-point, red Arial font. Outlook automatically sets some default formats for the most common conditions, such as unread, expired, and overdue e-mail. You can customize the settings for these default conditions or create your own conditions. To set conditional formatting, follow these steps:

1. From the View menu, point to Current View, and then click Customize Current View.
2. Click the Automatic Formatting button.
3. Click Add, and type a name for the formatting rule in the Name box, such as *Word Documents*.
4. Click the Font button, and select 12-point Arial as the font and red as the font color. Click OK.
5. Click the Condition button. As you can see, the dialog box is the same as the Filter dialog box we saw in Figure 3-21.

6. Click on the Advanced tab.
7. Click Field, point to All Mail Fields, and then click Message Class.
8. From the Condition drop-down list, select Contains. In the Value box, type *IPM.Document.Word.Document*.
9. Click Add To List and then OK.
10. Click OK two more times. Your screen should look like the one shown in Figure 3-22.

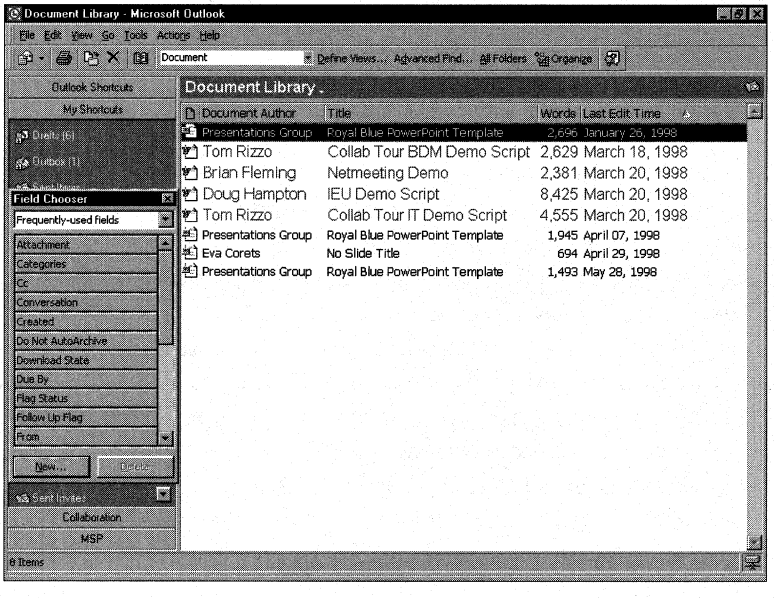


Figure 3-22. *The Document Library application after applying conditional formatting for Word documents.*

Limiting Views to Only Those Created for the Folder

Outlook, by default, provides several standard views in a folder based on the folder's default content type. In many cases, these default views are not relevant to your application, so you will not want them to appear in the Outlook view list. Outlook allows you to hide the default views and only show custom views created for the current folder by checking the Only Show Views Created For This Folder check box in the Define Views dialog box.

DISABLING DEFAULT VIEWS IN MICROSOFT OUTLOOK WEB ACCESS

If you are planning to use Outlook Web Access as one of the clients for your application, the Only Show Views Created For This Folder property will not disable the default views from appearing in Outlook Web Access. To do this, you have to customize the Active Server Pages of Outlook Web Access to hide the default views for your folder. Any custom table views that you create in the Outlook client will automatically be available to the Outlook Web Access client as long as the Automatically Generate Microsoft Exchange Views check box is checked.

Implementing Threaded Views

Many times, you'll want to display the information in a folder as a threaded view so that users can see the history of responses to an item. These responses are indented in the view to make it easier to follow the flow of information about the item. In a folder based on e-mail items, Outlook provides a default view called By Conversation Topic, which provides this threading capability. But suppose you don't build your application based on e-mail items but instead build it based on tasks. To create threaded views in these types of folders, Outlook supports two unique properties called Conversation and Conversation Index.

The conversation field is based on the message's subject field. This means that when you create a new item in a folder, the conversation field is automatically filled with the content of the item's subject field, so any replies inherit the conversation field from the original item.

The conversation index is a unique identifier used by Outlook to track the series of responses to an item. This index allows Outlook to know which item in the thread the user is responding to and where the response should be placed in the threaded view.

To implement threaded views for any of your Outlook folders, you must group by the conversation field and sort in ascending order by the conversation index. The sort by conversation index is usually the step that most developers forget about when trying to implement threading. Without it, your view will be grouped only by the conversation index and will be void of any indented text indicating responses to items.

To implement threaded views in Outlook folders, follow this procedure:

1. From the View menu, point to Current View, and then click Customize Current View.
2. Click the Group By button.

3. In the Group Items By area, select Conversation and click OK.
4. Click the Sort button.
5. In the Sort Items By area, select Conversation Index. To see this property, you might have to select All Mail Fields from the Select Available Fields From drop-down list.
6. Click OK twice.

Figure 3-23 shows a threaded view.

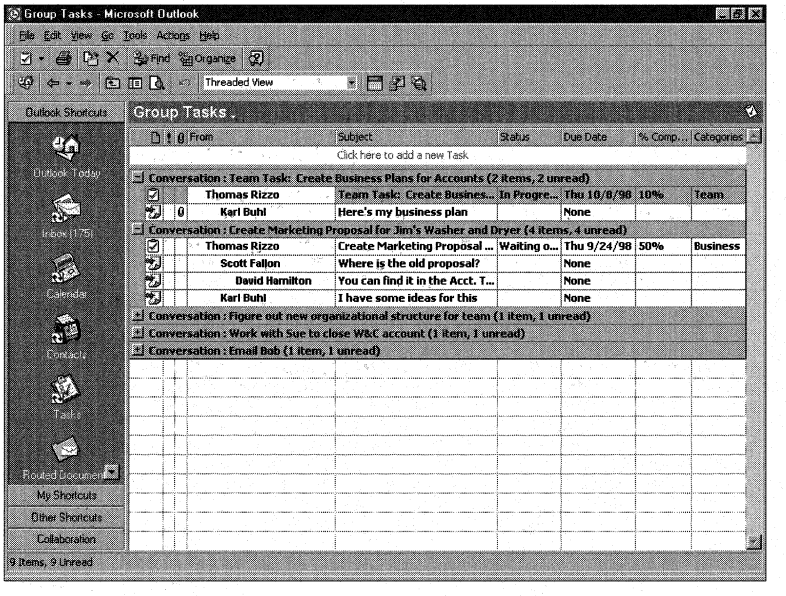


Figure 3-23. *A threaded view of a Group Tasks public folder.*

NOTE In all folders except those based on e-mail items, Outlook does not automatically make the Post Reply To This Folder menu option available. This menu option allows you to post replies in a folder that automatically inherit the conversation property from the original item. To enable this menu option, you can add this command to the Outlook menu or toolbar.

Chapter 4

Forms

You are already familiar with the capabilities of Microsoft Outlook forms since every item you view or use in Outlook is based on one. Customizing these forms can enhance the way you distribute and collect information electronically both inside and outside your organization.

Outlook allows you to build custom forms based on default Outlook items. When you customize built-in forms, your application inherits default capabilities. You can extend the functionality of these forms using custom controls and Microsoft Visual Basic Scripting Edition (VBScript). We'll look at VBScript in more detail in Chapter 5.

Outlook also allows you to base your forms on Microsoft Office documents, which prevents you from having to re-create existing functionality. For example, if you were building an expense reporting application, you could base your Outlook expense reporting form on a Microsoft Excel document, giving you the full power of Excel inside your application. You could then further extend your application using Microsoft Visual Basic for Applications (VBA) inside the Excel document. You could also use custom Excel properties inside your Outlook views to sort and group items.

Outlook Form Types

To help you understand the types of applications you can develop and when to customize certain forms in the Outlook environment, you need to know what the form types are and how they can be extended.

Message Forms

The Message form should be used for applications in which users have to send information to other users or to a folder. You inherit all built-in capabilities of the form,

such as automatic name resolution and nickname support, and all fields on the form can be customized. Figure 4-1 shows an example of a Message form in design mode. Different pages of the form are displayed by clicking the appropriate tab.

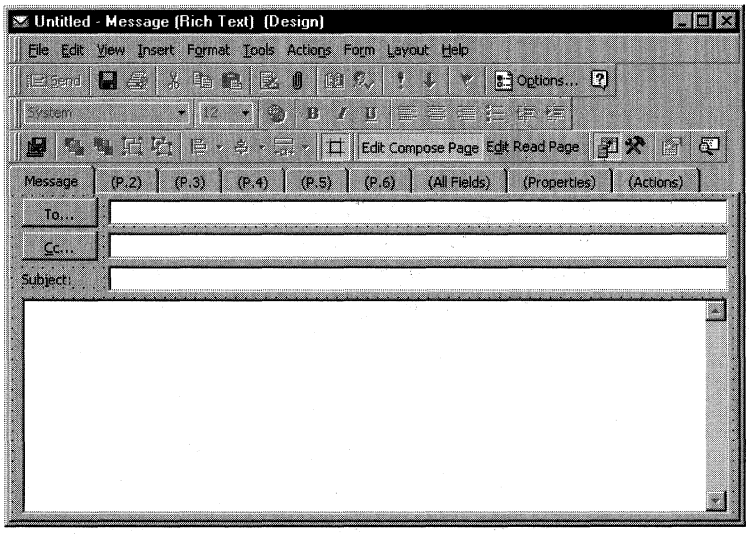


Figure 4-1. *The Message form in design mode.*

Post Forms

The Post form is best used in applications that post or retrieve messages in an Outlook folder. When you customize a Post form, Outlook automatically assigns the currently open Outlook folder to the In Folder field. For example, if you customize a Post form in a helpdesk public folder, Outlook automatically assigns the helpdesk public folder to the In Folder field. This automatic assignment means that even though the user can install, or publish, the Post form in any folder, any items the user creates with the form will be posted to the helpdesk public folder. Figure 4-2 shows the Post form.

Contact Forms

Use the Contact form in applications that track address or customer information. You can customize the first page of the Contact form; by doing so, you inherit the form's journaling, mapping, Microsoft NetMeeting, and address resolution capabilities. The other default pages in the form are not customizable but can be hidden. Figure 4-3 shows the Contact form.

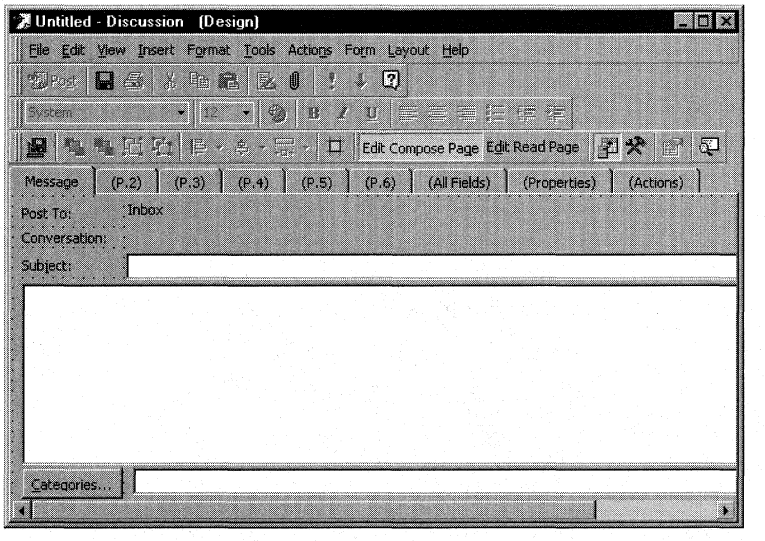


Figure 4-2. *The Post form in design mode.*

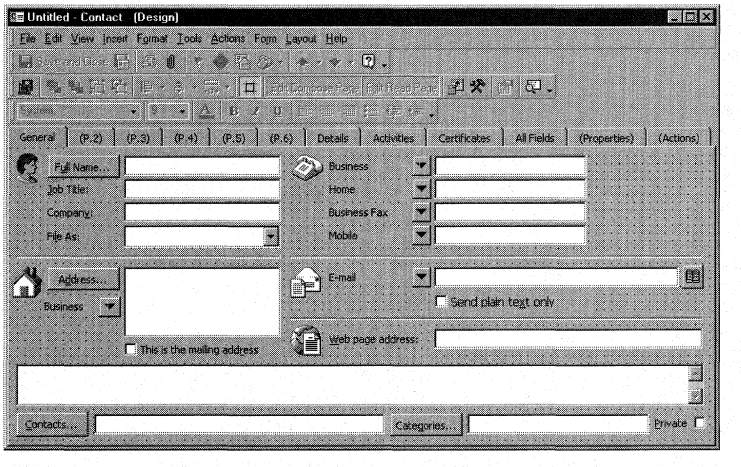


Figure 4-3. *The Contact form in design mode.*

NOTE You can customize a number of other Outlook forms, including the Appointment, Task, and Journal forms, by hiding them or adding new pages to them. However, you cannot customize any of the built-in pages of these forms.

Any custom applications you develop using the Journal form will post information to a user's personal journal.

Office Document Forms

Outlook supports embedding Microsoft Word, Excel, and Microsoft PowerPoint documents directly into a form, so you can send these documents to a user or post them to a folder. These types of forms are best used when you want the advanced replication and forms library support of Outlook, but you also want the functionality of other Office applications. Outlook places a wrapper around the Office application you use to design the form, so you cannot add custom tabs to the form. You can, however, customize the application by using the built-in capabilities and tools provided by the specific Office application. For example, you can use VBA to customize an Outlook Office document form. Figure 4-4 shows an Excel document form.

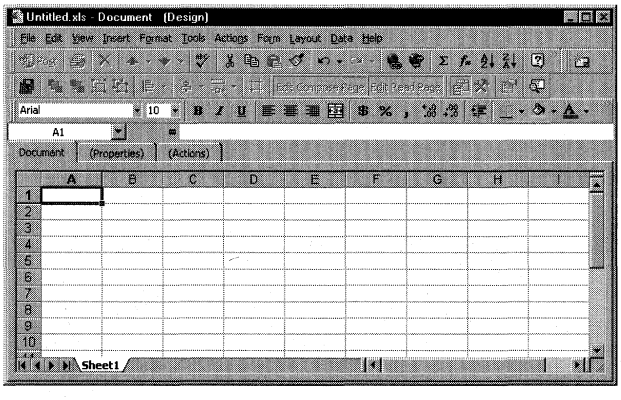


Figure 4-4. *An Excel Office document form in design mode.*

How Forms Work

Before we dive into building forms, let's step back and take a look at how forms work inside the Outlook and Microsoft Exchange Server environments. When you double-click on an Outlook item to open it, Outlook queries a property on the item named the message class. The message class uniquely identifies the form that the item is based on. For example, when you create a new e-mail message, you are creating a form with the message class IPM.Note. (The IPM stands for interpersonal message.) When you send the message to another user, the message class travels with the item as a property. You can see all the different message types simply by adding the message class property to your views. These message classes are extensible, so you can create your own types of forms with unique message classes.

When working with forms in the Outlook development environment, you have to base them on built-in forms. You cannot start with a blank slate as you can with Visual Basic forms. After you customize your form, you can publish it. This is where

you can customize the message class. For example, if you modify the standard Outlook Appointment form to make it a class registration system, you can publish the form with its own unique message class, such as IPM.Appointment.Class Registration Form. Although there are multiple message classes, the following list shows the message classes for the built-in Outlook forms:

Form	***Message Class***
Appointment	IPM.Appointment
Contact	IPM.Contact
Journal	IPM.Activity
Message	IPM.Note
Post	IPM.Post
Task	IPM.Task

These message classes work in conjunction with the different forms libraries in Outlook. For example, if a user tries to launch one of your custom forms, Outlook searches the different forms libraries to find it. First Outlook checks to see whether the item is a standard form such as the Note form or a Post form. If the item is not a standard form, Outlook checks its forms cache on the local machine. The forms cache is a folder located on the user's local machine, and by default, Outlook caches all custom forms into this folder to improve performance. When a user launches a form for the first time, Outlook downloads the forms definition into the cache. If you change the form, the version of the form in the forms cache will be updated automatically the next time the user tries to launch that particular form type. This automatic update feature ensures that your users always use the most recent version of the application even after you've modified your forms. The user can change the size of the cache in Outlook by selecting Options from the Tools menu, clicking the Other tab, clicking the Advanced Options button, and then clicking the Custom Forms button.

If the form is not in the cache, Outlook searches the forms library of the current folder. If the form is not in the current folder, Outlook searches the user's Personal Forms Library and the Organizational Forms Library. If the user has Web Services enabled in Outlook, Outlook searches the web forms library.

> **NOTE** Web Services is especially useful if you plan to convert your Outlook forms to HTML forms by using the Outlook HTML Forms Converter. To learn more about Web Services, refer to Chapter 7.

If Outlook cannot find the form in any of the forms libraries, the standard Outlook form on which the custom application is based is used. For example, if a user receives an appointment item with a message class of IPM.Appointment.Job Interview, and the form does not exist in any of the forms libraries, Outlook will use the standard Appointment form to open the message.

Outlook enables you to save the form definition directly with an item, so when a user does not have a copy of your custom form installed in any of her forms libraries or the user is in a different organization, she can still receive your item and view it. Since the form definition is being saved with the message, the size of the message you send to the user will increase slightly. You'll learn how to save the form definition with an item later in this chapter in the section "Publishing Forms."

Data Binding

To retrieve and set the underlying properties of the form, Outlook uses data binding. If you are new to developing with Outlook, it is important to understand data binding because misunderstanding it is often the cause of early design problems.

The layout of the form, or form definition, is separate from the data of the form. Form definitions, then, do not store any application data. Instead, they store data bindings. At run time, Outlook finds the field that the control on the form is bound to, and retrieves and sets the value of the control. The most common mistake new developers make is to add a new control on a form without setting its data binding. If the control is not bound to any field, Outlook does not maintain the data in the Exchange Server database. You will learn how to implement data binding later in this chapter.

DESIGNING FORMS

You need to consider a variety of issues when designing your forms. Once you determine the purpose of your form and which form to modify, you have to open the form in design mode and make decisions about the tasks in the following list. In the rest of this chapter, you will learn how to perform all these tasks.

- Which pages to display on the form
- Whether to separate the read layout from the compose layout
- Which fields to include on the form
- What information you want visible on the form
- Whether to use built-in or custom fields
- What the fields will look like

Outlook provides an environment for creating and editing forms, which is sometimes referred to as the forms designer. The forms designer is automatically installed with Outlook.

Opening a Form in Design Mode

Opening a form in design mode is easy. If the form is not based on an Office document, you can use one of the following two methods: open a standard form of the type you want to modify and enter design mode; or select a form from a list of available forms, which automatically opens the form in design mode. To use the second method, follow these steps:

1. From the Tools menu, point to Forms, and then select Design A Form. Outlook presents you with the Design Form dialog box, as shown in Figure 4-5.
2. From the Look In drop-down list, find the location of the form you want to modify. You can select forms from any forms library as well as from templates in the file system.
3. Click the Details button to display the properties of the currently selected form, which include the icon, description, contact, version, and message class of the form.
4. Click Open to open the selected form in design mode.

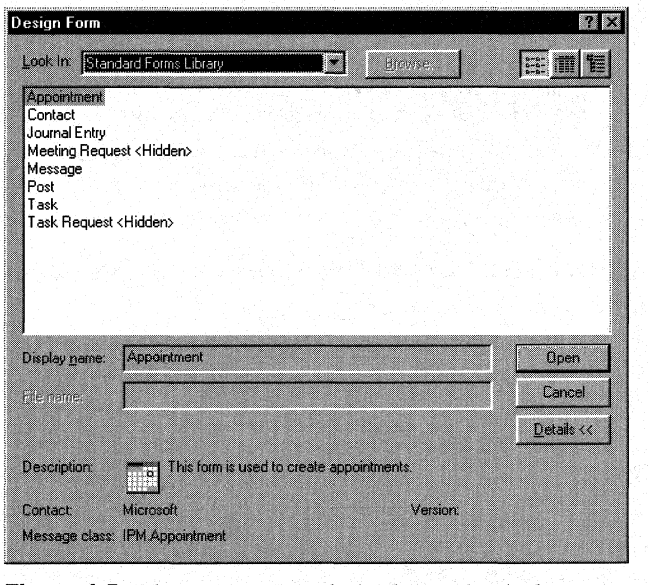


Figure 4-5. *The Design Form dialog box in Outlook.*

To create a new form based on an Office document and open it in design mode, do this:

1. From the File menu, point to New, and then select Office Document.
2. Double-click on the icon representing the type of Office document you want to create.
3. Select either the Post The Document In This Folder option or the Send The Document To Someone option, and click OK.
4. From the Tools menu of the freshly opened document, point to Forms and then select Design This Form to enter design mode.

For more information on customizing Office document forms, refer to the section "Extending Functionality with Office Document Forms," later in this chapter.

Choosing Display Properties

When designing your Outlook application, you need to decide whether you want to display, rename, or hide the default pages on the form. Your decision is based on what you will use the forms for. For example, if you wanted to preaddress an item sent to users to prevent them from modifying the values in the address field or knowing where the item was being sent, you could fill in the address information on the message and then hide the default pages. To change the display properties for a form page or to rename a default or custom page, enter design mode, click on the desired page, and then select the appropriate option from the Form menu. You can quickly see the display status of a form page in design mode since Outlook places parentheses around the name of any page that will be hidden at run time.

Separating the Read Layout from the Compose Layout

Outlook supports having separate layouts for compose and read form pages. The compose page appears when a user opens a form to compose a new item. The read page appears when a user double-clicks on an item to view it. The standard e-mail message is the best example of a form that effectively uses compose and read pages.

Outlook enables you to separate the compose page and the read page from a form so that you can add custom functionality to each of these user modes. Outlook supports compose and read pages on every customizable forms page. By default, the Message page on an Outlook Post form and a Message form have the Separate Read Layout option enabled. However, your custom pages, by default, do not have this option enabled. To enable Separate Read Layout, in design mode, select the form page

where you want separate compose and read layouts. On the Form menu, make sure there is a check mark next to Separate Read Layout. Outlook automatically copies the layout from the compose page to the read page. You can then select the layout you want to modify by displaying the Form menu and choosing either Edit Compose Page or Edit Read Page.

If you find that you are making extensive changes to the compose page, and you want to discard your read page and re-create it with the layout of the compose page, you can disable the Separate Read Layout option and then reenable it. Outlook will copy the layout of the compose page to the new read page.

Using the Field Chooser to Drag and Drop Fields

The Field Chooser provides a simple way to drag and drop built-in and custom fields onto your form. When you drag and drop a field from the Field Chooser onto the form, Outlook creates the appropriate controls. If the AutoLayout option is enabled on the Layout menu, Outlook automatically positions your controls on the form. The controls that Outlook creates are based on the data type of the associated field. For example, if you drag and drop a field with a data type of text, Outlook will automatically create a text box control and typically a label control with the name of the field. If you drag and drop a field with a data type of Yes/No, Outlook will create a check box control with the *Caption* property set to the name of the field. Figure 4-6 shows a Post form with the controls for the Attachment, Categories, From, Icon, and Importance fields added by using the Field Chooser.

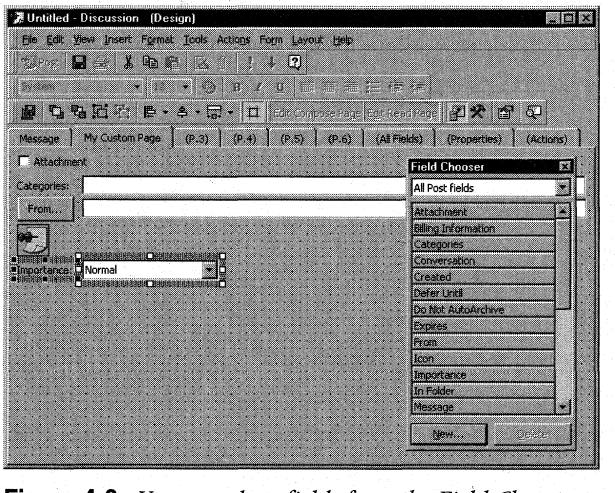


Figure 4-6. *You can drag fields from the Field Chooser to an Outlook form.*

Important Default Fields

Outlook includes some important default fields, such as the address fields, that you should take advantage of when designing your application. In this section, we will briefly discuss the address fields, the Subject field, and the Message field.

Address Fields

The address fields include From, To, Cc, and Bcc. The address fields enable you to preaddress your form to individual users or distribution lists either by typing an address at design time or by setting the initial value of the fields. (You will see how to set initial values later in this chapter.) You can enable these fields if you want to allow the user to change the address, but most likely you will want to disable the To field and just expose the Cc field so that a user can send a copy of the form to another user. By default, the To field is exposed only on a Message form, but you can use the Field Chooser to drag and drop the To field or any other address field onto another type of Outlook form and set its initial value. This ensures that if the user attempts to forward the form, the address you have supplied will automatically appear in the displayed Message form.

If you are preaddressing a form to a folder (as opposed to a user), either the folder has to be exposed in the Global Address List or you have to copy the folder address into your personal address book. To expose the folder in the Global Address List, you must have administrator rights on the Exchange Server system. To expose the folder in the Global Address List, launch the Microsoft Exchange Administrator program. Expand the folder trees to find and select the folder you want to expose. Choose File and then Properties. On the Advanced tab, uncheck the Hide From Address Book check box, and click OK.

SHARED FIELDS

Outlook supports shared fields in an item. Shared fields are any controls that are bound to the same field on both a compose page and a read page of a particular form. A user can modify the field on either page, and the changes will be available universally. Shared fields can also be used between Outlook forms—for example, when you respond to an item in a folder using a custom form, Outlook copies the values from the shared fields in the open item to the same field in the response item.

Subject Field

The Subject field is important for two reasons. First, the text in the Subject field is the caption that appears at the top of a form. Second, the Subject field typically takes its value from the Conversation field, which we briefly looked at in Chapter 5. If you want to have a threaded view of the items in your application, the second point is important to remember.

The value of the Subject field can also be determined by formulas. For example, you can create a formula that sets the value of the Subject field by combining two other fields on your form.

Message Field

The Message field is the only field for which text formatting, attachments, hyperlinks, and objects are supported. This extensive support allows you to embed instructions or other important material as an attachment, or to add shortcuts to web sites, files, or other Outlook items. To add an attachment to the Message field, open the item that you want to modify in run mode. From the Insert menu, select File and find the file you want to insert.

Note that if you are using HTML as your default mail format in Outlook, you can insert files only as attachments or text. If you are using the Outlook Rich Text mail format, you can insert files as attachments, shortcuts, text, or embedded objects. To add file shortcuts using the HTML mail format, you have to add hyperlinks in the Message field that points to your files.

Outlook supports many protocols that you can place in the Message field as hyperlinks. The most useful protocols with examples are listed here:

- *file://*. Use for adding hyperlink shortcuts to any file stored on a file server that your computer can access. Examples include:
  - *file://c:\temp*. Use for linking to files or directories on your own file system, and for pointing at files that are unique to each user's system.
  - *<file://docserver/docs/earning statements>*. Use for linking to files or directories on other servers. Useful for sending users shortcuts to shared files or directories. Notice that the hyperlink is placed in brackets—this is required if the hyperlink contains any spaces.
- *http://* or *https://*. Use for adding hyperlink shortcuts to Internet and intranet web sites. The difference between the protocols is that Hypertext Transfer Protocol, Secure (HTTPS) is a secure version of the Hypertext Transfer

Protocol (HTTP). HTTPS uses the secure socket layers to encrypt the traffic between the web browser and the Web server. Examples of using the HTTP protocols are:

- *http://www.microsoft.com/exchange*. Use for linking quickly to external sites.
- *http://finance/earnings.htm*. Use for linking to internal sites. These internal sites could be individual servers that users set up or part of your organization's intranet.

- *mailto:*. Use for adding shortcuts that allow the user to preaddress a message to the specified location, even if the user is not using Outlook but needs to respond to messages automatically sent by Outlook. You will see an example of this capability when we look at the Exchange Server Routing Objects in Chapter 13. Examples of mailto hyperlinks follow:
  - *mailto:thomriz*. Use if the default mail program of your users is Outlook, which enables you to use the user's simple address. Outlook automatically resolves the name by using an address book.
  - *mailto:thomriz@microsoft.com*. Use when you cannot guarantee that the user of the mailto hyperlink is an Outlook client.
  - *<mailto:thomriz@microsoft.com?subject=Great Book&body=I loved every minute>*. Use to pass additional information after the address of the item. The formatting of the information is exactly the same as when you pass variables along an HTTP query string. The mailto hyperlink automatically fills in the message subject and text.

- *Outlook*. Used to create a hyperlink directly to Outlook information. This protocol is supported only if Outlook is installed on the local machine. Examples follow:
  - *Outlook:Inbox\Subfolder*. Used for linking to Outlook folders. In this example, the *Subfolder* folder of the Inbox will appear on the machine of the user. You can replace *Inbox* with the name of another Outlook folder such as Tasks, Contacts, Calendar, and Journal.
  - *<Outlook:Contacts/~Thomas Rizzo>*. Used to link to a specific item in the folder. This example will open the Thomas Rizzo contact in the Contacts folder of the current user. Remember to place the hyperlink in angle brackets if it contains spaces.
  - *<Outlook:\\Public Folders\All Public Folders\Discussion>*. Used to link to a public folder. This example will open the Discussion public folder. You can also use a similar syntax to open only the

public folder tree or access a user's favorite folders stored in Public Folders\Favorites.

- *Outlook:EntryID.* Used to link to items in the Outlook environment by using the EntryID. This is useful if you are generating mail messages using the Microsoft Event Scripting Agent or the Microsoft Routing Objects and you need to send a link to an item in a public folder. Collaboration Data Objects (CDO), today, does not provide this functionality, but using Outlook:EntryID is a great workaround. The only problem is that the Outlook protocol requires Outlook on the local machine. Here is an example of this protocol:

  Outlook:EF000000D4B32904495CD111921D08002BE4F322646C2700

  This is a short-term EntryID. You can also use long-term EntryIDs.

You can also express spaces in any of these protocols by using the characters *%20*. For example, to link to a specific message in your Inbox with the subject *Earnings reports*, you would use the following syntax for the Outlook protocol: *Outlook:Inbox/~Earnings%20reports*.

The Message control and its underlying field, the Message field, provide extensive functionality to your applications. However, there are some restrictions on usage of the Message control inside of an Outlook form. First, you should create only one Message control on an Outlook form. Since the Message control is automatically bound to the Message field, more than one control will cause a conflict regarding which content should be saved to the Message field. For example, if you have three Message controls on different pages in the Outlook form, and a user writes a different value to each control, Outlook will save the contents of only one of those controls. This means that only one value of the Message field will be displayed in all three Message controls.

> **NOTE** Outlook automatically displays a warning message if you attempt to place multiple Message controls on a form. If you need to sidestep this restriction because your application needs to span multiple pages with Message controls across each page, you should use a MultiPage control and keep only one Message control at the bottom of the form. For more information on how to use MultiPage controls, see the next section, "Using Controls."

Second, only the Message control accepts attachments or the hyperlinks we reviewed earlier in the chapter. The other controls in Outlook do not understand hyperlinks and will not automatically display and link information.

Third, the Message control is always bound to the Message field, and Outlook automatically establishes this binding. You cannot change it, nor can you set initial values of the Message control in design mode. You must either insert your hyperlinks, text, or attachments before designing the form or insert them programmatically.

USING CONTROLS

So far, you've learned how to use the Field Chooser to add controls to a form. The Field Chooser automatically binds the control to the appropriate field. You can also add controls to your form manually.

You can set properties for the controls you use on forms, such as change the name, change the display properties, and bind the control to an Outlook field. You access the properties for a control via the Properties dialog box: in design mode, right-click on the control and select Properties. From the Properties dialog box, shown in Figure 4-7, you can select options that control the behavior and appearance of the different Outlook controls. The following sections describe some of the Outlook controls and how to manipulate some of their properties.

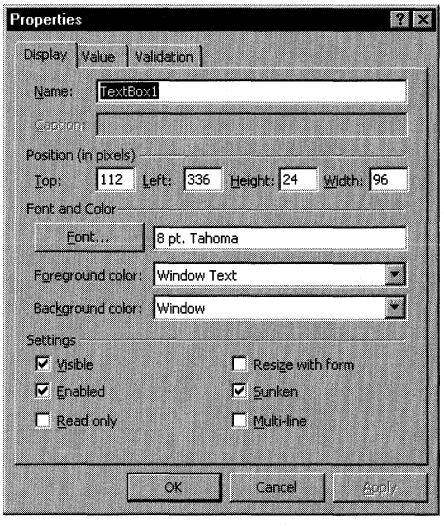


Figure 4-7. *The Properties dialog box.*

Accessing Controls from the Control Toolbox

The Control Toolbox, shown in Figure 4-8, provides an easy way for you to pick controls when you design your forms. To launch the Control Toolbox in Outlook, enter into design mode and choose the Control Toolbox option from the Form menu. You can customize the Control Toolbox to assist you in designing forms more quickly. For example, you could add the controls that you drag and drop most often onto the toolbox. You can customize the Control Toolbox in the following ways:

- Add pages to the toolbox.
- Rename the pages.
- Move controls from one page to another.
- Add other controls, such as ActiveX controls.
- Copy modified controls or groups of controls from your custom forms to the toolbox for use on other forms.

To add controls to the toolbox from an Outlook form, select the controls that you want to add and then drag and drop the controls onto the toolbox. You can select individual controls or groups of controls.

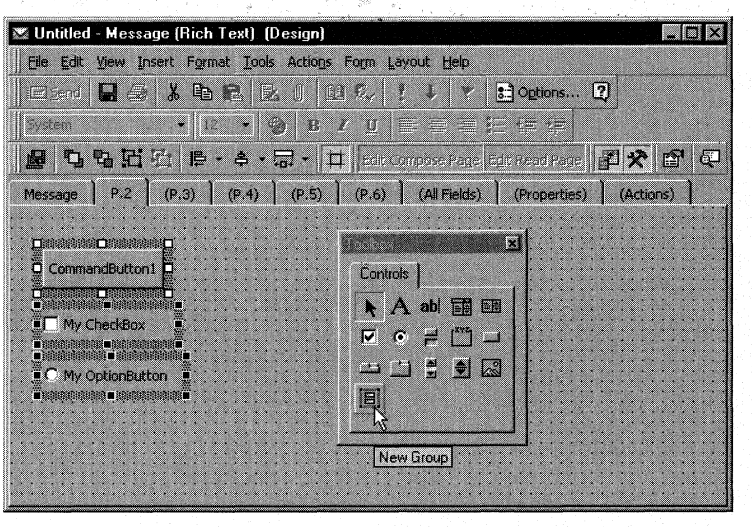


Figure 4-8. *The Control Toolbox and a group of controls that have been dragged and dropped to the toolbox.*

Renaming Controls

By default, when you add a control to an Outlook form, Outlook automatically assigns a name to it—for example, Outlook would assign the name *TextBox1* to a new TextBox control. However, it is good practice to rename the control to a name that better describes its function. Just make sure you give it a name different from the name of the field the control is bound to. By giving the control and the field different names, you can avoid some confusion when writing your applications. Place an abbreviation at the beginning of the control to identify the content type. For example, if you

add a TextBox control that holds the job status information for a potential job candidate, you might want to precede the control name with the abbreviation *txt* to indicate the type. Therefore, a sample TextBox name would be *txtJobStatus*. When extending your form with VBScript, you need to use the name of the control inside your script.

Assigning Captions

The *Caption* property is available to only some of the Outlook controls: CheckBox, CommandButton, Frame, Label, OptionButton, MultiPage, and ToggleButton. You set the *Caption* property via the Properties page. Setting the *Caption* property for each control has a different effect. For the Label and CommandButton controls, the *Caption* property specifies the text that appears in the control. For the MultiPage control, the *Caption* property applies to each page of the control and specifies the text that appears as the tab name.

You can use the *Caption* property to set up accelerator keys (single-character shortcuts) for your controls. For example, you might assign the ALT-A accelerator key combination to a CommandButton control named *Automatically Populate Fields*. To turn any letter in the text of the caption into an accelerator key, place an ampersand (&) before the letter.

For the Outlook controls that do not support captions, such as the TextBox control, you can create individual Label controls for identification. For example, suppose you created a TextBox control that takes the name of a user as input. You could create a label control with the text *User Name:* and position it to appear before the TextBox control.

Setting the Font and Color

Outlook allows you to set the font and color of your controls. The colors you set for your controls can be relative to the colors that your users have set for their system. For example, you can set the background color for a control to be the same as the system window color, which the user establishes.

Using different properties for a control creates unique effects in your application. For example, to create shading, set the background color of a label control and then layer it behind other controls on your form. This effect is shown in Figure 4-9.

> **NOTE** If you want to provide shading for a whole page of your form, use the advanced Properties window of the form to set the *BackColor* property rather than add a label that stretches across the entire page. To learn how to set the advanced properties for a control or a page, see the section titled "Setting Advanced Control Properties" later in this chapter.

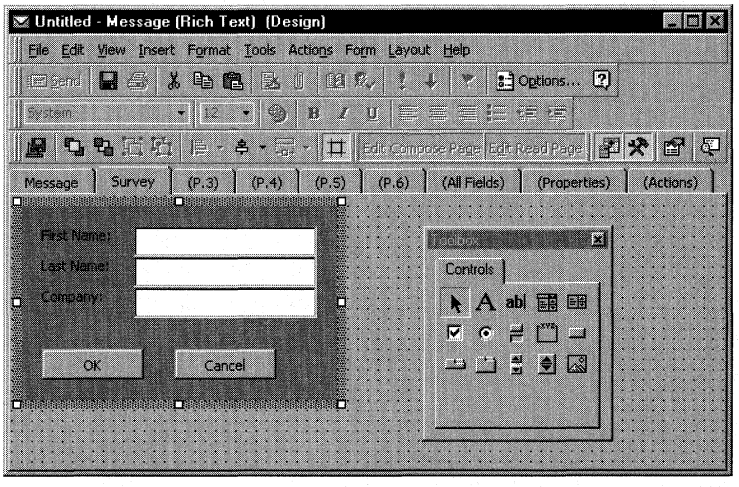


Figure 4-9. *Set the background color of a Label control to provide shading on a certain portion of the form.*

Establishing Display Settings

Outlook provides six settings for the display of a control:

- *Visible*
- *Enabled*
- *Read Only*
- *Resize with Form*
- *Sunken*
- *MultiLine*

With the first three of these properties, you can make your controls behave differently depending on the user accessing the form.

Binding Controls

Sometimes, when you create a new control, you will want to bind the control to an existing field rather than create a new one. To do so, in design mode, right-click the desired control, select Properties, click on the Value tab, and click Choose Field. From the drop-down list, choose the field you want to bind to your control. Since this control is bound to a field, any changes made to the information in the control will be reflected in the field.

Setting Initial Values

You can set initial values for the controls on your form that are either static or calculated via the Value tab of the Properties dialog box, as shown in Figure 4-10. You can also select whether Outlook should use your initial value only when a user is composing an item based on the form, or automatically whenever any of the values used to calculate the initial value change or a user opens the form. For example, you can set Outlook to automatically use your name as the initial value in the Subject field of your form. Whenever a user composes or reads the form, your name will always appear in the Subject field. When calculating initial values, you can use the same functionality used in formula fields. To take advantage of initial values, you must bind your control to an Outlook field.

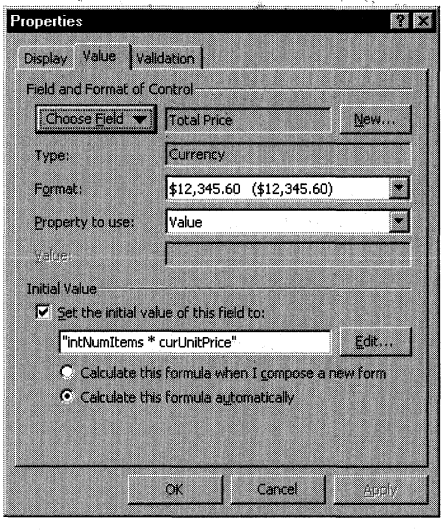


Figure 4-10. *Using a formula in the initial value of a control. This formula calculates the total sale price based on the total number of items times the unit price per item.*

Requiring and Validating Information in Fields

You can require that any field or any control bound to a field have a value. If the user does not enter any information in the field and attempts to close, save, or send the form, Outlook returns an error message that tells the user a value is required.

Outlook can also compare the value the user has entered to the validation criterion you specify. This criterion can range from simple text to complex formulas. If the validation fails, you can specify the message to appear. This message should be used to tell the user what the expected values are in the field. For example, if you create a TextBox control on your form that is supposed to have a number value greater

than 0 in it, you can use a validation formula to make sure that the user does not enter negative numbers in the field. If the user attempts to enter a negative number, your custom message can ask the user to enter a positive number in the field. You set these properties via the Validation tab of the Properties dialog box.

Built-In Outlook Controls

Outlook provides 14 built-in controls that you can add to your forms, each of which provides unique functionality. The following sections introduce these built-in controls, with the exception of the ScrollBar control.

> **NOTE** In addition to the 14 built-in controls, you can also use ActiveX controls. We'll discuss ActiveX controls in the section titled "Using Custom or Third-Party Controls" later in the chapter.

Label Control

Use the Label control to display descriptive text such as titles, captions, company logos, or other identifying information. For example, you would use a Label control to identify a text box or to display read-only information. The only time you typically bind a Label control to a field is when you want to just display a field value. Figure 4-11 shows a timecard application where a user enters the number of hours worked each day. The Label control displays the total number of hours worked.

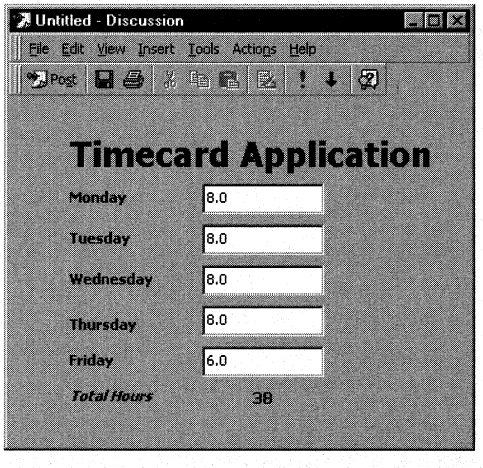


Figure 4-11. *A timecard application that uses a Label control to display the total number of hours worked.*

> **NOTE** You can display pictures inside of Label controls. However, you cannot crop or size the picture unless you use an Image control, which is described later in this chapter.

TextBox Control

Use the TextBox control to display or gather information from a user, gathering information being the more common use. You could use the TextBox control on a customer survey form to gather comments. If you bind the TextBox control to a field, the information entered by the user is saved in that field. Figure 4-12 shows some TextBox controls on a form.

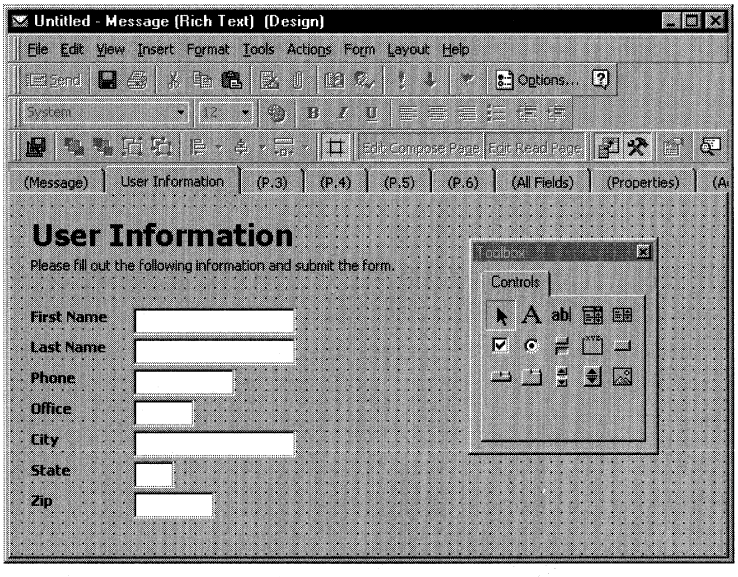


Figure 4-12. *TextBox controls on an Outlook form. These TextBox controls are combined with Label controls to gather information from a user.*

The TextBox control is a highly customizable control. You can govern its functionality by setting specific properties on it. For example, you can make a TextBox control automatically adjust its size to fit text entered by the user. You can also add multiline functionality so that users can enter more than one line of text in the TextBox control. You can enable or disable the *AutoSize*, *MultiLine*, and *WordWrap* properties for a TextBox control by right-clicking on a placed control and selecting the Advanced Properties option.

> **WARNING** Avoid using the *AutoSize* property when using a TextBox control that already has *WordWrap* and *MultiLine* enabled. When the TextBox control is empty, the size of the control will appear only 1 character wide and 1 character high. Further, when the user adds text to the control, the control will automatically resize itself to one long line of text rather than split the text into multiple lines.

ListBox Control

The ListBox control displays multiple values, of which users can select one or more. A ListBox control offers you two different presentation styles:

- Each item in the ListBox appears on a separate row, and the user can select items by highlighting one or more rows.
- Items are presented as option buttons or check boxes. For option buttons or check boxes to appear, the ListBox control must be bound to a Keywords field.

If you set the *MultiSelect* property for the ListBox control on the advanced Properties window to 1 - Multi, the user can select multiple items in the ListBox. When a user selects multiple items in the ListBox, the selections are entered in the field as comma-separated values. You can create views that group or sort by these different values.

You create a list of values in a ListBox control in two ways:

- Right-click on the ListBox control that is placed on the form, select Properties and click on the Value tab. Click the New button, and create a new field that has a type of Keywords. After creating a new field, enter the desired values in the Possible Values box. You must separate the fields with a semicolon. Do not include quotes around your text unless you want these quotes to appear in the ListBox control. Figure 4-13 shows where you set possible values for a ListBox control.
- Establish the values programmatically at run time.

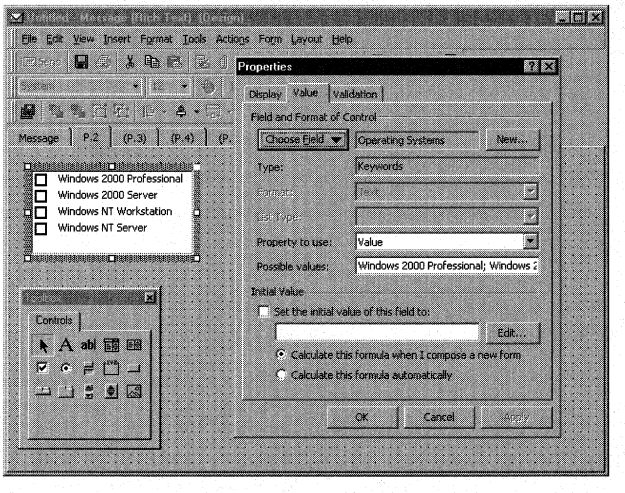


Figure 4-13. *Setting the possible values for a ListBox control. You can see the check boxes that appear when the ListBox control is bound to a Keywords field and the* MultiSelect *property is set to 1 - Multi.*

ComboBox Control

A ComboBox combines the features of a ListBox control and a TextBox control, so it enables you to provide a list from which the user can select an item and a text box into which a user can type information.

You add possible values in the drop-down list of a ComboBox control in the same way you add them for the ListBox control. You can specify the list type on the Value tab of the Properties dialog box, in the List Type drop-down list. Here are the List Type options:

- Dropdown, to allow the user to type new text into the control
- Droplist, to force the user to select a value from the drop-down list

Figure 4-14 shows how properties can be set for a ComboBox control.

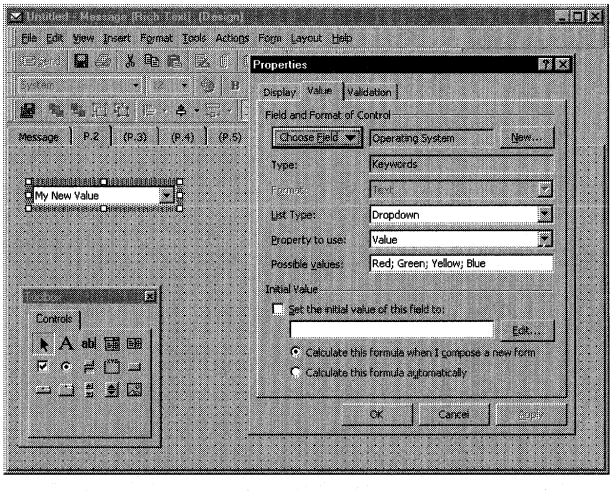


Figure 4-14. *This ComboBox allows users to type in the text portion of the control because the List Type drop-down list is set to Dropdown.*

CheckBox and ToggleButton Controls

Use the CheckBox and ToggleButton controls to give the user a choice between two values, such as on or off, yes or no, or true or false. You should bind the CheckBox and ToggleButton controls to a Yes/No field type for the control to work properly.

OptionButton Control

The OptionButton control is identical to the CheckBox and ToggleButton controls in that it gives the user two choices, but it differs from them in that a group of OptionButton controls are mutually exclusive. For example, on a helpdesk form, OptionButton controls could be associated with possible operating systems users are running, such as Microsoft Windows 2000 Professional, Microsoft Windows 2000

Server, or Microsoft Windows NT, as shown in Figure 4-15. You can bind all of the OptionButton controls to the same field—for example, the txtUserOS field. The value of the txtUserOS field will be the value of the currently selected OptionButton control. You set the value for an OptionButton control on the Value tab.

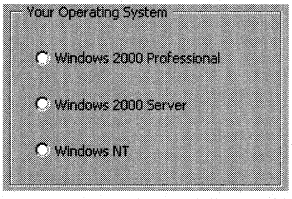


Figure 4-15. *A set of OptionButtons grouped together with a Frame control. OptionButton controls provide mutually exclusive options to users.*

You can group OptionButton controls together using containers, such as the Frame control or MultiPage control. Figure 4-15 showed a Frame control being used to group the OptionButton controls. When you bind one of the OptionButton controls in the group to a particular field, the other controls automatically bind to that same field. Be careful to drag and drop the OptionButton controls onto a container in your form, not onto the form page—Outlook automatically groups all OptionButton controls on the form page together, which might not produce the desired functionality.

You can also group your OptionButton controls by using the *GroupName* property, which identifies related OptionButton controls on a form. To set the *GroupName* property for an OptionButton control, in design mode, right-click on the OptionButton control that you want to modify and select Advanced Properties. Double-click on the *GroupName* property, and type in a unique name for the group. Click Apply, and repeat the steps for all the other OptionButton controls in your group.

> **NOTE** If you want to create a group of OptionButton controls on a TabStrip control, you have to use the *GroupName* property. The TabStrip control is not a container.

Frame Control

Frame controls are used to create groups of related controls. As you saw earlier, a frame can hold a group of mutually exclusive OptionButton controls, but it can hold other types of controls as well. Add controls to and remove controls from a frame by dragging and dropping them.

CommandButton Control

CommandButton controls provide custom functionality when clicked by the user, so you write the script that responds to its click event. The click event is the only event for the CommandButton control. For information on writing scripts in Microsoft Outlook forms, see Chapter 5.

MultiPage and TabStrip Controls

The MultiPage control and TabStrip control are similar in that they offer multiple pages, or tabs, for holding information. The distinction is that every page in a MultiPage control is its own form, so you can customize the layout and background colors of each page as well as place unique controls on them. The TabStrip control must contain the same controls on every page, so you do not have flexibility with layout. Use it if you want a single layout for your data that you can then map a unique set of data to. Figure 4-16 shows how the MultiPage control is used for the Account Tracking application we'll look at in Chapter 6.

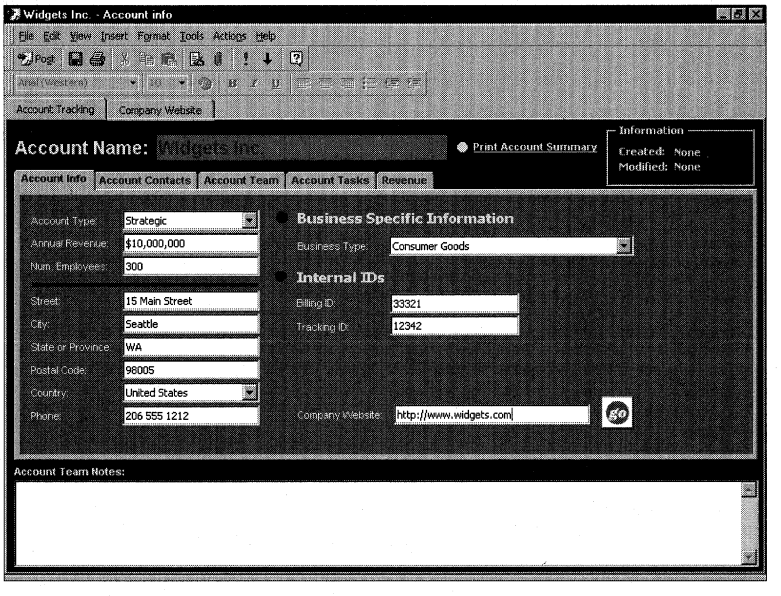


Figure 4-16. *The MultiPage control for the Account Tracking application provides several pages of information.*

To add controls to and remove controls from a MultiPage or TabStrip control, drag and drop them. To customize a page of a MultiPage control, enter design mode, right-click the desired tab, and select Insert, Delete, Rename, or Move. The Insert command always places the page as the last tab, so to position the tab correctly after you've added it, use the Move command.

SpinButton Control

The SpinButton control has arrows that allow you to increment or decrement a number. It accepts custom script—you choose whether or not to write it. If you want to use the SpinButton control to increment and decrement values in another control such as a TextBox control, rather than write script, simply bind the data to the same field for both controls. Figure 4-17 shows the SpinButton control and TextBox control bound to the same field.

Figure 4-17. *Both the SpinButton and TextBox controls are bound to the same field. The user can use the spin button to increment and decrement the value in the TextBox control.*

Image Control

The Image control displays an image on your form, as shown in Figure 4-18. The Image control supports the following file formats:

- .bmp
- .cur
- .gif
- .ico
- .jpg
- .wmf

Here are some of the properties you can set in the advanced Properties window for the Image control:

- *AutoSize*. Control automatically grows or shrinks based on the size of its associated graphic.
- *Picture*. Specifies the picture to display.
- *PictureAlignment*. Determines where the image appears in the control if the height and width of the Image control is greater than the size of the image.
- *PictureSizeMode*. Lets you clip, stretch, or zoom the image.
- *PictureTiling*. Creates picture tiles that fill all available space in the Image control.

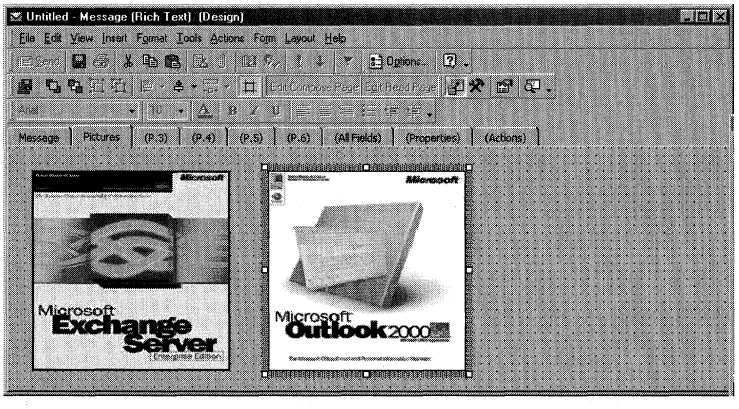


Figure 4-18. *An Image control on an Outlook form displays a picture.*

You can also set the *Picture* property for the Image control by using VBScript in your Outlook form. By setting the *Picture* property programmatically, you can dynamically change the graphic contained in the Image control.

Using Custom or Third-Party Controls

There might be times when the built-in controls of Outlook do not meet the requirements of your application. In these cases, you can extend the Outlook forms environment by adding controls, such as ActiveX controls. You can create these controls by using development tools such as Microsoft Visual Basic or Microsoft Visual C++, or you can use controls developed by third-party companies. Figure 4-19 shows three ActiveX controls placed on an Outlook form.

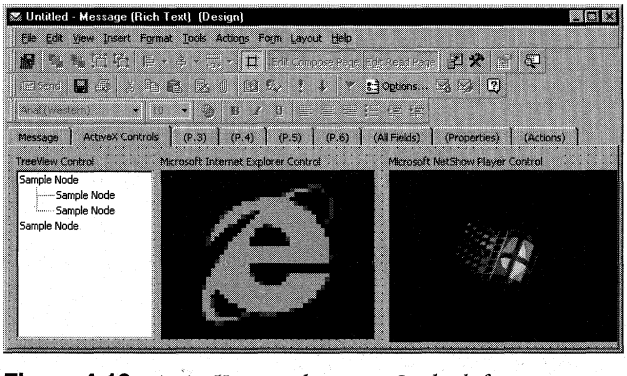


Figure 4-19. *ActiveX controls on an Outlook form.*

To add custom controls to your Control Toolbox, in design mode, right-click on the Control Toolbox and select Custom Controls. In the Additional Controls dialog box, check the custom controls you want to add and click OK.

> **NOTE** For the controls placed on your form to work correctly, they must be available on the computers of your users. Outlook does not automatically distribute ActiveX controls with the form.

When you add one of these controls to your form, you can take advantage of its unique functionality by binding specific properties of the control to Outlook fields, by setting the advanced properties for the control, or by automating the control through VBScript. For example, you can place the TreeView control included with Visual Basic on your form, and then bind the selected node of the tree to your custom text field in Outlook so that you can track which node the user has selected. (You could also capture this information by using a click event handler to place the value in your custom field.)

Usually, you want to bind the default property of the control, typically the value of the control. However, there might be times when you want to bind to other properties in the control. To bind a custom control property to an Outlook field, access the Value tab of the Properties dialog box, select the desired field from the Choose Field drop-down list, and in the Property To Use drop-down list, select the custom property of the control you want to bind the field to.

Different controls support different properties. To learn about the properties of a control, access the help file included with the control or use an object browser, such as the VBA Object Browser, to browse the properties of the control. Using the VBA Object Browser is discussed in Chapter 5.

Setting Advanced Control Properties

There are many times when you need to change the advanced properties of a control on your form, such as the background color of the control or the way the control lists the values contained in it. Below are some of the advanced properties for controls in the advanced Properties window:

- *ForeColor*. This property determines the foreground color of the control, which in turn affects any text associated with the control. This text can be inside of the control or the caption of the control. You can either use selected system colors or create your own color scheme for this property.
- *BackColor and BackStyle*. These properties determine the background of the control. *BackStyle* determines whether a control is transparent. *BackColor* specifies the background color. For this property, you can either use selected system colors or create your own color scheme for this property.

- *BorderColor, BorderStyle, and SpecialEffect.* These properties apply to the border of the control. By using these properties, you can specify whether a control has a unique color around its border or whether a control looks sunken into the form or raised off the form. *BorderStyle* and *SpecialEffect* are mutually exclusive; when you assign a value to one of these properties, Outlook automatically resets the other.
- *Picture.* This property determines the picture on the control. You can set this property for many Outlook controls.
- *ControlTipText.* This property determines the short, descriptive text that appears when a user holds his mouse over the control.
- *MousePointer and MouseIcon.* These properties are used in conjunction to determine the icon used when the mouse passes over the control. You can set the *MousePointer* property to one of 16 built-in icons such as the hourglass icon or the arrow with a question mark icon. When you set the *MousePointer* property to 99 – Custom, the *MouseIcon* property is used to determine which icon to use. You can specify your own icons in the *MouseIcon* property.
- *PasswordChar.* This property specifies the placeholder character that is displayed in the control in place of the real characters. This property is available only for TextBox controls. You can use this property to provide some protection when users enter sensitive information into the form.

Setting the Tab Order

The tab order determines which control the focus moves to when the user presses either the Tab key or Shift-Tab keys. Setting up a logical tab order for your form makes it easier for users to quickly enter information in the controls. To set the tab order, right-click on the desired page on the form, being careful not to click on the controls, and then select Tab Order. In the Tab Order dialog box, select the various controls from the list box and click the Move Up button or Move Down button to adjust the order. You can select more than one control at a time by holding down the Ctrl key and clicking on the desired controls.

> **NOTE** Label controls are included in the Tab Order dialog box. However, at run time, these controls are not included in the tab order.

Layering Controls on a Form

By using the layering capabilities of Outlook forms, you can create dynamic visual effects. For example, you can shade different areas of the forms by creating colored label controls that serve as the background for other controls on the form. Outlook

layers controls by using the z-order (depth) axis, which determines whether controls are in front of or behind other controls. To layer the controls on your Outlook form, in design mode, select the control or controls you want to adjust the order for. From the Layout menu, select Order and then select the desired placement option.

FORM PROPERTIES

Before publishing your forms, you will want to set various form properties. These properties can be default or advanced.

Setting Default Form Properties

Default form properties include the name of the form, the form description, and icons for the form. You set these properties through the Properties page, as shown in Figure 4-20.

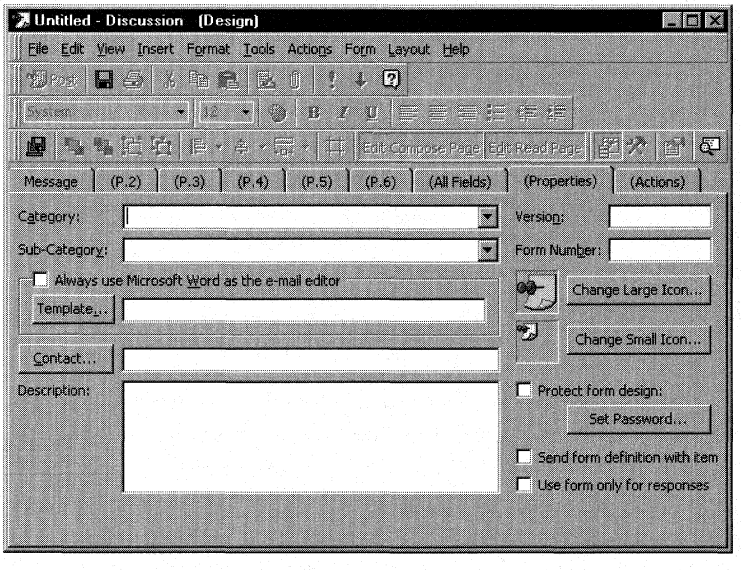


Figure 4-20. *The Properties page in an Outlook form.*

The following list describes the properties you can set on the Properties page:

- *Category.* Allows you to specify a category name for your form, which will ultimately appear in the Choose Form dialog box. Use category names to group similar forms.
- *Sub-Category.* Allows you to specify a subcategory for your form so that you can divide your forms into groups. For example, a marketing forms category could be divided into advertising, events, pricing, and promotions.

- *Always Use Microsoft Word As The E-Mail Editor.* Allows you to specify Word as the editor for the Message control of the form. By selecting this property, users can take advantage of the advanced features of Word.
- *Template.* Allows you to specify which Word template to use for the Message control on your form.
- *Contact.* Specifies the contact responsible for the form. This name appears in the Choose Form dialog box if the user selects to view the details about the form, and in the About dialog box for this form when the user selects About This Form from the Help menu. (The About This Form option is not available if you send the form definition with the item.)
- *Description.* Allows you to enter the form description, which will appear in the Choose Form dialog box if the user selects to view the details about the form, and in the About dialog box for this form.
- *Version.* Allows you to specify a version number for the form. This version number will appear in the Choose Form dialog box and in the About dialog box for this form.
- *Form Number.* Allows you to specify a number for the form, which appears in the About dialog box for this form.
- *Change Large Icon.* Allows you to specify the large icon that will be used when a user selects a large icon view for any folder containing this custom form.
- *Change Small Icon.* Allows you to specify the small icon that will appear in the default, custom table, timeline, or icon view of the form.
- *Protect Form Design.* Allows you to specify a password to protect the design of your form after publishing it. You should do this for all of your custom forms unless you want to give users the ability to modify or customize your form. For example, you should not password-protect business form templates, such as a joint marketing agreement form, that users will have to modify to meet their specific needs.
- *Send Form Definition With Item.* Allows you to save the form layout with the item. Use this option if you are not going to publish the form in a forms library. When you save the form definition with the item, Outlook considers these forms to be one-off forms, meaning that if you modify the form later and publish it to the forms library, any older items will continue to use the form definition saved inside of the item. This feature is useful for sending forms to users who are not in your organization or who do not

have a copy of the Outlook form. For example, you would use this property if you wanted to create an event registration form that you wanted users both inside and outside of your organization to use. Since this form is not useful beyond the event, you would not publish it into a forms library. Note that when you save the form definition with the item, Outlook displays a security warning if the form contains VBScript and is not published anywhere in the user's Outlook system.

- *Use Form Only For Responses.* Makes your form available only as a response to other forms, so users cannot create your form directly from the Choose Form dialog box. Instead, they must open the parent form and use the methods provided in the parent to create the form. This option is useful if you want to create hidden forms for your application. (Outlook uses this feature to implement meeting responses.)

Setting Advanced Form Properties

In addition to setting default form properties, you can set advanced form properties to help you create visually appealing forms. These advanced properties allow you to specify the individual background colors or images for the pages of your forms and also the default mouse pointers. To set the advanced properties of your forms, in design mode, right-click on the desired page and select Advanced Properties. (Be sure to click on the page, not on a control.) In the Advanced Properties window, choose the options you want to modify. Figure 4-21 shows the Advanced Properties window.

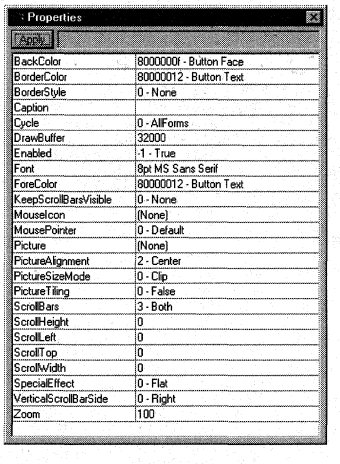


Figure 4-21. *The Advanced Properties window for a page on an Outlook form.*

TESTING FORMS

Outlook enables you to test your forms as you develop them. Since the design and run environment are built into Outlook, you do not have to compile or save your forms before testing them. You can start a separate run mode so that you can test the form's functionality while making changes to it in design mode. You can also enable multiple instances of your forms in run mode, which is useful if you want to try different versions of the form and test different areas of functionality. To test your forms in run mode, enter design mode and select Run This Form from the Form menu. Outlook will automatically create a new instance of your form in run mode. To get back to design mode, just close the running instance of the form. Figure 4-22 shows an Outlook form in both design and run modes.

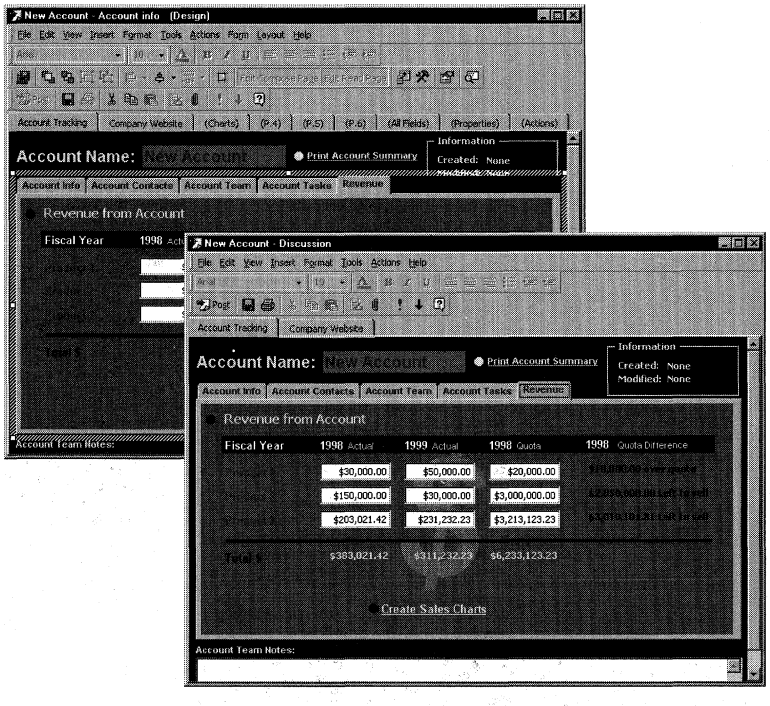


Figure 4-22. *An Outlook form both in design and run modes. These two separate modes make it easier to test your Outlook applications.*

PUBLISHING FORMS

After customizing your forms, you need to make them available to your users. There are three primary ways you can distribute your forms to users:

- *Publish the form in a forms library.* The forms library can be the Organizational Forms Library, a folder forms library, your Personal Forms Library, or a Web forms library. This is the most common distribution method.
- *Save the form definition with the item, and send the item to your users.* This option is best suited for one-off forms where users need to use the form only once.
- *Save the form to the file system as an .oft file, and attach it to an e-mail that you send to your users.* Your users can then use the form from their file systems or publish the form in their Personal Forms Library.

The following sections describe each of these methods in detail.

Publishing Forms in a Forms Library

As you learned in Chapter 2, Outlook supports four types of forms libraries, and each type meets a specific need for forms publishing:

- *The Organizational Forms Library.* Use this library to publish public forms that should be available to the entire company.
- *Folder forms library.* Use this library to publish forms in specific folders. To compose a form from this library, users must either click on the folder in the Choose Form dialog box or open the folder and launch the form from the Actions menu. Any user can create forms in his or her own personal folders, and users with editor, publishing editor, or owner permissions can create forms in a public folder.
- *Personal Forms Library.* Use this library to publish personal forms. This library, which is stored on the local machine, cannot be shared with other users in the organization. Publish personal templates and forms to this library, and test your forms there before deploying them to your users.
- *Web forms library.* A web forms library is stored using Microsoft Outlook Web Access. It can contain Outlook forms that were either converted to HTML or that you created using HTML.

Note that Outlook also allows you to create personal folder files (.pst files). These files implement the same functionality as your personal mail folders, so you can create new folders in these personal store files and publish forms to the folders. Since you can save the forms to your local hard disk, you can e-mail or copy them to a floppy for distribution.

To publish your forms to a forms library, follow these steps:

1. In design mode, from the Tools menu, point to Forms and then select the Publish Form As option.
2. From the Look In drop-down list, select the forms library where you want to publish the form.
3. In the Display Name box, type the friendly name of your form. Outlook automatically fills in the Form Name box for you. (Note that this name will appear in the caption at the top of the form.) If you want to use a separate form name from the display name, type a name in the Form Name box.
4. Click Publish.

Saving the Form Definition with the Item

As you learned earlier in this chapter, you should save the form definition with an item when you know that the users will not have the form anywhere in their systems. If the definition is saved with the item, Outlook uses the saved version of the form, which is the most current. If the definition is not saved with the item, Outlook looks for the form definition in other locations. To save the form definition with the form, in design mode, click the form's Properties tab. Check the Send Form Definition With The Item check box to enable it.

You need to keep two issues in mind when you consider whether to save the form definition with the item. The first issue is security—particularly when VBScript is used to customize the form. To alleviate security concerns, Outlook provides a security measure when users receive an item with a form containing VBScript. Since Outlook supports customizing forms with VBScript, this is a necessary precaution. Without it, users could send malicious forms containing VBScript which could, for example, delete data on your hard drive. This security measure displays a warning message box, as shown in Figure 4-23, allowing the user to either enable or disable the VBScript in the form. This security warning will appear only if the form has the definition saved with it, is not published in any of the forms libraries, and has VBScript included with it.

The second issue to note is that when you save the form definition with the item, you cannot take advantage of the automatic update capabilities of Outlook forms. For example, if you change the form, the new version of the form will be included only with new items you create based on it. Any old items will use whatever form definition was originally saved with the item.

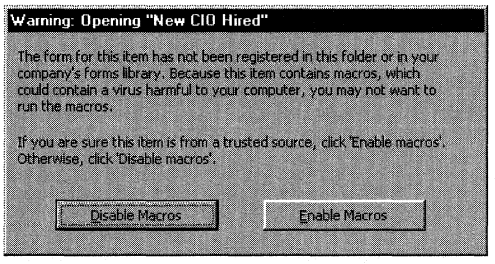


Figure 4-23. *The warning message that is displayed when a form has the definition saved with the item and also contains VBScript.*

Saving the Form as an .oft File

Outlook allows you to save your forms as Outlook template, or .oft, files. This enables you to embed the form in a mail message and send it to users who are both internal and external to the organization. Your users open the form using the attachment, and they either return the form completed or publish the form in a forms library. Saving your custom forms as .oft files is one way to create backups of your custom forms. To save a form as an .oft file, in design mode, select Save As from the File menu. In the Save In box, select the location to which you want to save your file. In the File Name box, type a name for your file.

ENHANCING FORMS

You can enhance your forms and applications by taking advantage of additional features available in Outlook. If you have any Microsoft Office products installed, you can tie into some of their capabilities. You can also add actions to your forms by customizing Reply and Forward and by using voting buttons. In this section, we'll take a look at these features.

Extending Functionality with Office Document Forms

Office document forms are similar to built-in Outlook forms, but there are enough differences to warrant highlighting this technology in its own section. With Office document forms, you can take advantage of the power of other Microsoft Office products in your applications as well as any development skills you already have. To create an Office document form, select Office Document from the File/New menu. Select the type of document you would like to create. Outlook will prompt you to

either e-mail or post this document to someone else in your organization. Select the action that best suits your application: if you need to e-mail your new form to co-workers, create a new e-mail Office form; if you're posting the information to a public folder for others to read at their convenience, post the new Office form.

After creating your Office document form, you can customize the form by using either VBA inside of the Office document or VBScript inside of Outlook. Unlike the standard Outlook forms, Office document forms won't let you add new tabs. Therefore, the only way to customize the Office document form is to use the tools available for the particular Office application you used to create the document form. For example, to customize the values and layout in an Microsoft Excel spreadsheet form, you use Excel tools. A customized Excel document form is shown in Figure 4-24.

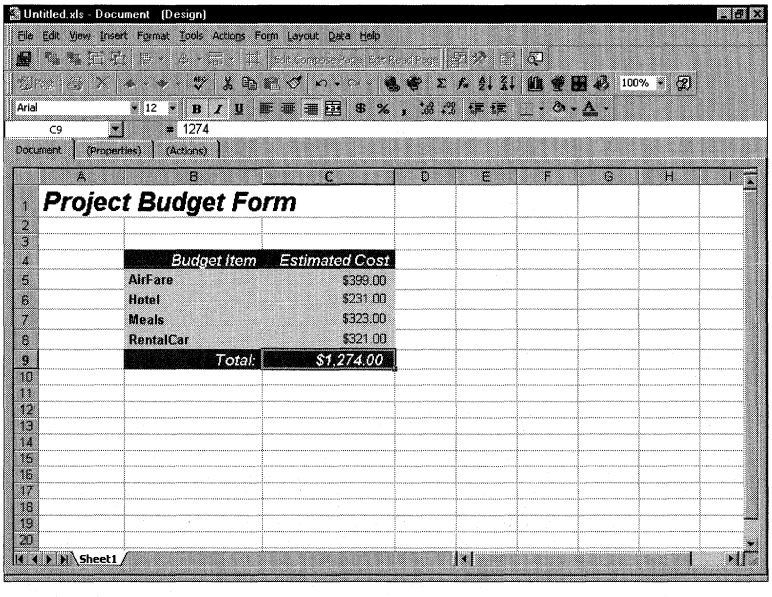


Figure 4-24. *An Excel document form in design mode. Notice how the first page has been customized using the tools provided in Excel.*

You can also use the custom properties inside your Office forms as Outlook view properties. For example, an Excel expense reporting form could use its cell values as properties in an Outlook view. This sharing of properties is implemented by creating custom properties in the Office application. You create custom properties in

Word and in PowerPoint the same way, except that in Word, the custom properties link to custom bookmarks, and in PowerPoint, the custom properties link to objects in the presentation. Following is a description of how to create custom properties in Excel:

1. In your Office document form, which is based on an Excel Worksheet, select a cell that contains the value you want to use as a custom property in your Outlook view.
2. From the Insert menu, point to Name and then select Define.
3. In the Names In Workbook box, type the custom name you want for the cell, such as *Total*, and click Add. Click Close.
4. From the File menu, select Properties.
5. Click the Custom tab. In the Name box, type the name for your custom property. This can be the same as the name you selected for the cell.
6. From the Type drop-down list, select the type that corresponds to the values contained in the cell. For example, for a *Total* field, you should select Number.
7. Check the Link To Content check box to enable it. From the Source drop-down list, select the custom cell name you created in step 3.
8. Click Add. In the Properties box, you should see the name, value, and type of the property you just added.
9. Click OK.

After you create a custom property to hold the value from the Office document, you can add it to your Outlook view. To do this, select the folder where you posted or e-mailed the Office document form. Right-click on the column headings for the view, and select Field Chooser from the menu. In the Field Chooser, open the drop-down list and choose User-Defined Fields In Folder. The *Total* field should be listed. You can use this property to group, sort, and filter your view. The only restriction on using this property is that you cannot edit it using in-cell editing in Outlook. The properties are read-only inside of Outlook, and you must use the Office document form to edit the properties. Figure 4-25 on the next page shows a view of expense reports inside an Outlook folder.

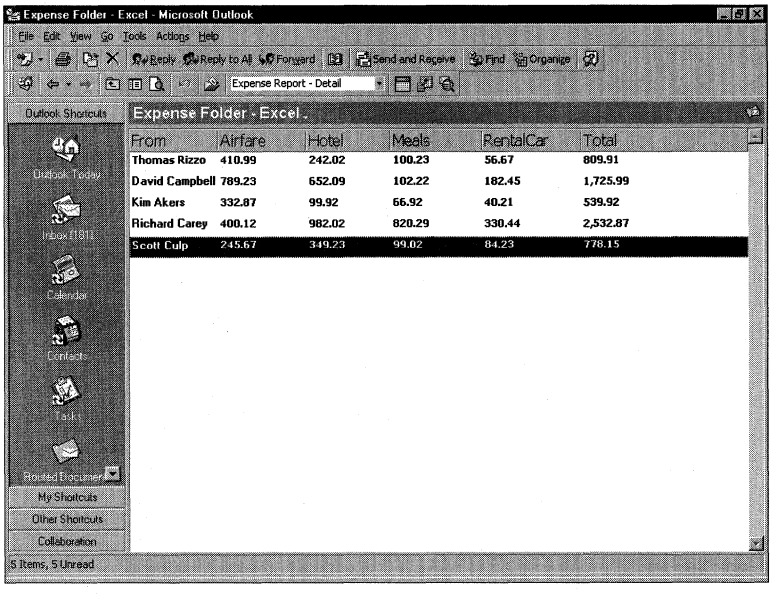


Figure 4-25. *A view of expense reports inside an Outlook folder. The properties in the view are actually values from an Excel document form.*

In design mode for Office documents, you can set the same form properties that you can set for default Outlook forms. These properties include changing the icon for the form, password-protecting the form, and establishing the form contact. For more information on these properties, see the section "Setting Default Form Properties" earlier in this chapter.

> **NOTE** Outlook also allows you to create custom actions for the form, which is discussed in the next section.

In the same way you can publish custom forms, you can publish your Office document forms so that they are available to all users in an organization. Office document forms display a unique message class determined by the Office application that they are based on. For example, an Excel expense report form could have a message class named IPM.Document.Excel.Sheet.8.Expense Report, and a Word beta agreement form could have a message class named IPM.Document.Word.Document.8.Beta Agreement.

Creating Actions

Actions are either built-in or custom responses to a particular item on a form. They add dynamic functionality to your application with no coding at all. For example, in the Account Tracking application, individual contacts for the company can be created

off the master company contact information. In a threaded discussion application, new items are created as responses to posted items. The following section explains how to create actions for Outlook items and associate those actions with custom forms, and it provides strategies for using actions in your applications.

By default, Outlook provides you with four built-in actions for items: Reply, Reply To All, Forward, and Reply To Folder. In many cases, you'll want to customize these actions or create your own. Figure 4-26 shows the Actions page, in which you create actions.

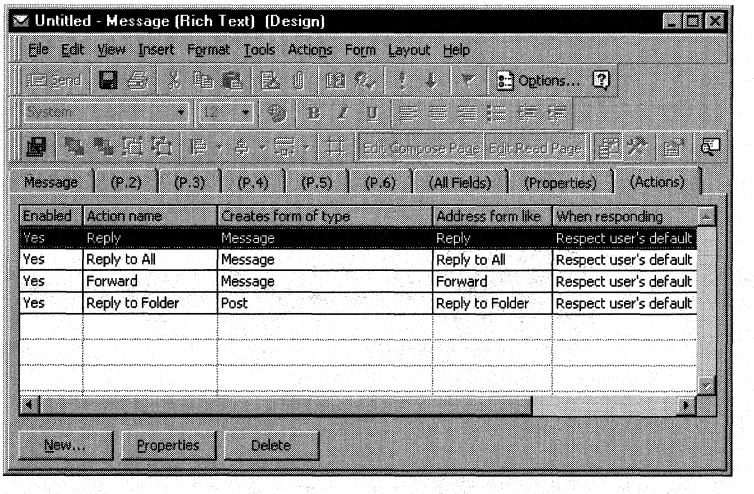


Figure 4-26. *The Actions page of a form.*

To create a new action, follow these steps:

1. In design mode, click on the Actions tab.
2. Click the New button at the bottom of the form.
3. Type the name of your custom action in the Action Name box. This name is used by Outlook to display your custom action on the form or menu.
4. From the Form Name drop-down list, select a custom form. By default, Outlook displays the custom forms in the active folder. If the desired form is somewhere besides the active folder, you can locate it in one of two ways:
   - ❑ Select the Forms option to launch the Choose Form dialog box. You can see and select all hidden forms in the different forms libraries.
   - ❑ Type the name of a form in the drop-down list. After you select the form, Outlook automatically enters its message class in the Message Class box.

5. From the When Responding drop-down list, select the method by which the contents of the original item will be copied to the new form. These are your options:

   - *Do Not Include Original Message.* Outlook does not include the original item in the action.
   - *Attach Original Message.* Outlook will attach the original message as an icon in the message body itself or in a separate window at the bottom of the message. The mail format of the attached message will be based on the user's settings.
   - *Include Original Message Text.* Outlook includes the original message text and some carriage returns before the text in the message body of the form.
   - *Include And Indent Original Message Text.* Outlook includes the original message text, indented in the message body of the form.
   - *Prefix Each Line Of The Original Message.* Outlook prefixes each line of the message with the default prefix character the user selected from the Options/E-mail Options menu. By default, the character is a greater-than sign (>).
   - *Attach Link To Original Message.* Outlook attaches a shortcut to the original message. This functionality is useful when the reply item is a message form that is sent to the user and the original item is posted in a public folder. It allows the user to double-click on the reply item and quickly find the original item without searching the public folder tree for the folder.
   - *Respect User's Default.* Outlook uses the settings for replying and forwarding messages that were selected by the user from the Options/E-mail Options menu.

6. From the Address Form Like A drop-down list, choose the way you want to address the form. These are the address options:

   - *Reply.* Outlook copies the contents of the original From field to the To field in the reply form. No Cc and Bcc information is copied. The Subject field contains the text from the original item.

- *Reply To All.* Outlook copies the contents of the original From field to the To field in the reply form as well as Cc information. The Subject field contains the text from the original item.
- *Forward.* Outlook clears the address information in the reply form so that the To, Cc, and Bcc fields are empty. However, Outlook does copy the subject of the message to the reply form.
- *Reply To Folder.* Outlook places in the Post To field the active folder so that the reply automatically posts to the folder. Outlook also copies the subject of the message into the Conversation field and clears the Subject field. This allows you to create threaded views of your items.
- *Response.* This option is used only with voting buttons.

7. Check the Show Action On check box to make your custom action appear either on both the Actions menu and the toolbar or just on the Actions menu.

   NOTE In some cases, you will want only the action to appear on the menu. For example, if you create a CommandButton control on your form and program the CommandButton control to execute the action when clicked, you might not want the action to also appear in the toolbar. In other cases, you will not want the action to appear on either the toolbar or the menu—for example, when the action is used in an event such as the close event for an Outlook form. In this case, you would not want users to be able to launch this action, only your application.

8. Select an option from the This Action Will area. The default option, Open The Form, will probably be the one you use most often, but you can choose options to send the form immediately, or to prompt the user about whether to send the form or open it for modification.
9. In the Subject Prefix box, enter the characters that should precede the Subject in the reply form. Outlook will automatically place a colon after the characters.
10. Click OK. That's it. Now you have custom functionality on your form without writing a single line of code.

Modifying and Disabling Actions

In addition to creating new actions, Outlook allows you to modify and disable built-in and custom actions so that you can control which actions are available to the user. When you disable a built-in action such as Reply All, you can replace it with your own. Because disabling doesn't actually delete the action, you can reenable it. You can also modify a built-in action so that its functionality is consistent with the functionality of your application.

You modify, disable, and delete built-in actions via the Actions page. To modify or disable the action, select the desired action on the Actions page and click the Properties button to access the Form Action Properties dialog box. To delete an action, select the desired action and click the Delete button. Once you delete an action, you have to re-create it to get it back. If there is any chance that you will need the action in the future, you're better off disabling it.

ACTIONS AND HIDDEN FORMS

Actions can take advantage of hidden forms in your applications. Remember that checking the Use Form Only For Responses check box on the form's Properties page hides the form from the user when the user chooses a form to compose. Suppose you are writing a helpdesk application and on your form you want to include an action called "Resolved." The Resolved action launches a new form, which the technician fills out to mark a ticket as resolved. Suppose also that you do not want users creating their own resolved forms. You can hide the resolve form by marking it as a response form and then setting permissions on the folder that prevent users from reading the items posted there. Technicians can open the items and click the resolved action to resolve the ticket. Users will not see the resolved form in their Choose Forms dialog boxes.

Chapter 5

Programming Outlook with VBScript

In Chapters 3 and 4, you learned about folders and the form design tools that assist you with developing collaborative applications in Microsoft Outlook. To help you further extend your applications, Outlook provides a built-in development environment that uses the Microsoft Visual Basic Scripting Edition (VBScript) programming language. Using VBScript, you can write procedures to manipulate items, folders, controls, and the other objects Outlook contains in its object library. VBScript also lets you automate applications, such as Microsoft Excel, inside your own application so that you can take advantage of their functionality.

The examples in this chapter are VBScript examples, as are many other examples in the book—Chapter 12 on Collaboration Data Objects (CDO), Chapter 13 on event scripting, Chapter 14 on routing objects, and Chapter 15 on Active Directory Services Interfaces (ADSI) are all interlaced with VBScript examples. By learning the VBScript language, you can easily use the different tools and APIs provided by Microsoft Exchange Server and Outlook. This is the power of the Exchange Server platform: you need to learn only one language to develop applications in many different contexts.

While this chapter is not an extensive tutorial on VBScript, it does provide you with the requisite amount of information on the language to work through the examples in this book. For more information, check out the VBScript help file available on the companion CD. It has a language reference section and a simple tutorial. There are also many great books published by Microsoft Press that cover VBScript in more detail as well as Web sites that provide excellent information. One Web site in particular has up-to-date information on VBScript and VBScript-related files available for downloading: *http://msdn.microsoft.com/scripting*.

You can, of course, use other development languages to create solutions that take advantage of Outlook, CDO, and the other Exchange Server tools. However, VBScript is integrated with the Outlook environment, and Outlook provides a number of great tools that take advantage of this integration.

THE OUTLOOK SCRIPT EDITOR

While creating forms and actions in the last two chapters, you might have wondered what the View Code option on the Form menu did. If you had selected the View Code command, the Outlook Script Editor would have appeared, showing you firsthand how VBScript is integrated with Outlook. Figure 5-1 shows the Script Editor displaying code from a sample application.

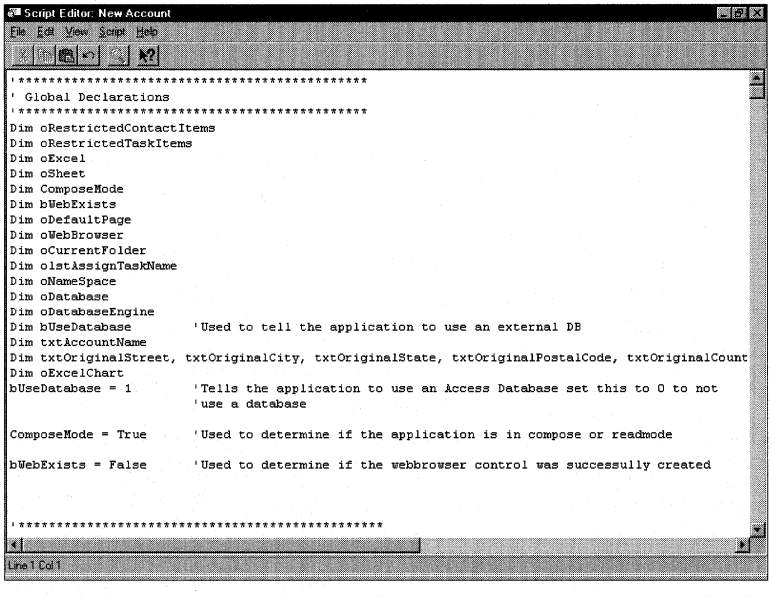


Figure 5-1. *The integrated Outlook Script Editor.*

The Script Editor allows you to add custom VBScript procedures and variables to your Outlook applications as well as use Outlook objects in your applications. Using

the Script Editor, you can write VBScript to handle Outlook events. The Script Editor provides an Insert Event Handler dialog box, which lets you select the event you want to write a handler for. We'll discuss Outlook events more later in this chapter.

Outlook provides line numbers when reporting errors, and in the Script Editor, you can jump to a specific line of code. This feature makes it easier to debug any errant code in your application. To use this functionality, in the Script Editor, select Go To from the Edit menu, specify the line number, and click OK. Your cursor will be inserted at the specified line.

VBSCRIPT FUNDAMENTALS

VBScript should be familiar to any developer who has experience with Microsoft Visual Basic or Microsoft Visual Basic for Applications (VBA) because VBScript is a subset of them. Version 5.1 of VBScript, which includes many enhancements from previous versions, ships with Outlook 2000. As you would expect, you write VBScript applications in the same way you write applications in other programming languages—by using variables, procedures, and objects.

> **NOTE** For more information on the different versions and features of the VBScript language, visit *http://msdn.microsoft.com/scripting.*

Working with Variables

Variables in VBScript correspond to locations in memory where you can store information while your application is running. Variable names are easily identifiable words or phrases such as *myColor*, *myObject*, and *myTotal*. There are some restrictions on the naming of variables inside of the VBScript environment, including the following:

- The variable name must begin with an alphabetic character. For example, *myTotal* is a legal variable name, but *$Total* is not.
- The variable name cannot contain an embedded period. For example, *this.total* is not a valid variable name.
- The variable name cannot contain more than 255 characters.
- The variable name must be in a unique scope when declared. (Scope will be discussed shortly.)

Declaring Variables

To make variables available to your application, you need to either explicitly declare the variable or have the VBScript language implicitly declare the variable for you. The easiest way to declare a new variable is to use the Dim statement inside your VBScript code. For example, to declare three new variables, you could write either of the code fragments on the next page.

```
'Declaring variables using separate lines
Dim myColor
Dim myObject
Dim Total

'Or you can use the same line and separate the names by commas
Dim myColor, myObject, Total
```

VBScript does not require you to explicitly declare variables. If you do not declare a variable and you use that variable in your code, VBScript will automatically save storage space for your data and use your variable as the friendly name for that storage space. However, if you accidentally misspell a variable in your code, errors are likely to occur in your application. These errors are particularly hard to track down because the VBScript interpreter does not know that a variable has been misspelled. To make these types of errors more manageable, VBScript provides the Option Explicit statement. The Option Explicit statement forces all VBScript variables to be explicitly declared. If you do not declare all your variables, VBScript will display an error. The Option Explicit statement must appear before any procedures and is typically the first statement in your code.

Scope and Lifetime of a Variable

The lifetime of a variable is determined by the scope in which the variable was declared. There are two levels of scope for a variable inside VBScript:

- Global, or script-level scope
- Local, or procedure-level scope

Global variables can be called from any procedure inside the running script. To create global variables, all you need to do is declare your variables outside any procedure in your script. It is best to group all global variables at the top of your script instead of scattering them throughout your code.

Local variables are declared within procedures and can be accessed only by the code in that procedure. If you attempt to call one of these variables from other procedures, VBScript displays an error. Also, procedure-level variables in different procedures can have the same name.

The next code snippet shows both script-level and procedure-level variables being declared. To test this code, create a new Message form in Outlook and enter design mode. Drag and drop a CommandButton control onto the second tab of the Outlook form. The CommandButton control will automatically be named *CommandButton1*. Display the Script Editor by selecting View Code from the Form menu. Type the following code in the Script Editor. When finished, try running the form by choosing Run This Form from the Form menu and then clicking the button on the second tab.

```
'Make sure to Dim our variables before using them
Option Explicit
'Global/script-level variables
Dim strName
strName = "Joe User"

Sub CommandButton1_Click
Dim strLocation  'Procedure-level variable
strLocation = "Seattle, Washington"
msgbox strName & " is located in " & strLocation
End Sub
```

The length of time a variable exists is called its lifetime. The scope in which the variable is declared affects the lifetime of the variable. For example, if a variable is declared with script-level scope, it will exist the entire time the script is running. If the variable is declared with procedure-level scope, that variable will be created when the procedure begins and destroyed when the procedure ends. This is why procedure-level variables are good as temporary storage space inside your VBScript procedures. Some restrictions apply to VBScript variables and their lifetimes:

- You can have up to 127 procedure-level variables. Arrays count as only one variable.
- You can have up to 127 script-level variables.

The following code snippet demonstrates variable lifetimes:

```
'Make sure to Dim our variables before using them
Option Explicit
'Global/script-level variables
Dim strName
strName = "Joe User"

Sub CommandButton1_Click
Dim strLocation  'Procedure-level variable
strLocation = "Seattle, Washington"
msgbox strName & " is located in " & strLocation
End Sub

Sub CommandButton2_Click
'Attempt to use variable from the previous procedure
msgbox strLocation
End Sub
```

When you run this code, click CommandButton1, and then click CommandButton2. You will receive a "Variable is undefined" error message from Outlook because the variable *strtLocation* is a procedure-level variable, and it is destroyed after the *CommandButton1_Click* procedure is complete.

Data Types in VBScript

If you're a Visual Basic or VBA programmer, you've probably been wondering how in VBScript you can explicitly declare variables as different data types. Well, the answer is you can't because VBScript supports only one data type, Variant. The Variant data type is special in that it can hold many categories of information, such as text, numbers, dates, times, floating point numbers, and objects. These categories of the Variant data type are called subtypes. The Variant data type works in such a way that VBScript can figure out what subtype to use based on the information—for example, if you place a number in a VBScript variable, VBScript assumes the subtype should be numeric and treats it as a number.

By using the built-in conversion functions of VBScript, you can turn any variable into different subtypes. For example, with the *CStr* function you can convert an expression into a string. By using the *VarType* function, you can obtain the current subtype for a variable.

Working with Objects

You work with objects in VBScript in the same way you work with variables except for one difference: when working with objects, you use the Set statement to set the variable to point to the object. When you work with variables, you do not use the Set statement. To illustrate why, consider the previous code example in which you set the *strtName* variable to the string *Joe User* by using the following statement:

```
strName = "Joe User"
```

For an object, the syntax would be different. Let's look at an example in which a hypothetical object named Information has a property named *UserName*. To access this object in VBScript, you first need to set a variable to the object and then use the dot (.) operator to access the specific object property. You can also use the dot operator to access a specific method on the object. Following is a code snippet that shows the variable *myInformation* being set to the Information object. Using the *myInformation* variable, properties and methods can be accessed with the dot notation.

```
'The variable that will hold the Information object
Dim myInformation

'Set a variable to the Information object
Set myInformation = Information
'Initialize the UserName property of the Information object
myInformation.UserName = "Michael Dunn"
'Get the UserName property of the Information object
msgbox "Current User is " & myInformation.UserName
'Call the Print method of the Information object
myInformation.Print
```

EARLY BINDING VS. LATE BINDING IN VBSCRIPT CODE

Binding is a term that describes the association of a variable with an object. In Visual Basic, there are basically two times in which this binding can occur, compile time and run time. The binding that occurs at compile time is called early binding. The advantage of early binding is that the compiler can perform some data-type and function-name checking. Binding at run time, or late binding, has the advantage of being flexible because the object does not have to be explicitly specified at compile time. The disadvantage of late binding, however, is that it requires additional code, which does make it slower than early binding. Since all variables in VBScript are declared as Variant data types, you cannot take advantage of early binding in your code. Instead, VBScript uses late binding.

Constants in VBScript

When using VBScript in Outlook, you sometimes must refer to predefined constants, such as olMailItem. You can't use the constant by name, but rather you must use the constant's numeric value. For example, if you wanted to use the constant olMailItem in your code, you would need to use the value *0*. VBScript does support user-defined constants and some intrinsic constants, but if the constants are defined in another file, you must either use the constant's numeric value or explicitly declare the constants in your code. You can find a list of all the constants and their values in the Outlook help file, or look up a particular constant in the Outlook Object Browser. The Outlook Object Browser is discussed later in this chapter.

Error Handling

The VBScript engine provides very basic error-handling functionality. For example, if a run-time error occurs, VBScript will display a message and stop execution. You'll probably want to override the default error handler since it returns messages that are not properly formatted for the user—they're really more for the developer. To override the default error handler, you need to follow two steps in your VBScript code.

First you need to tell VBScript how to proceed when an error occurs. By default, all run-time errors in VBScript are considered fatal errors. This means that an error message will appear and the script will stop running. To override the built-in error message in your VBScript code, add the On Error Resume Next statement in each procedure in which you want custom error handling. This statement informs VBScript to continue executing beginning at the line following the code that caused the error.

Second, you need to figure out what the error was and what the application should do about the error. VBScript provides a global object, the Err object, which

has properties you can check to get information about the error. These properties include a number that identifies the error and descriptive text about the error. Typically, you would check the error number property to see whether it is a number other than *0*. If the property is *0*, no error has occurred. The following code shows how to check the error number using VBScript:

```
If Err.Number <> 0 Then
'Put your error-handling code here, and exit the procedure
Exit Sub
End if
```

VBScript lets you clear the Err object once an error has been raised so that other error-handling routines in your application do not reprocess the error. VBScript also automatically clears the Err object for you after it encounters any of the following three statements: On Error Resume Next, Exit Sub, or Exit Function. The following code snippet demonstrates how to display more detailed error messages by using the properties of the Err object:

```
Dim strMessage
strMessage = "The following error " & Err.Description & _
" has occurred in the " & Err.Source & " application. " & _
"The error number was " & Err.Number & ". " & _
"Please report this error to the help desk."
MsgBox strMessage,16,"Run-time Error #" & Err.Number
```

In determining what action to take in your error-handling code, you should consider whether to exit the procedure or just continue executing the code in the procedure. If the error is nonfatal, display a dialog box to inform the user of the error and continue executing. If the error is fatal, gracefully exit the procedure and continue the application if possible.

THE SCRIPT DEBUGGER

Using the Microsoft Script Debugger, shown in Figure 5-2, you can debug the VBScript you add to an Outlook application. You also use this debugger for Microsoft Active Server Pages, so learn to use it—it will assist you as you develop all your Exchange Server applications.

To use the Script Debugger, you must first install it. The Script Debugger is part of the Development Tools component in Outlook. You can add this component by running Add/Remove Programs for Outlook 2000.

The Script Debugger can be launched in two ways. The first way is by inserting a Stop statement in your VBScript. When you run your form and the Stop statement is executed, the Script Debugger should automatically be launched. The second way is to launch it manually from a form that contains VBScript and is in run mode by choosing Tools/Forms and then Script Debugger.

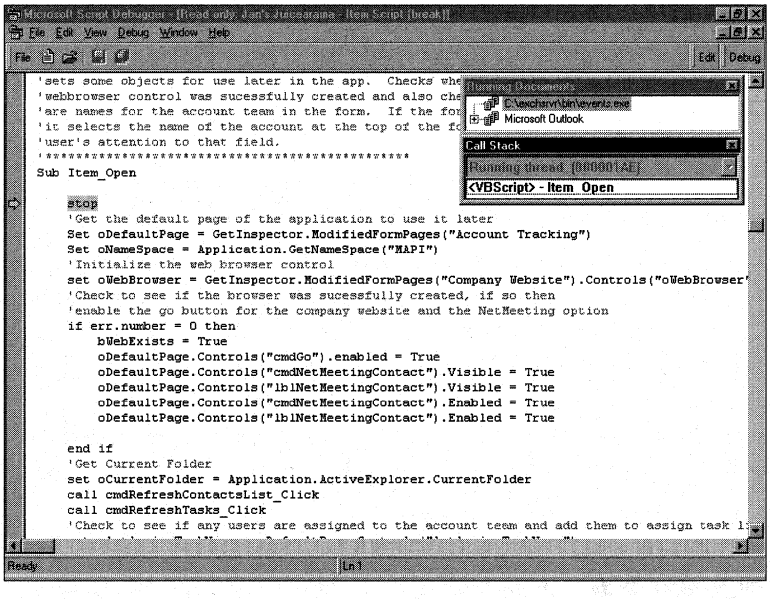


Figure 5-2. *The Microsoft Script Debugger.*

The following list describes some of the common tasks you will want to perform with the Script Debugger:

- *Set a breakpoint.* To set a breakpoint, insert a Stop statement in your VBScript code. When this Stop statement is encountered, the Script Debugger should be automatically launched.
- *Control script execution.* Once in the Script Debugger, you can control the execution of the script. You can either step through a procedure line by line or step over procedures. You can also cause the script to continue executing normally after it has been stopped.
- *View and change values at run time.* Through the Command window of the Script Debugger, you can view and change the values for specific variables in your application. These changes are preserved only in the context of the current script. For example, you can print out to the Command window the value of a variable, such as the Subject property of the current item, by using the command ? Item.Subject. You can also change the value by typing in an assignment statement such as Item.Subject "My Debugged Script". You can execute methods inside of your script by calling them directly.
- *Trace the call stack.* The Script Debugger includes all the currently running procedures in your script. This allows you to see how a particular

procedure was called, which is especially helpful when that procedure is a part of a nested procedure. (Note that you can view the source of your script in the Script Debugger, but it's read-only. To make changes to your script, you need to go back to the Outlook Script Editor.)

WORKING WITH OUTLOOK OBJECTS

There are two distinct object libraries you should know about when creating Outlook applications: the Microsoft Forms 2.0 object library and the Microsoft Outlook object library. The Microsoft Forms 2.0 object library contains all the built-in visual interface controls for Outlook forms (discussed in Chapter 4) including text boxes, list boxes, and multipage controls. This object library is contained in the file called Fm20.dll. If you've developed Office applications, you'll be familiar with these controls—they are the same controls that you use to create forms in the other Office applications.

The second object library you should know about is the Microsoft Outlook object library. The object library for Outlook 2000 is contained in the file Msoutl9.olb. This file contains the objects that you can use to develop custom Outlook solutions. It is not necessary to have a reference to either of these two libraries in your Outlook forms. Outlook automatically references the libraries for you, so you can start taking advantage of their powerful features.

Getting Help with Outlook Objects

You should take a look at some of the documentation provided by Outlook to help you. For Outlook 2000, a help file is useful when creating applications: Vbaoutl09.chm. This help file is normally stored in Program Files\Microsoft Office\Office. (You can also find it on the companion CD.) The help file includes information about the Outlook object library and the Forms object library and is shown in Figure 5-3.

These help files include not only detailed information about the Outlook objects and controls in the libraries but also code samples that will help you get started with the objects. This documentation is a great reference tool to use in conjunction with this book: the documentation outlines the objects and their properties, methods, and events, and this book shows you how to implement those objects to create complete solutions.

To access the Outlook object help file in Outlook, open a form in design mode. From the Form menu, select View Code. From the Help menu, select Microsoft Outlook Object Library Help.

The Outlook object library contains many objects. Due to the sheer volume and functionality of these objects, this chapter will not cover them in detail. Instead, I have

included a supplement on the companion CD, named "Programming Outlook and Exchange Supplement," that discusses many of the Outlook objects and collections and includes some sample code. The following is a list of the Outlook objects and collections discussed in this supplement:

- Application object
- Explorer object
- Inspector object
- Pages collection
- Page object
- Controls collection
- Control object
- NameSpace object
- AddressLists collection
- AddressList object
- AddressEntries collection
- AddressEntry object
- Folders collection
- MAPIFolder object
- Items collection
- PostItem object
- MailItem object
- ContactItem object
- AppointmentItem object
- MeetingItem object
- TaskItem object
- Recipients collection
- Recipient object
- UserProperties collection
- UserProperty object
- FormDescription object

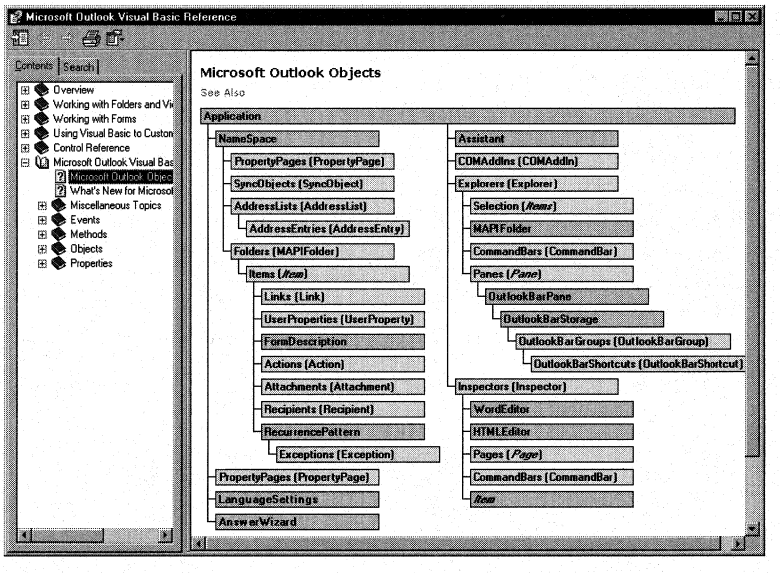


Figure 5-3. *The Outlook help file.*

The Outlook Object Browser

To make it easy for you to find objects in the Outlook object model, Outlook provides an object browser, which is shown in Figure 5-4. The Outlook Object Browser lists the available Outlook objects with their methods and properties. You can quickly add these objects to your code by clicking the Insert button. Clicking Object Help opens the Outlook object library help file.

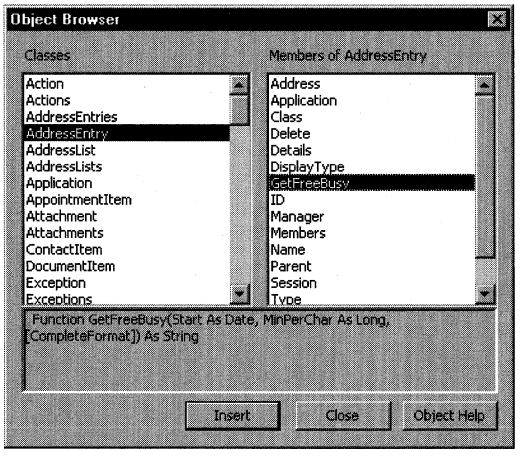


Figure 5-4. *The Outlook Object Browser is accessible from the Script Editor. The object browser allows you to insert objects into your code as well as get help on all the objects.*

While Outlook does provide an object browser for the Outlook object library, it does not provide an object browser for the Microsoft Forms 2.0 object library. To browse the objects contained in this library, you need to use the VBA object browser from another product or from within Outlook. Since VBA is integrated into the Office products, you can use the VBA Object Browser. And because the Microsoft Forms 2.0 object library is shared across the Office products, you do not need to add a reference to this library in the object browser: the library is added by default to the VBA Object Browser. The following steps explain how to view the Microsoft Forms 2.0 object library from within Outlook. The same steps could be used in Microsoft Word 2000 or Microsoft PowerPoint 2000.

1. From the Tools menu, select Macro and then Visual Basic Editor.
2. From the View menu, select Object Browser. The VBA Object Browser is displayed.
3. To view the Microsoft Forms 2.0 object library, select MSForms from the Project/Library drop-down list, as shown in Figure 5-5.

   NOTE If you don't see MSForms in your drop-down list, add it to your references, as explained in the next step.

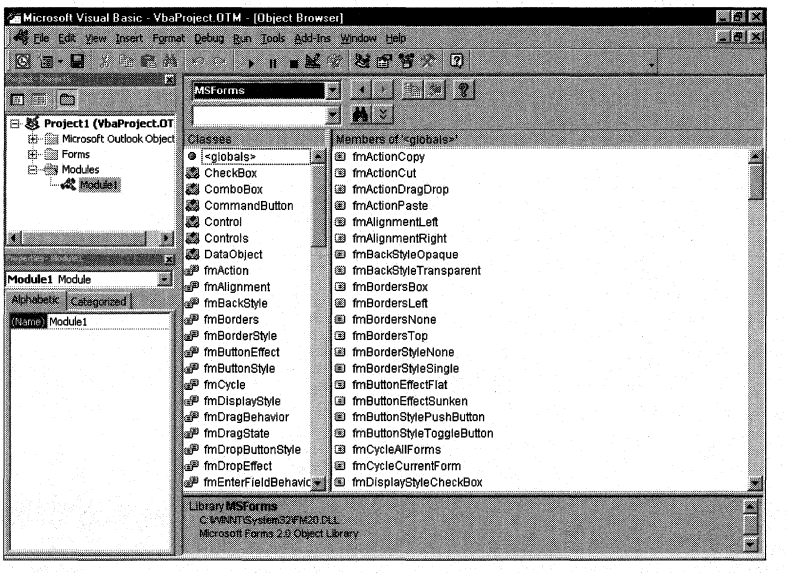


Figure 5-5. *The VBA Object Browser being used from Excel 2000 to view the Microsoft Forms 2.0 object library.*

4. To view other object libraries, such as the Outlook object library, you need to add a reference to the library: from the Visual Basic Editor Tools menu, select References. Check the library that you want to add as a reference. For Outlook, check Microsoft Outlook 9.0 Object Library and click OK.

5. From the Project/Library drop-down list, select Outlook to view only the Outlook object library.

The Outlook Object Hierarchy

Let's look briefly at the Outlook object hierarchy so that you gain a basic understanding of how these objects can be used. In Chapter 6, we will bring a lot of the concepts we have learned together by looking at an Account Tracking application.

The Outlook object library is a hierarchy of unique objects, as shown in Figure 5-6. This hierarchy makes it easier to understand the object library. To create or edit certain instances of the objects, you need to traverse the hierarchy.

In the Outlook object library, user interface objects are separated from data objects. This allows you to change the controls on your forms without having to modify the underlying data and gives you great flexibility in controlling the user interface presented to your users.

The Outlook object library is also built on the notion of items and collections. An item is a distinct object, such as the ContactItem object, the TaskItem object, and the PostItem object. A collection is a group of related objects. For example, the Items collection is a container for Outlook items.

In Outlook, you can access the specific items in a collection in two ways. The first way is to use the Items collection with an index that references the specific item you want to access. The following code snippet illustrates this approach. It shows how you can use the Items collection to retrieve the items in your Inbox and display the message text:

```
Sub CommandButton1_Click
'Open the Inbox using the GetDefaultFolder method
Set oInbox = Application.GetNameSpace("MAPI").GetDefaultFolder(6)
Set oItems = oInbox.Items
'Notice how you can get the count
Msgbox "Number of items in your Inbox: " & oItems.Count
For counter = 1 to 3
Msgbox oItems(counter).Subject
Next
End Sub
```

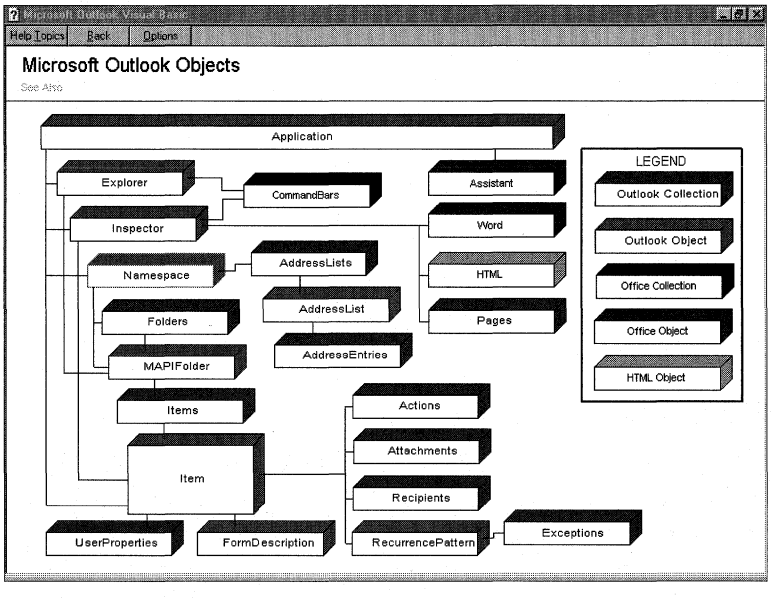


Figure 5-6. *The Outlook object hierarchy.*

The second way you can access specific items in a collection is by using a named argument. For example, instead of calling the Items collection with an index, you can just pass in a name that corresponds to the default property for the item and uniquely identifies the item. For example, the Subject property is the default property for the MailItem object. Therefore, you can pass in a unique subject name to identify a specific message. The following code illustrates this:

```
Sub CommandButton1_Click
'Open the Inbox using the GetDefaultFolder method
Set oInbox = Application.GetNameSpace("MAPI").GetDefaultFolder(6)
'Point to the Inbox Items collection
Set oItems = oInbox.Items
'Display the body of the message that has the Subject
'My Unique Subject
Msgbox oItems("My Unique Subject").Body
End Sub
```

The topmost object in the Outlook object library is the Application object. All other Outlook objects are created either directly from the Application object or from a child object of the Application object. The Application object provides four key functions that you can take advantage of in your applications. First, the Application object provides one-step creation of any of the built-in Outlook types, such as the Contact, Task, or Message item. Second, the Application object allows you to obtain the currently active user interface elements, such as the current Outlook window the user is displaying and the current Outlook window that contains the folder the user is viewing. Third, the Application object provides an entry point for you to access the data that is stored inside of Exchange Server by using the Outlook objects. Fourth, the Application object, since it is the topmost object, enables you to create or reference the other Outlook objects in your application. When you write code, you'll find that you will use the Outlook Application object extensively.

The code you write in Outlook is automatically passed both a precreated Application object and an Item object. The Application object corresponds to the currently active instance of Outlook, and the Item object corresponds to the currently active instance of the form. This frees you from having to write code that creates both of these objects in your application. In fact, it is strongly recommended that you do not attempt to use the *CreateObject* function in Outlook to create another instance of Outlook. Instead, you should use the Application object that is available in your VBScript code.

Since the Application and Item objects are already created for your application, you can use their methods and properties immediately. For example, you can change the subject or body of the currently displayed item without having to search the current folder for the item, as shown in the following code:

```
Sub CommandButton1_Click
Item.Subject = "This is using the built-in Item object"
Item.Body = "This can make writing code easier"
Importance = 2   'High
'Notice how you do not have to include the Item keyword.
'However, it's a good practice to include the explicit Item
'keyword to make your code more readable.
End Sub
```

> **NOTE** This implicit Item object and its methods and properties correspond to the type of Outlook item the form is based on. For example, the properties and methods for an implicit Item in a Contact form are different from those for a Message form. You need to be aware of this when developing your applications.

As you saw with the Item object, the Application object is also automatically available to you in your VBScript code. The following code shows you how to use some of the properties of the built-in Application object. Note that unlike the Item object, the built-in Application object requires you to explicitly place the word *Application* before any of its methods or properties.

```
Sub CommandButton1_Click
Msgbox Application.Version
Application.Quit 'Quits Outlook
End Sub
```

OUTLOOK EVENTS

Outlook supports 11 events that are called when users of your Outlook applications try to use some of the built-in capabilities of Outlook. You can write code to handle these events as well as to disable these events. Let's look at how to add some event-handling code to your application and review the sequence of the events.

Writing Event Handlers

The Outlook Script Editor makes it easy to add event handlers to your VBScript code. In the Script Editor, select Event Handler from the Script menu. Select the event you want to write a handler for, and click Add. Outlook will automatically add a skeleton function with the correct parameters for you. All you need to do is fill in your specific code to handle the event. The Script Editor does not, however, provide the option to add skeleton code to handle the events your controls support. In this case, you will need to write your event handlers from scratch. In Outlook, the event names are preceded with the word *Item*. For example, the open event handler is named Item_Open.

Disabling Events

You can prevent events and their default behavior from occurring by setting the function value, or the name of the event, to *False*. You can then place your own code inside the event handler to replace the standard behavior. For example, if you write a custom Contact form that synchronizes the information with a database, you can disable the Write event when the database is not available. This will prevent Outlook from saving the item and also prevent the two data sources, Outlook and the database, from becoming out of sync. The following code sample shows an example of this:

```
Function Item_Write()
'Call the CheckDatabase function
boolIsOnline = CheckDatabase()
if boolIsOnline = False then
'No database. Do not save the item.
Item_Write = False
msgbox "The database is offline"
end if
End Function

Function CheckDatabase
'You would put your database connection code here
CheckDatabase = False
End Function
```

Sequence of Events

Here is a list of the built-in Outlook events, which are discussed in more detail in the "Programming Outlook and Exchange Supplement" on the companion CD:

- Item_Close
- Item_CustomAction
- Item_CustomPropertyChange
- Item_Forward
- Item_Open
- Item_PropertyChange
- Item_Read
- Item_Reply
- Item_ReplyAll
- Item_Send
- Item_Write

Outlook also includes another event, named Click, that can be used with your custom controls. The Click event is the only control event supported in Outlook. The following list describes the sequence of some of these events when you perform common Outlook tasks:

- *Creating a new item using a form or in-cell editing.* When a user attempts to create a new item in Outlook either by clicking the new mail icon to open a form or by starting to type information into a new item row in a view with in-cell editing enabled, Outlook will fire the Item_Open event.
- *Sending an item.* When a user attempts to send an item in Outlook, the Item_Send event is fired first followed by the Item_Write event and then the Item_Close event. If you disable any event in the sequence, the other events will not fire.

WORKING WITH ITEMS THAT CONTAIN VBSCRIPT

Sometimes you'll want to open your Outlook form without executing the VBScript code contained in the form. To do this, hold down the Shift key while opening the form. This method of opening forms is useful while designing your applications because it prevents VBScript functions from adding undesirable data into the form.

- *Posting an item.* When a user attempts to post an item, the Item_Write event is fired and then the Item_Close event is fired. If you disable any event in the sequence, the subsequent events will not fire.
- *Saving an item.* When a user tries to save an item, the Item_Write event is fired.
- *Opening an item.* When a user opens an item, the Item_Read event is fired followed by the Item_Open event.
- *Closing an item.* When a user attempts to close an item, the Item_Close event is fired.

OTHER COMMON TASKS IN OUTLOOK DEVELOPMENT

After you start to create applications with Outlook, you might think of development tasks you want to accomplish that are beyond the standard Outlook object library. This section highlights three common development tasks in Outlook: automating Office documents, automating Outlook from other applications, and using CDO in Outlook applications.

Automating Outlook Office Documents

In Outlook, you can use Office documents as the basis for your collaborative applications. For example, you can create an expense report application that uses the calculation features of Excel while giving users the ability to e-mail and categorize expense reports using the features of Outlook.

When using Office documents as forms, you can customize your application in two ways: through VBA in the Office document, or through VBScript in Outlook. Let's examine both of the ways you can automate an Office document application.

Using VBA with an Outlook Office Document

The following example shows you how to add VBA code to an Outlook Office document based on Excel 2000.

1. In Outlook, on the File menu, point to New and then select Office Document. In the New Office Document dialog box, select Microsoft Excel Worksheet and click OK.
2. In the displayed Microsoft Outlook dialog box, select either the Post or the Send option and click OK.
3. From the Tools menu on the Excel form, point to Forms and then select Design This Form.
4. From the Tools menu, point to Macro and then select Visual Basic Editor.
5. Expand the project explorer, which is on the left, until you locate the ThisWorkbook object. Double-click on the object to display the code window.
6. From the Object drop-down list, select Workbook. Excel should automatically place a *Workbook_Open* subroutine in the code window.
7. In the procedure, use the *MsgBox* function to display some text. For example, you could add this:

```
Private Sub Workbook_Open()
Msgbox "This is from Excel"
End Sub
```

8. Close the Visual Basic Editor.
9. On the Excel form, select Run This Form from the Form menu.
10. In the displayed message box, click Yes to indicate that you trust the macros in the workbook. After you click Yes, the message box that you added earlier will be displayed.

Using VBScript with an Outlook Office Document

There might be times when you would rather automate the Office application embedded in the Outlook Office document than create VBA code in the Office document. The most common example of this automation strategy is to write VBScript code that retrieves information from Outlook sources and places it in the Office document. You would use the *GetObject* method to get the currently running instance of the Office application. The next set of steps shows you how to create an Outlook Office document based on Word 2000 and automatically take the name of a contact and place it into the Word document.

1. In Outlook, from the File menu, point to New and then select Office Document. Select Microsoft Word Document, and click OK.

2. In the displayed Microsoft Outlook dialog box, select either the Post or the Send option and click OK.
3. From the Tools menu on the Word form, point to Forms and then select Design This Form.
4. Select View Code from the Form menu.
5. Type the following lines of code into the Script Editor:

```
Sub Item_Open()
set oWord = GetObject(,"Word.Application")
set oNS = Application.GetNameSpace("MAPI")
set oContact = oNS.GetDefaultFolder(10).Items.GetFirst
oWord.Selection.TypeText "Dear " & oContact.Subject
End Sub
```

6. On the Word form, from the Form menu, select Run This Form. A Word document will be displayed with the contact name already entered.

Automating Outlook from Other Applications

Since Outlook supports automation, you can access the Outlook objects from other applications. To access the Outlook objects, you typically set a reference to the Outlook object library. For example, to add a reference to the Outlook object library in Visual Basic, select References from the Tools menu. In the References dialog box, check the Microsoft Outlook 9.0 Object Library option and click OK. The next code sample shows you how to use Visual Basic to automate Outlook to return the first Calendar appointment and display it. Notice that the Outlook constant olFolderCalendar can be used, and it is not necessary to replace it with the actual value as is required in VBScript.

```
Private Sub Command1_Click()
Set oOutlook = CreateObject("Outlook.Application")
Set oNS = oOutlook.GetNameSpace("MAPI")
Set oCalendar = oNS.GetDefaultFolder(olFolderCalendar)
Set oItems = oCalendar.Items
Set oFirst = oItems.GetFirst()
oFirst.Display
End Sub
```

Using CDO in Outlook

As you have seen, Outlook provides an extensive object library with which you can develop custom applications. However, at times you'll need to extend this environment by using other object libraries. The object library most commonly used to extend Outlook applications is Collaboration Data Objects (CDO). CDO provides some functionality for dealing with data stored in Exchange Server beyond that provided by the Outlook object library.

You'll need this additional functionality in the Account Tracking application, which is discussed in Chapter 6. One requirement for the application is that it keep track of the internal team assigned to work with a particular account. Keeping track of the team includes capturing the team's directory and e-mail information so that other internal users who have questions about the account can send team members e-mail. The easiest way for users to pick account team members is to display the address book. Outlook does not support displaying the address book and returning the individual that the user selected, but CDO does. To take advantage of the CDO functionality, the Account Tracking application is extended to call the specific CDO functions, as shown here:

```
Sub FindAddress(FieldName, Caption, ButtonText)
On Error Resume Next
Set oCDOSession = application.CreateObject("MAPI.Session")
oCDOSession.Logon "","", False, False, 0
txtCaption = Caption
if not err then
Set orecip = oCDOSession.addressbook (Nothing, txtCaption, _
True, True, 1, ButtonText, "", "", 0)
end if
if not err then
item.userproperties.find(FieldName).value = orecip(1).Name
end if
   oCDOSession.logoff
   oCDOSession = Nothing
End Sub
```

As you can see from the preceding code, to take advantage of CDO, you use the *CreateObject* method of the Application object. You then pass to this method the ProgID of CDO, which is MAPI.Session. Next CDO requires that you log on to a session. Because Outlook already has an active session, the parameters passed to the CDO *Logon* method force CDO to use the already established Outlook session. From there, the application uses the CDO *AddressBook* method to bring up the address book with a specific caption and buttons, which enables the user of the application to select a person from the address book. The application then uses the Outlook object library to place the selection of the user in a custom Outlook property. The final task the application performs is to call the *Logoff* method of CDO and set the object reference to CDO to Nothing. These two steps are important because you do not want stray objects left around after your application ends.

As you have seen, you can leverage CDO in your Outlook applications. However, the integration does not stop there—you can also leverage the Outlook library in your CDO applications using a similar technique. For more information on the features of CDO and how you can use them in your Outlook application, see Chapter 12.

Chapter 6

Putting It All Together: The Account Tracking Application

This chapter discusses the Account Tracking application, which was constructed based on the techniques described in Chapters 1 through 5. The application allows you to track contacts, team members, tasks, and revenue for different company accounts. The application is included on the book's companion CD.

Overview of the Account Tracking Application

This Account Tracking application uses a customized Post form, a customized Contact form, the standard Task form, a customized Outlook Today page, a customized folder, and an optional database. To create a new account, the user either installs the

Account Tracking form in a forms library or uses the Create New Account hyperlink in the customized Outlook Today page. After the account is created, the user can create new contacts or new tasks for the account. The user can even track internal team members who service the account. The Account Tracking application also connects to a Microsoft Access database, which enables users to retrieve revenue information. The Account Tracking application does not write to the database. By studying this application, you should learn how to:

- Customize a Post form and a Contact form in Microsoft Outlook.
- Use ActiveX controls on an Outlook form.
- Programmatically restrict items by using the Outlook object library.
- Automate other programs from an Outlook form.
- Connect a database to an Outlook form, and use a database as a data source for Outlook fields.
- Customize the Outlook Today page.

After you review the features of the application, you'll learn how to set it up and examine the code that drives it.

The Account Tracking Folder

The primary way the user interacts with this application is by accessing data through the Account Tracking folder, so this folder provides a number of views that enable the user to find desired information quickly. Although users can create their own data views for the folder, the application does provide some unique views by default. One view, called the Accounts view, lets a user see all accounts and related contacts and tasks in a threaded view. All associated tasks and contacts are threaded from the original form for the account, as shown in Figure 6-1.

Two other views use filters so that the user can quickly find only contacts or only tasks without looking at the other items in the folder. The Accounts-Color view uses conditional formatting features to color-code accounts, contacts, and tasks.

Recall that Outlook supports multiple types of views, such as timeline and card views, in a single folder. Our Account Tracking folder offers these special types of views. Figure 6-2 shows the contacts for different accounts in a view named Account Contacts. This view allows users to print out the contacts list so that it's portable—they can take it with them in paper-based planners.

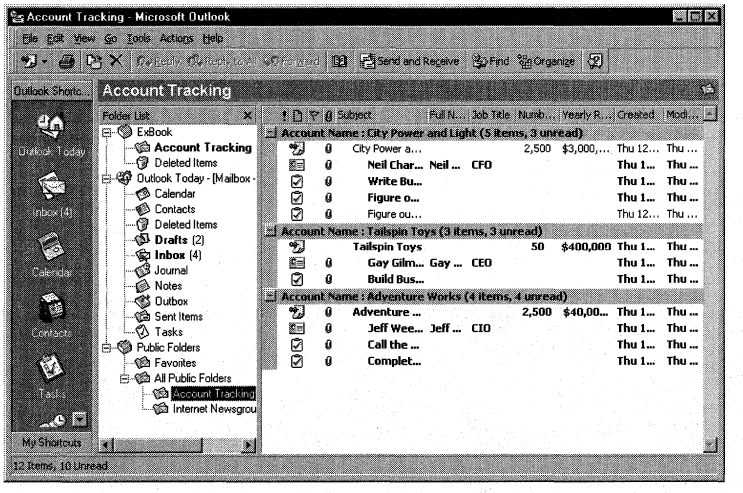


Figure 6-1. *The Accounts view in the Account Tracking folder.*

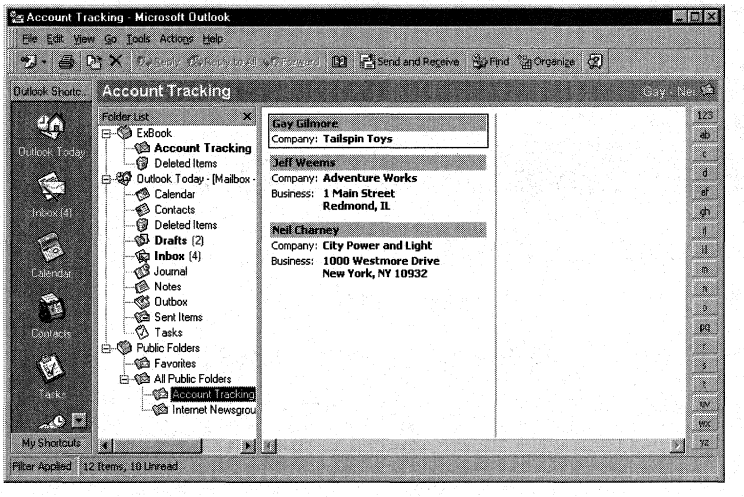


Figure 6-2. *The Account Contacts view of the Account Tracking folder, which shows only the contacts for the different accounts.*

The Account Tracking Form

Users employ the Account Tracking form to create a new item, such as a new account, a contact, a task, a letter, or a NetMeeting. The form is accessed by double-clicking on an account or by selecting New Account Info from the Actions menu. Figure 6-3 shows the Account Tracking form. It consists of a customized Post form with multiple tabs and ActiveX controls.

Every action, such as creating a task or a contact, creates an item in the folder or launches an external program such as Microsoft Word. Depending on the action invoked, a specific Outlook form is displayed. This form is then automatically posted into the folder as a reply to the original Account Tracking form, which allows the application to use the conversation topic and conversation index fields to create threaded views of the accounts, including their associated contacts and tasks. Figure 6-4 shows the Account Contact form, which is customized, and Figure 6-5 shows the Account Task form, which is a standard Outlook Task form.

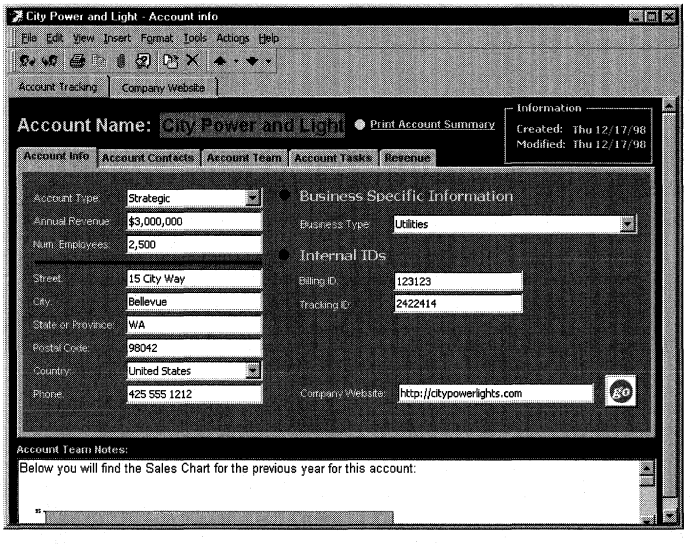


Figure 6-3. *The Account Tracking form.*

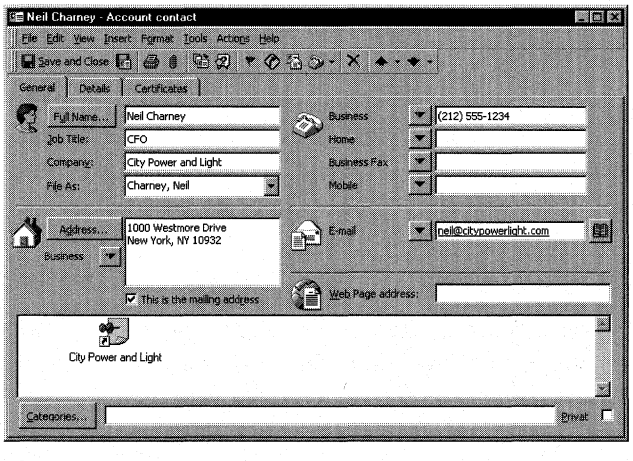


Figure 6-4. *The Account Contact form is a customized version of the Outlook Contact form.*

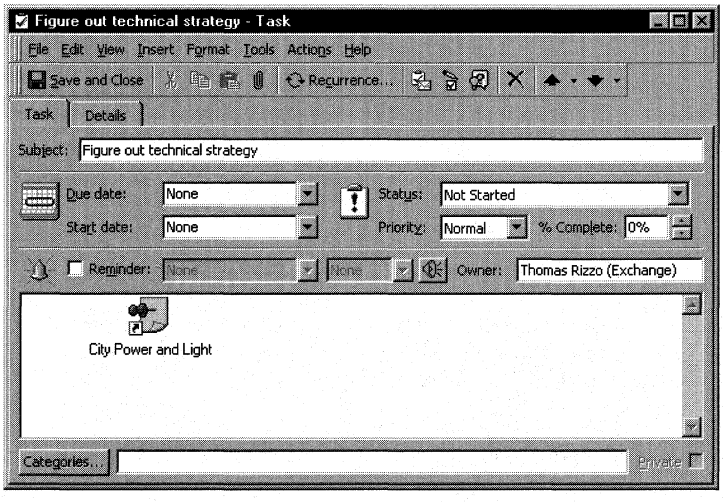


Figure 6-5. *The Account Task form uses the standard Outlook Task form.*

The Account Tracking form includes an ActiveX control—the Microsoft Internet Explorer component—which is embedded on the Company Website tab of the form. Using this control, the user can browse an account's Web site by entering the Web site address in the Company Website text box on the form's Account Info tab, as was shown in Figure 6-3, and then clicking the Go button. This control is automated by using the Microsoft Visual Basic Scripting Edition (VBScript) coder.

Optionally, the Account Tracking form can connect to a database using Data Access Objects (DAO) version 3.5. The database can be used to retrieve financial information for the Revenue tab of the form. If information is pulled from the database, the Revenue tab includes Outlook formula fields that total the actual revenues and the quota for each product, as shown in Figure 6-6.

If the user has Microsoft Excel 2000 installed, the Account Tracking application can create reports and charts in Excel. If the user clicks on the Create Sales Charts link on the Revenue tab, Outlook launches Excel and passes the numbers from the Revenue tab to the Excel chart, as shown in Figure 6-7. This functionality is accomplished by using VBScript in the form.

The final feature of the Account Tracking form is the use of Outlook events, such as Item_Open and Item_Close, to set up the environment for the user and ensure that objects needed by the application are available. For example, when a user changes the address of an account using the Account Tracking form, the Item_Close event detects this and asks whether the user wants to update all the addresses for all contacts of that account. If the user answers yes, the application finds all the account contacts and changes the address of each, as long as the original address is the same as the account address.

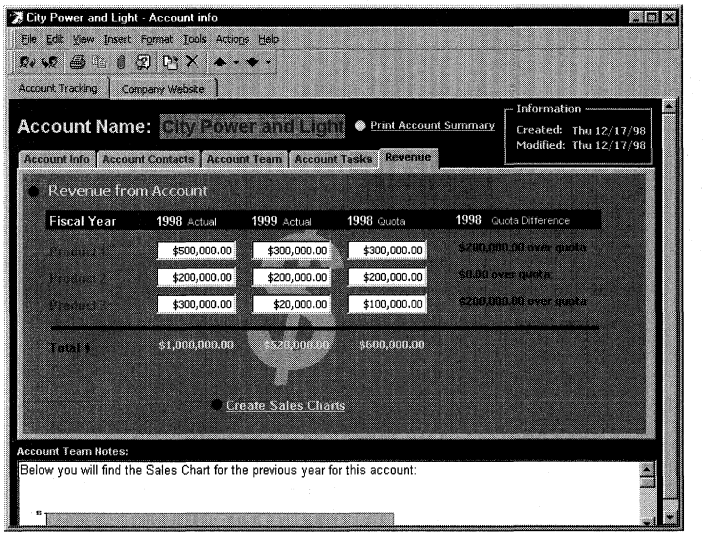


Figure 6-6. *The Revenue tab in the Account Tracking form. You can optionally use a database to populate the fields in this page. Outlook will automatically calculate the totals for each of the products using Formula fields.*

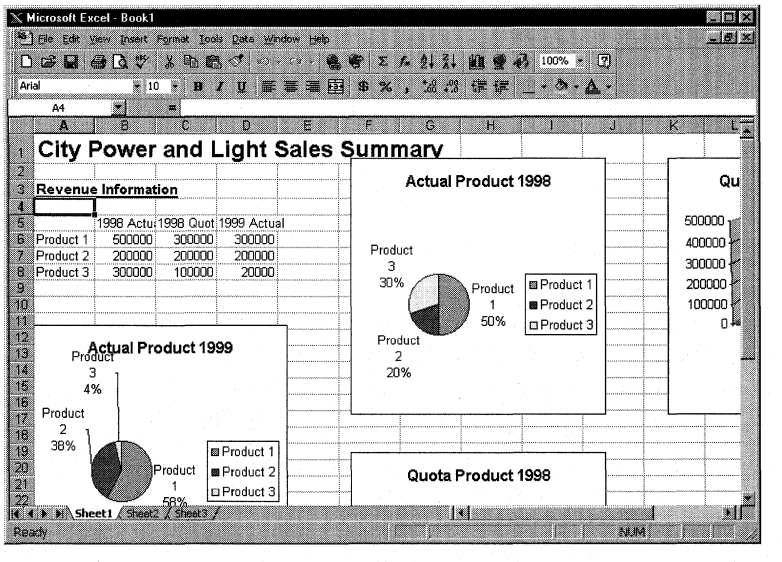


Figure 6-7. *The Excel charts are created by clicking on the Create Sales Charts link on the Revenue tab.*

SETTING UP THE APPLICATION

To install the Account Tracking application, you'll need a machine that has Outlook 2000 and at least one application from Microsoft Office 2000 installed, preferably Excel 2000. You'll also need to have a user account on a Microsoft Exchange Server. Now let's step through setting up the Account Tracking application.

> **NOTE** Files copied from CDs have their Read-only flags set. When setting up the applications included with this book, be aware that you might need to clear these Read-only flags after copying files from the companion CD.

Copying the Account Tracking Folder

First you will need to copy the Account Tracking folder from the .pst file included on the companion CD to your Public Folders in Exchange. To do this, copy the ExBook.pst file from the CD to your local hard drive. Clear the Read-only flag for this file. In Outlook 2000, choose File, point to Open, and then select Personal Folders File (.pst). In the Open Personal Folders dialog box, locate ExBook.pst on your hard drive, select it, and click OK. At this point, the file folder named ExBook should be displayed in your Outlook Folder List. Expand the ExBook file folder to display the Account Tracking folder. While pressing the Ctrl key, drag and drop the Account Tracking folder to the location in the Public Folder tree where you want the folder to appear. Pressing the Ctrl key will make a copy of the Account Tracking folder.

> **NOTE** If you do not copy the Account Tracking folder from the ExBook.pst file folder to your Public Folders, the Assign Task To functionality will not work.

Copying the Product Sales Database

Included on the companion CD is a sample Access database named Sales.mdb. The application can use this sample product sales database to retrieve product sales and quota information for an account previously entered in the database. The default location for this sales database is in the root of your C: drive, but you can change this location by modifying some of the VBScript code in the Account Tracking form. By default, the application does not use the database, but you can change this setting in the VBScript for the form. To configure the Access database, follow these steps:

1. From the companion CD, copy the file named Sales.mdb to your local hard drive and clear its Read-only flag.
2. In the Outlook Public Folders list, select the Account Tracking folder you just copied.
3. Launch the Account Tracking form by selecting New Account Info from the Actions menu.

4. Select Tools\Forms\Design This Form.
5. From the Form menu, select View Code.
6. If you do want to use the sales database, in the Global Declaration section, change the line

```
bUseDatabase = 0
```

to

```
bUseDatabase = 1
```

If you want to change the location of the database, find the *Item_Open* subroutine. Change the parameter in the line

```
InitializeDatabase "c:\sales.mdb"
```

to reflect the location of the database. For example, if the database is located on a file share, you would change the line to

```
InitializeDatabase "\\fileserver\fileshare\sales.mdb"
```

NOTE Although an Access database is being used in this sample application, you could also use a Microsoft SQL Server database.

7. Publish the form to save your changes by opening the Tools menu from the Account Tracking form, pointing to Forms, and then selecting Publish Form.

Setting Permissions on the Folder

After configuring the database, you need to set permissions on the folder. For example, you might want to give all users the ability to submit new items to the folder but give only the internal sales teams the ability to read and edit items in the folder. You also might want to create multiple folders for the different internal sales teams so that each team accesses only its own accounts, contacts, and tasks. To set up permissions for the application, right-click on the Account Tracking folder in the folder list and select Properties. Click on the Permissions tab, and then use the menus to set the permissions for the different users of the application. Consider using distribution lists to simplify setting permissions for teams of individuals.

TECHNIQUES EMPLOYED BY THE ACCOUNT TRACKING APPLICATION

The Account Tracking application demonstrates many techniques that you can employ in your Outlook applications. For example, it shows you how to connect databases to Outlook, automate applications from your Outlook application, and use the

Outlook object library to modify the controls on your form at run time. Let's review some of the interesting techniques used in the Account Tracking application.

Setting Global Variables

The first technique used by the Account Tracking application is employing global variables in VBScript to keep objects and variables in memory throughout the lifetime of the application. This technique also uses global variables to set the preferences for the application, such as whether to use a database for the product sales information. The following code shows the global variables and global initializations:

```
'*****************************************
'Global Declarations
'*****************************************
Dim oRestrictedContactItems
Dim oRestrictedTaskItems
Dim oExcel
Dim oSheet
Dim ComposeMode
Dim bWebExists
Dim oDefaultPage
Dim oWebBrowser
Dim oCurrentFolder
Dim olstAssignTaskName
Dim oNameSpace
Dim oDatabase
Dim oDatabaseEngine
Dim bUseDatabase    'Used to tell the application to use an
                    'external DB
Dim txtAccountName
Dim txtOriginalStreet, txtOriginalCity, txtOriginalState
Dim txtOriginalPostalCode, txtOriginalCountry
Dim oExcelChart
bUseDatabase = 0    'Tells the application whether to use an Access
                    'database
                    'Set this to 1 to use a database, otherwise
                    'set it to 0.
ComposeMode = True  'Used to determine whether the application is in
                    'compose or read mode
bWebExists = False  'Used to determine whether the WebBrowser control
                    'was successfully created
```

Determining Compose or Read Mode: The Item_Read Event

The Item_Read event is used to determine whether the user is creating a new account or reading an existing account from the folder. Determining the mode is important because during compose mode, the code for reading the database and updating the

contact address should not be run. After the mode is determined, the code sets a global variable, *ComposeMode*, which is used throughout the application. Because the VBScript in an Outlook form runs whether an item is being composed or read, you can use the Read event and global variable approach to identify the application mode and have your application behave appropriately. The following code shows the *Item_Read* subroutine, which runs only when an item is being read:

```
'*********************************************
'Sub Item_Read
'
'This is the standard Read event for an Outlook form.
'It checks to see whether the user is in read or compose mode
'on the form.
'*********************************************
Sub Item_Read
    'Check to see if the application is in compose mode
    ComposeMode = False
End Sub
```

Initializing the Application: The Item_Open Event

The Item_Open event in the Account Tracking form is used to perform some application initialization, in this order:

1. It initializes the global variables used most commonly throughout the application, including the Page object for the default page of the form and the NameSpace object in Outlook.
2. It checks whether the WebBrowser control on the Company Website tab is successfully created. If it is, Item_Open enables a number of controls on the form by using the Controls collection.
3. It checks whether the user has filled in the internal account team. If so, it adds these users to the list box on the Account Tasks tab to make it easy for users to assign tasks to the account team members.
4. It stores the original information for the address of the company. This information is used later in the Item_Close event.
5. It initializes and opens the database using helper functions in the script.

The entire Item_Open subroutine is shown here:

```
''*********************************************
'Sub Item_Open
'
'This is the standard Outlook Open event. This subroutine
'sets some objects for use later in the app. Checks whether
'the WebBrowser control was successfully created and also checks
'to see whether there are names for the account team in the form.
'If the form is in compose mode, the subroutine selects the name of
'the account at the top of the form to draw the user's attention to
'that field.
'*********************************************
Sub Item_Open
    'Get the default page of the application to use later
    Set oDefaultPage = GetInspector.ModifiedFormPages( _
        "Account Tracking")
    Set oNameSpace = Application.GetNameSpace("MAPI")
    'Initialize the WebBrowser control
    set oWebBrowser = GetInspector.ModifiedFormPages( _
        "Company Website").Controls("oWebBrowser")
    'Check to see if the browser was successfully created; if so,
    'enable the Go button for the company Web site and the
    'NetMeeting option
    If err.number = 0 Then
        bWebExists = True
        oDefaultPage.Controls("cmdGo").enabled = True
        oDefaultPage.Controls("cmdNetMeetingContact").Visible = True
        oDefaultPage.Controls("lblNetMeetingContact").Visible = True
        oDefaultPage.Controls("cmdNetMeetingContact").Enabled = True
        oDefaultPage.Controls("lblNetMeetingContact").Enabled = True
    end if
    'Get Current Folder
    set oCurrentFolder = Application.ActiveExplorer.CurrentFolder
    call cmdRefreshContactsList_Click
    call cmdRefreshTasks_Click
    'Check to see if any users are assigned to the account team and
    'add them to assign task list
    Set olstAssignTaskName = oDefaultPage.Controls( _
        "lstAssignTaskName")
    CheckFor "txtAccountSalesRep"
    CheckFor "txtAccountSE"
    CheckFor "txtAccountConsultant"
    CheckFor "txtAccountSupportEngineer"
    CheckFor "txtAccountExecutive"
```

(continued)

```
    If not(ComposeMode) then
        txtOriginalStreet = _
            Item.UserProperties.Find("Account Street")
        txtOriginalCity = _
            Item.UserProperties.Find("Account City")
        txtOriginalState = _
            Item.UserProperties.Find("Account State")
        txtOriginalPostalCode = _
            Item.UserProperties.Find("Account Postal Code")
        txtOriginalCountry = _
            Item.UserProperties.Find("Account Country")
        oDefaultPage.Controls("lblDistrict").visible = True
        set oDistrict = oDefaultPage.Controls("lstDistrict")
        oDistrict.visible = True
    end if

    If not(ComposeMode) and bUseDatabase then
        txtAccountName = item.Subject
        'Initialize DB
        InitializeDatabase "c:\sales.mdb"
        GetDatabaseInfo "[1998 Actual]", "cur1998ActualProd1", _
            "cur1998ActualProd2","cur1998ActualProd3"
        GetDatabaseInfo "[1999 Actual]", "cur1999ActualProd1", _
            "cur1999ActualProd2","cur1999ActualProd3"
        GetDatabaseInfo "[1998 Quota]", "cur1998QuotaProd1", _
            "cur1998QuotaProd2","cur1998QuotaProd3"
    End If

    If ComposeMode Then
        oDefaultPage.txtName.SetFocus
        oDefaultPage.txtName.SelStart = 0
        oDefaultPage.txtName.SelLength = 11
    End If
End Sub
```

Connecting to the Sales Database: The *GetDatabaseInfo* Subroutine

If you have enabled a database for the sales information, the *GetDatabaseInfo* subroutine is called to retrieve the sales information from the database and place this information into Outlook fields. This subroutine uses DAO 3.5 to query the database and retrieve the sales information associated with accounts previously entered in the database. Once this information is placed in the form, Outlook formula fields determine whether the current sales of the product are exceeding the quota for the

product. Outlook then displays how much the account team needs to sell to reach its quota or how much over quota the account team is. The following code shows the *GetDatabaseInfo* subroutine:

```
'*********************************************
'Sub GetDatabaseInfo
'
'This subroutine retrieves the product revenue information
'from the database using the passed-in table name as well as
'field names and the current account name from the open item.
'You can customize this subroutine to meet your specific needs.
'*********************************************
Sub GetDatabaseInfo(TableName, FieldName1, FieldName2, FieldName3)
    strSQL = "Select Product1, Product2, Product3 FROM " & _
        TableName & " WHERE AccountName = '" & txtAccountName & "';"
    Set oRS = oDatabase.OpenRecordset(strSQL)
    If Err.Number <> 0 Then
        MsgBox Err.Description &  Err.Number & Chr(13) & _
            "OpenRecordset failed"
        Exit Sub
    End If
    oRS.MoveFirst

    Item.UserProperties.Find(FieldName1).Value = oRS.Fields(0)
    Item.UserProperties.Find(FieldName2).Value = oRS.Fields(1)
    Item.UserProperties.Find(FieldName3).Value = oRS.Fields(2)
End Sub
```

Displaying an Address Book Using CDO: The *FindAddress* Subroutine

Because Outlook does not natively support displaying an address book in its object library, the application needs to be extended with the CDO library, which will display address books and return the values selected by the user. To use CDO in the Account Tracking application, the VBScript code in the form has to create a CDO object by using the *CreateObject* method of the Outlook Application object. When the object is created, a subroutine starts a session using the CDO methods and displays an address book using the caption and button text, which are passed in as parameters. Then the subroutine stores the results selected by the user in a specific Outlook field, which is also passed in as a parameter. Finally, the subroutine logs out of the CDO session and destroys the CDO object. Figure 6-8 shows how the address book looks when you click one of the address book buttons on the Account Team tab.

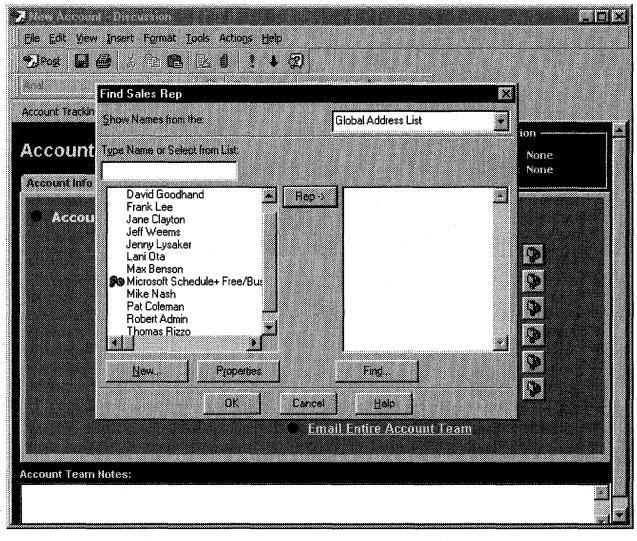


Figure 6-8. *Displaying the address book in an Outlook form by using CDO.*

The following code shows how the address book is displayed using CDO:

```
'************************************************
'Sub FindAddress
'
'This subroutine takes the Outlook field that stores
'the returned value and the caption for the dialog box as
'well as the button text for the dialog box, and then it
'displays the AddressBook dialog box by using CDO
'
'************************************************
Sub FindAddress(FieldName, Caption, ButtonText)

    On Error Resume Next
    Set oCDOSession = application.CreateObject("MAPI.Session")
    oCDOSession.Logon "", "", False, False, 0
    txtCaption = Caption
    If Not err Then
        set orecip = oCDOSession.addressbook (Nothing, txtCaption, _
            True, True, 1, ButtonText, "", "", 0)
    End If
    If Not err Then
        item.userproperties.find(FieldName).value = orecip(1).Name
    End If
    oCDOSession.logoff
    oCDOSession = Nothing
End Sub
```

Creating Account Contacts: The *cmdAddAccountContact* Subroutine

After assigning internal people to the account team, the user can add new account contacts for the company. The application has a custom action that creates a reply in the folder by using the custom Account Contact form. Because you are using an action, the command for the action, Create New Account Contact, will appear on context menus. For example, if you right-click on an account item in Outlook, Create New Account Contact will be one of the choices. Using an action also makes it easy for Outlook to automatically create a conversation thread for the account contact. Finally, using an action allows the application to attach the original item to the contact, in this case the account item, as a shortcut without any coding. The *cmdAddAccountContact* subroutine, shown in the following code snippet, executes the custom action by using the Actions collection on the account form. This code is similar to the *cmdAddTasks* subroutine, but instead of displaying an Account Contact form it displays an Account Task form for the user to fill in.

```
'*****************************************
'Sub cmdAddAccountContact_Click
'
'This subroutine creates a new contact and displays
'the form for the new contact as a modal dialog box
'*****************************************
Sub cmdAddAccountContact_Click
    Item.Save
    Set AccountContactForm = item.Actions( _
        "Create New Account Contact").Execute
    AccountContactForm.Display(True)
    call cmdRefreshContactsList_Click
End Sub
```

Refreshing the Contact List Box: The *cmdRefreshContactsList* Subroutine

When the form initially opens, or when users add or delete contacts or tasks in the folder, the ListBox control that contains these items must be refreshed and filled with the most recent information from the folder. To do this, the application calls subroutines that restrict the folder based on the item type and on the account the item belongs to. The application then programmatically fills the list box with the correct information for the account. The list box is shown in Figure 6-9.

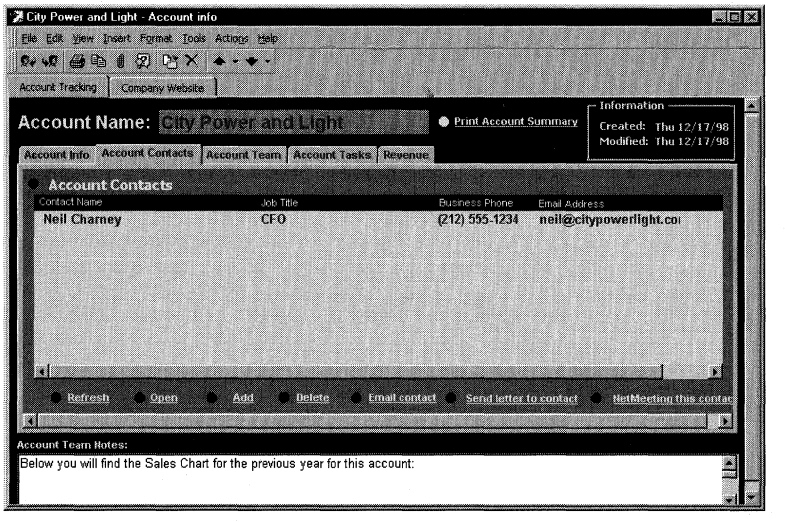


Figure 6-9. *The Account Contacts list box for the Account Tracking application. It is dynamically filled in using the contact items contained in the folder.*

The following code shows the *cmdRefreshContactsList* subroutine at work:

```
'************************************************
'Sub cmdRefreshContactsList_Click
'
'This subroutine refreshes the list box of contacts by
'applying a restriction on the folder
'
'************************************************
Sub cmdRefreshContactsList_Click
    'Initialize ListBox
    set oListBox = oDefaultPage.Controls("lstContacts")
    oListBox.Clear
    oListBox.ColumnWidths = "0;172;140;80;120"

    'Create search criteria
    RestrictString = ""
    RestrictString = "[Message Class] = " & _
        """IPM.Contact.Account contact"" and [Conversation] = """ & _
        item.ConversationTopic & """"
    Set oRestrictedContactItems = _
        oCurrentFolder.Items.Restrict(RestrictString)
    For i = 0 to oRestrictedContactItems.Count - 1
        oListBox.AddItem
        oListBox.Column(1,i) = oRestrictedContactItems(i+1).FullName
        oListBox.Column(2,i) = oRestrictedContactItems(i+1).JobTitle
```

```
        oListBox.Column(3,i) = _
            oRestrictedContactItems(i+1).BusinessTelephoneNumber
        oListBox.Column(4,i) = _
            oRestrictedContactItems(i+1).Email1Address
    Next
End Sub
```

Performing Default Contact Actions: E-Mail, Letters, and NetMeeting

Because users of this application will want to perform many actions for the account contacts they create, the Account Tracking application provides the most common actions as default controls on the Account Contacts tab. The user can click the Email Contact link to e-mail a contact. This action uses the *CreateItem* method on the Application object to create an e-mail message, and then uses the name of the selected contact to fill in the address information for the e-mail.

If Word 2000 is installed, the user can also send a letter to the contact by clicking the Send Letter To Contact link. This action takes advantage of Outlook by using the CommandBars collection on Outlook forms to trigger toolbar actions. Then, by using the *FindControl* method and the *Execute* method of the CommandBar object, the application launches the New Letter To Contact option from the Actions menu for a contact. This, in turn, launches the Microsoft Word Letter Wizard, which uses the contact information to automatically populate the address information for the letter.

Finally, the user can start a Microsoft NetMeeting with the contact by clicking the NetMeeting This Contact link. This action uses the WebBrowser control. If the WebBrowser control is available and the user launches the action, the application uses VBScript in Outlook to automate the WebBrowser control, which starts NetMeeting and connects the user to the Account contact using the NetMeeting client.

The following code shows the subroutines that enable the user to send an e-mail or a letter and to set up a NetMeeting:

```
'*********************************************
'Sub cmdEmailContact_Click
'
'This subroutine creates an e-mail message for the selected
'account contact. If there is no selected contact, it displays an error.
'*********************************************
Sub cmdEmailContact_Click
    Set oListBox = oDefaultPage.Controls("lstContacts")
    If oListBox.ListIndex = -1 Then
        MsgBox "No selected account contact. Please select one.", _
            48, "Email Account Contact"
```

(continued)

```
    Else
        Set oItem = oRestrictedContactItems(oListBox.ListIndex + 1)
        'Create an e-mail message
        Set oNewMessage = Application.CreateItem(0)
        oNewMessage.Recipients.Add oItem.Email1Address
        oNewMessage.Recipients.ResolveAll
        oNewMessage.Display
    End If
End Sub

'*********************************************
'Sub cmdSendLettertoContact_Click
'
'The following subroutine uses the commandbars
'property to automate the Contact form in Outlook
'to select the Send Letter To A Contact menu
'command. This in turn launches the Word Letter
'Wizard.
'*********************************************
Sub cmdSendLettertoContact_Click
    Set oListBox = oDefaultPage.Controls("lstContacts")
    If oListBox.ListIndex = -1 Then
        MsgBox "No selected account contact. Please select one.", _
            48, "Send letter to Account Contact"
    Else
        Set oItem = oRestrictedContactItems(oListBox.ListIndex + 1)
        oItem.Display
        oItem.GetInspector.CommandBars.FindControl(,2498).Execute
    End If
End Sub

'*********************************************
'Sub cmdNetMeetingContact_Click
'
'This subroutine checks the contact to see if the
'NetMeeting information is filled in and, if so, it
'automates the WebBrowser control to use the NetMeeting
'callto: syntax to start a NetMeeting
'*********************************************
Sub cmdNetMeetingContact_Click
    Set oListBox = oDefaultPage.Controls("lstContacts")
    If oListBox.ListIndex = -1 Then
        MsgBox "No selected account contact.  Please select one.", _
            48, "NetMeeting Account Contact"
    Else
        Set oItem = oRestrictedContactItems(oListBox.ListIndex + 1)
        If oItem.NetMeetingAlias = "" Then
            MsgBox "The NetMeeting information is not filled" & _
```

```
            " in for this contact.", 48, _
            "NetMeeting Account Contact"
        Exit Sub
    End If
    If oItem.NetMeetingServer = "" Then
        MsgBox "The NetMeeting information is not filled" & _
            " in for this contact.", 48, _
            "NetMeeting Account Contact"
        Exit Sub
    End If
    On Error Resume Next
    txtNetMeetingAddress = "callto:" & oItem.NetMeetingServer _
        & "/" & oItem.NetMeetingAlias
    oWebBrowser.Navigate txtNetMeetingAddress
    If err.number <> 0 Then
        MsgBox "NetMeeting is either not installed or not" & _
            " configured correctly.", 48, _
            "NetMeeting Account Contact"
        Exit Sub
    End If
  End If
End Sub
```

Automating Excel: The *cmdCreateSalesChart* and *cmdPrintAccountSummary* Subroutines

If the user has Excel 2000 installed, the Account Tracking application can automate Excel to create charts, as was shown earlier in Figure 6-7. One way to start the chart creation process is to click the Create Sales Chart control on the Revenue tab of the application. An even easier way to start this process is to use the context menu in the Outlook window. Depending on the item type, you can right-click on an item and select Create Account Sales Charts without opening the item. The application does this through a custom action. The application captures the Item_CustomAction event when the user selects the Create Account Sales Charts action, and it calls its own subroutine to handle the action rather than displaying a response form. The subroutine then creates sales charts by using VBScript to automate Excel. (Notice in the *Item_CustomAction* event procedure that I also tried to create an action for printing an Excel account summary. Unfortunately, this action did not work from the context menu.)

```
'*********************************************
'Function Item_CustomAction
'
'This is the standard CustomAction event for an Outlook form.
'This event is captured so that the Create Account Sales Chart
```

(continued)

```
'as well as the Print Account Summary actions can appear on the menu.
'However, these actions actually call VBScript functions. This
'is why these actions are canceled after the VBScript functions
'automate Excel to create the reports. Otherwise, a reply form
'would appear to the user.
'************************************************
Function Item_CustomAction(ByVal Action, ByVal ResponseItem)
   select case Action
      case "Create Account Sales Charts"
         cmdCreateSalesChart_Click()
         'Disable the action so that a response form does not appear
         Item_CustomAction = False
      case "Print Account Summary"
         cmdPrintAccountSummary_Click()
         Item_CustomAction = False
   end select
end Function

'************************************************
'Sub cmdCreateSalesChart_Click
'
'This subroutine responds to the Click event of the
'Create Sales Charts control. It automates Excel
'to create both a worksheet and embedded charts on that worksheet.
'You can modify this subroutine to meet your specific needs.
'************************************************
Sub cmdCreateSalesChart_Click
    Set oExcel = Item.Application.CreateObject("Excel.Application")
    oExcel.Visible = True
    oExcel.Workbooks.Add
    Set oSheet = oExcel.Workbooks(1).Worksheets("Sheet1")
    'Set the title for the worksheet
    oSheet.Activate
    set oSheetTitle = oSheet.Range("A1")

    oSheetTitle.Value = item.Subject & " Sales Summary"
    oSheetTitle.Font.Bold = -1
    oSheetTitle.Font.Size = 18
    oSheetTitle.Font.Name = "Arial"

    oExcel.Application.ActiveCell.Offset(2,0).Select
    oExcel.Application.ActiveCell.Value = "Revenue Information"
    oExcel.Application.ActiveCell.Font.Bold = -1
    oExcel.Application.ActiveCell.Font.Name= "Arial"
    oExcel.Application.ActiveCell.Font.Size = 11
    oExcel.Application.ActiveCell.Font.Underline = 2
    oExcel.Application.ActiveCell.Offset(1,0).Select
```

```
    oSheet.Range("A6").Value = "Product 1"
    oSheet.Range("A7").Value = "Product 2"
    oSheet.Range("A8").Value = "Product 3"

    oSheet.Range("B5").Value = "1998 Actual"
    oSheet.Range("B6").Value = item.userproperties( _
        "cur1998ActualProd1")
    oSheet.Range("B7").Value = item.userproperties( _
        "cur1998ActualProd2")
    oSheet.Range("B8").Value = item.userproperties( _
        "cur1998ActualProd3")

    oSheet.Range("C5").Value = "1998 Quota"
    oSheet.Range("C6").Value = item.userproperties( _
        "cur1998QuotaProd1")
    oSheet.Range("C7").Value = item.userproperties( _
        "cur1998QuotaProd2")
    oSheet.Range("C8").Value = item.userproperties( _
        "cur1998QuotaProd3")

    oSheet.Range("D5").Value = "1999 Actual"
    oSheet.Range("D6").Value = item.userproperties( _
        "cur1999ActualProd1")
    oSheet.Range("D7").Value = item.userproperties( _
        "cur1999ActualProd2")
    oSheet.Range("D8").Value = item.userproperties( _
        "cur1999ActualProd3")

    'Create charts
    set oChart = oSheet.ChartObjects.Add(250, 20, 200, 200)
    oChart.Chart.ChartWizard oSheet.Range( _
        "a6:B8"),5,,2,1,,,"Actual Product 1998"
    set oChart = oSheet.ChartObjects.Add(0, 150, 200, 200)
    oChart.Chart.ChartWizard oSheet.Range( _
        "a6:A8, D6:D8"),5,,2,1,,,"Actual Product 1999"
    set oChart = oSheet.ChartObjects.Add(250, 250, 200, 200)
    oChart.Chart.ChartWizard oSheet.Range( _
        "a6:A8, C6:C8"),5,,2,1,,,"Quota Product 1998"
    set oChart = oSheet.ChartObjects.Add(500, 20, 200, 200)
    oChart.Chart.ChartWizard oSheet.Range( _
        "a6:c8"),3,,2,1,,,"Quota vs Actual 1998"
    oSheet.ChartObjects(4).Chart.ChartType = 54
end Sub
```

When the user clicks the Print Account Summary control on the Account Tracking tab, an account summary is created in Excel. The Excel Account Summary sheet is shown in Figure 6-10.

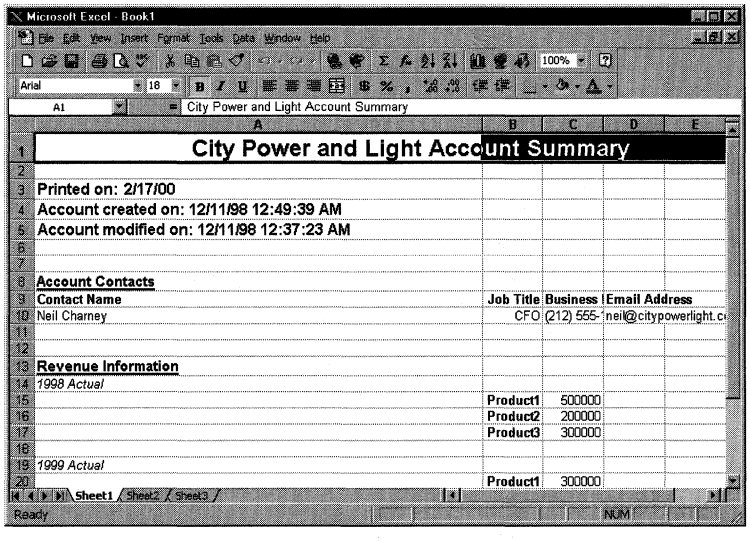


Figure 6-10. *The Excel Account Summary sheet, which is programmatically created by the Account Tracking application.*

The code to create the Account Summary is shown here:

```
'*********************************************
'Sub cmdPrintAccountSummary_Click
'
'This subroutine calls the helper subroutine to
'print the Account Summary. You can replace the
'helper subroutine without having to replace the controls
'on the form.
'*********************************************
Sub cmdPrintAccountSummary_Click()
    CreateExcelSheet
End Sub

'*********************************************
'Sub ExcelPrintProductRevenue
'
'This subroutine is a helper subroutine that prints
'the passed-in product name as well as the current
'sales numbers. You can replace this subroutine
'with your own.
'*********************************************
Sub ExcelPrintProductRevenue(ByVal txtType, txtProd1, txtProd2, _
    txtProd3,curProd1,curProd2,curProd3)
    oExcel.Application.ActiveCell.Value = txtType
    oExcel.Application.ActiveCell.Font.Italic = -1
    oExcel.Application.ActiveCell.Offset(1,1).Value = txtProd1
    oExcel.Application.ActiveCell.Offset(1,1).Font.Bold = -1
```

```
    oExcel.Application.ActiveCell(2,3).Value = curProd1
    oExcel.Application.ActiveCell.Offset(2,1).Value = txtProd2
    oExcel.Application.ActiveCell.Offset(2,1).Font.Bold = -1
    oExcel.Application.ActiveCell(3,3).Value = curProd2
    oExcel.Application.ActiveCell.Offset(3,1).Value = txtProd3
    oExcel.Application.ActiveCell.Offset(3,1).Font.Bold = -1
    oExcel.Application.ActiveCell(4,3).Value = curProd3
end Sub

'*********************************************
'Sub CreateExcelSheet
'
'This subroutine automates Excel to create an Account
'Summary report. You can replace this subroutine
'with your own.
'*********************************************
Sub CreateExcelSheet
    Set oExcel = Item.Application.CreateObject("Excel.Application")
    oExcel.Visible = True
oExcel.Workbooks.Add
    Set oSheet = oExcel.Workbooks(1).Worksheets("Sheet1")
    'Set the title for the worksheet
    oSheet.Activate
    set oSheetTitle = oSheet.Range("A1")

    oSheetTitle.Value = item.Subject & " Account Summary"
    oSheetTitle.Font.Bold = -1
    oSheetTitle.Font.Size = 18
    oSheetTitle.Font.Name = "Arial"

    'Put in the printout date
    oSheet.Range("A3").Value = "Printed on: " & Date
    oSheet.Range("A3").Font.Bold = -1
    oSheet.Range("A3").Font.Name = "Arial"
    oSheet.Range("A3").Font.Size = 12
    oSheet.Range("A3").Font.Color = RGB(0,0,255)

    'Put in the date the item was created
    oSheet.Range("A4").Value = "Account created on: " & _
        item.CreationTime
    oSheet.Range("A4").Font.Bold = -1
    oSheet.Range("A4").Font.Name = "Arial"
    oSheet.Range("A4").Font.Size = 12
    oSheet.Range("A4").Font.Color = RGB(0,0,255)

    'Put in the date the item was last modified
    oSheet.Range("A5").Value = "Account modified on: " & _
        item.LastModificationTime
```

(continued)

```
oSheet.Range("A5").Font.Bold = -1
oSheet.Range("A5").Font.Name = "Arial"
oSheet.Range("A5").Font.Size = 12
oSheet.Range("A5").Font.Color = RGB(0,0,255)
oSheet.Range("A7").Activate

'Retrieve contact information
oExcel.Application.ActiveCell.Offset(1,0).Select
oExcel.Application.ActiveCell.Value = "Account Contacts"
oExcel.Application.ActiveCell.Font.Bold = -1
oExcel.Application.ActiveCell.Font.Name= "Arial"
oExcel.Application.ActiveCell.Font.Size = 11
oExcel.Application.ActiveCell.Font.Underline = 2
oExcel.Application.ActiveCell.Offset(1,0).Select

'Refresh the contact listbox
cmdRefreshContactsList_Click
'Retrieve the data from the listbox
set oPage = GetInspector.ModifiedFormPages("Account Tracking")
set oListBox = oPage.lstContacts
If oListBox.ListCount > 0 Then
    oExcel.Application.ActiveCell.Value = "Contact Name"
    oExcel.Application.ActiveCell.Font.Bold = -1
    oExcel.Application.ActiveCell.Offset(0,1).Value = _
        "Job Title"
    oExcel.Application.ActiveCell.Offset(0,1).Font.Bold = -1
    oExcel.Application.ActiveCell.Offset(0,2).Value = _
        "Business Phone"
    oExcel.Application.ActiveCell.Offset(0,2).Font.Bold = -1
    oExcel.Application.ActiveCell.Offset(0,3).Value = _
        "Email Address"
    oExcel.Application.ActiveCell.Offset(0,3).Font.Bold = -1
    oExcel.Application.ActiveCell.Offset(1,0).Activate
    For intLB = 0 to oListBox.ListCount -1
        oExcel.Application.ActiveCell.Value = _
            oListBox.Column(1,intLB)
        oExcel.Application.ActiveCell.Offset(0,1).Value = _
            oListBox.Column(2,intLB)
        oExcel.Application.ActiveCell.Offset(0,2).Value = _
            oListBox.Column(3,intLB)
        oExcel.Application.ActiveCell.Offset(0,3).Value = _
            oListBox.Column(4,intLB)
        oExcel.Application.ActiveCell.Offset(1,0).Activate
    Next
Else
    oExcel.Application.ActiveCell.Value = _
        "No contacts for this account"
End If
```

```
    'Retrieve revenue information
    oExcel.Application.ActiveCell.Offset(2,0).Select
    oExcel.Application.ActiveCell.Value = "Revenue Information"
    oExcel.Application.ActiveCell.Font.Bold = -1
    oExcel.Application.ActiveCell.Font.Name= "Arial"
    oExcel.Application.ActiveCell.Font.Size = 11
    oExcel.Application.ActiveCell.Font.Underline = 2
    oExcel.Application.ActiveCell.Offset(1,0).Select
    'Retrieve the user properties for the revenue information
    set ouserprop = item.userproperties

    ExcelPrintProductRevenue "1998 Actual","Product1","Product2", _
        "Product3",ouserprop("cur1998ActualProd1"), _
        ouserprop("cur1998ActualProd2"), _
        ouserprop("cur1998ActualProd3")
    oExcel.Application.ActiveCell.Offset(5,0).Select _
       ExcelPrintProductRevenue "1999 Actual","Product1","Product2", _
        "Product3",ouserprop("cur1999ActualProd1"), _
        ouserprop("cur1999ActualProd2"), _
        ouserprop("cur1999ActualProd3")
    oExcel.Application.ActiveCell.Offset(5,0).Select
    ExcelPrintProductRevenue "1998 Quota","Product1","Product2", _
        "Product3",ouserprop("cur1998QuotaProd1"), _
        ouserprop("cur1998QuotaProd2"), _
        ouserprop("cur1998QuotaProd3")

    'Format the output
    oSheet.Columns("A:B").EntireColumn.AutoFit
    oSheet.Columns("B:B").HorizontalAlignment = -4152
    oSheet.Range("A1:F1").Select
    oSheet.Range("A1:F1").HorizontalAlignment=7
End Sub
```

Unloading the Application: The Item_Close Event

When the user is finished using the application, the Item_Close event for the application is invoked. In the event handler, the application checks to see whether the user has updated any account address information. If the user has updated information, the application prompts the user about whether she wants to update all the contacts for that specific account in the folder. If the user answers yes, all the accounts are updated by using the properties of the standard Outlook contact. Figure 6-11 shows the message box that is displayed when the user changes the address in the Account Tracking form.

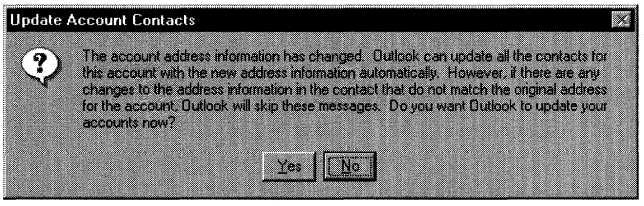


Figure 6-11. *If the user wants to change the default address for each contact, this message box prompts the user about whether to change the addresses of all associated contacts for the account.*

In the code that follows, notice how the *Save* method of the account contact is called only once after all the properties are changed. Outlook automatically parses the individual address properties such as *BusinessAddressStreet*, *BusinessAddressCity*, and *BusinessAddressPostalCode* to create the overall *BusinessAddress* property. If the code saved the item after making a change in each property, Outlook would automatically overwrite the previous changes when it parsed the individual properties to create the *BusinessAddress* property. Instead a temporary variable, *boolSaveItem*, is used to notify the code at the end of the If statements about whether the contact items that are being modified need to be saved or not. The *Item_Close* event handler also contains the code to destroy any database objects used in the application so that they are not left in memory after the application closes. The following code shows the *Item_Close* event procedure:

```
'**********************************************
'Function Item_Close
'
'This function fires on the standard Outlook close
'event and prompts the user about whether to update
'all contacts for the company if the user changed the
'master address for the company. This routine will
'update only the contacts that have the same text in
'the address fields as the original since users can
'change the address fields to reflect different
'locations or addresses for customers. This function
'also cleans up any open database objects that are left.
'**********************************************
Function Item_Close()
   boolSomethingDirty = 0        'False
   If not(ComposeMode) then
      'Divided into multiple ifs to pinpoint changed property on
      'exit for faster performance when updating
      If oDefaultPage.Controls("txtStreet").Value <> _
      txtOriginalStreet then
         boolStreetIsDirty = 1
         boolSomethingDirty = 1
```

```
End if
if oDefaultPage.Controls("txtCity").Value <> _
   txtOriginalCity then
   boolCityIsDirty = 1
   boolSomethingDirty = 1
End if
if oDefaultPage.Controls("txtState").Value <> _
   txtOriginalState then
   boolStateIsDirty = 1
   boolSomethingDirty = 1
End if
if oDefaultPage.Controls("txtPostalCode").Value <> _
   txtOriginalPostalCode then
      boolPostalCodeIsDirty = 1
      boolSomethingDirty = 1
End if
if oDefaultPage.Controls("lstCountry").Value <> _
   txtOriginalCountry then
   boolCountryIsDirty=1
   boolSomethingDirty = 1
End if
If boolSomethingDirty then
   'Make sure the user wants to update all the
   'contact addresses
   intResponse = msgbox("The account address " & _
      "information has changed. Outlook can update " & _
      "all the contacts for this account with " & _
      "the new address information automatically. " & _
      "However, if there are any changes to the " & _
      "address information in the contact that do " & _
      "not match the original address for the " & _
      "account, Outlook will skip these messages. Do " & _
      "you want Outlook to update your accounts now?", _
      292, "Update Account Contacts")
   if intResponse = 6 then      'Yes
      for counter = 1 to oRestrictedContactItems.Count
         boolSaveItem = 0
         set oItem = _
            oRestrictedContactItems.Item(counter)
         if boolStreetIsDirty then
            if oItem.BusinessAddressStreet = _
               txtOriginalStreet then
               oItem.BusinessAddressStreet = _
               oDefaultPage.Controls("txtStreet").Value
               boolSaveItem = 1
            end if
         end if
```

(continued)

```
                if boolCityIsDirty then
                   if oItem.BusinessAddressCity = _
                      txtOriginalCity then
                      oItem.BusinessAddressCity = _
                      oDefaultPage.Controls("txtCity").Value
                      boolSaveItem = 1
                   end if
                end if
                if boolStateIsDirty then
                   if oItem.BusinessAddressState = _
                      txtOriginalState then
                      oItem.BusinessAddressState = _
                      oDefaultPage.Controls("txtState").Value
                         boolSaveItem = 1
                   end if
                end if
                if boolPostalCodeIsDirty then
                   if oItem.BusinessAddressPostalCode = _
                      txtOriginalPostalCode then
                      oItem.BusinessAddressPostalCode = _
                         oDefaultPage.Controls( _
                         "txtPostalCode").Value
                         boolSaveItem = 1
                   end if
                end if
                if boolCountryIsDirty then
                   if oItem.BusinessAddressCountry = _
                      txtOriginalCountry then
                      oItem.BusinessAddressCountry = _
                      oDefaultPage.Controls("lstCountry").Value
                      boolSaveItem = 1
                   end if
                end if
                If boolSaveItem then
                   'Make sure address information is only
                   'parsed once by Outlook
                   oItem.Save
                end if
             next
          end if
       end if
    end if
    'Close the database if enabled
    if ComposeMode=False and bUseDatabase then
       oDatabase.Close
       set oDatabaseEngine = Nothing
    end if
End Function
```

OUTLOOK TODAY AND THE ACCOUNT TRACKING APPLICATION

Users have a secondary way to interact with the Account Tracking application—through a customized Outlook Today page. Outlook Today is discussed more in Chapter 7, but here is an overview. Microsoft Outlook 2000 includes a feature that takes advantage of the HTML support in Outlook. This feature, Outlook Today, allows users to view all their information in one HTML window rather than as separate modules. You can customize Outlook Today's HTML code so that you can provide your Outlook or intranet information in a single window view as well.

When customized for the Account Tracking application, Outlook Today allows users to quickly create new accounts; find account contacts; and open the Account Tracking folder to see how many accounts, tasks, and contacts are contained there. The customized Outlook Today page is shown in Figure 6-12.

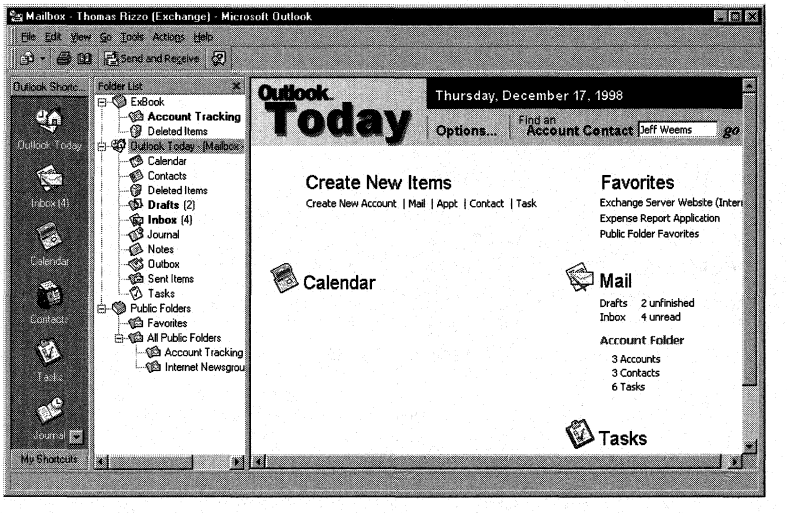


Figure 6-12. *The customized Outlook Today page in the Account Tracking application. Notice how users can quickly create or find information pertaining to their accounts right from this page.*

Viewing the Customized Outlook Today Page

The customized Outlook Today page for the Account Tracking application shows you how you can use your VBScript and Outlook object library skills in another medium, HTML. The code in the customized Outlook Today page uses the Outlook object library to count the number of items in the account folder as well as search the folder for specific account contacts. The code on the following page is taken from the Account.htm file, which is the Outlook Today page customized for the Account Tracking application.

```
<script language="VBScript">

'*****************************************************
'In-line code
'
'These lines of code are run when the browser reaches
'them while parsing the document. They set up the global
'variables that are needed throughout the application.
'*****************************************************
Set oApplication = window.external.OutlookApplication
Set oNS = oApplication.GetNameSpace("MAPI")

'Change this to your location for the Account Tracking Folder
set oAccountFolder = oNS.Folders("ExBook").Folders("Account Tracking")

'*****************************************************
'Sub FindAccountContact
'
'This subroutine takes the name of the contact that the
'user types into the Outlook Today page and searches the
'contact folder for the contact. If the contact is
'found, it displays the contact. If the contact is not
'found, it displays a message box.
'*****************************************************
Sub FindAccountContact(ContactName)
   if ContactName <> "" then
      boolFound = 0
      RestrictString = ""
      RestrictString = "[Message Class] = " _
         """IPM.Contact.Account contact"""
      Set oContacts = _
         oAccountFolder.Items.Restrict(RestrictString)
      for i = 1 to oContacts.Count
         set oContact = oContacts.Item(i)
         if oContact.FullName = ContactName then
            oContact.Display()
            boolFound = 1
            exit for
         end if
      next
      if boolFound = 0 then
         msgbox "No contact by that name was found",, _
            "Find Account Contact"
      end if
   end if
End Sub
```

```
'*************************************************
'Sub CreateAccount
'
'This subroutine creates a new Account info form and
'displays it for the user to fill in
'*************************************************
Sub CreateAccount()
   set oAccount = oAccountFolder.Items.Add("IPM.Post.Account info")
   oAccount.Display()
End Sub

'*************************************************
'Sub DisplayAccountFolder
'
'This subroutine finds and displays the Account Tracking
'folder in a separate Outlook window. The reason for this
'is to create a new Explorer object separate from the
'current Explorer object in Outlook Today.
'*************************************************
Sub DisplayAccountFolder()
   'Change this location to your folder location
   set oFolder = oNS.Folders("ExBook").Folders("Account Tracking")
   oFolder.Display()
End Sub

'*************************************************
'Sub GetAccountFolderCounts
'
'This subroutine calculates how many accounts, contacts,
'and tasks are in the Account Tracking folder and
'displays this information
'*************************************************
Sub GetAccountFolderCounts()
   RestrictString = ""
   RestrictString = "[Message Class] = ""IPM.Post.Account info"""
   Set oAccounts = oAccountFolder.Items.Restrict(RestrictString)
   oAcctCount = oAccounts.Count
   AccountCount.innerHTML = oAcctCount & " Accounts"

   RestrictString = ""
   RestrictString = "[Message Class] = " _
      """IPM.Contact.Account contact"""
   Set oContacts = oAccountFolder.Items.Restrict(RestrictString)
   oContactCount = oContacts.Count
   ContactCount.innerHTML = oContactCount & " Contacts"
```

(continued)

```
    RestrictString = ""
    RestrictString = "[Message Class] = ""IPM.Task"""
    Set oTasks = oAccountFolder.Items.Restrict(RestrictString)
    oTasksCount = oTasks.Count
    TaskCount.innerHTML = oTasksCount & " Tasks"
end Sub
</script>
```

Setting Up the Customized Outlook Today Page

You need to modify the Outlook Today page so that it knows the location of the Account Tracking folder and also modify your Registry to point Outlook at the customized Outlook Today page. If you want to deploy the customized Outlook Today page to users in your organization, consider writing a simple Microsoft Visual Basic program that modifies their Registries and points them to a Web server containing the customized Outlook Today page. If you have not deployed Outlook to your organization yet, you can visit the site *http://www.microsoft.com/office/features/ofc2000tour* to see the demo showing how to set the default location for the Outlook Today page.

The following steps show you how to modify the Outlook Today page and your Registry for the Account Tracking application:

1. On the companion CD, open the file named Account.htm in a text editor such as Notepad.
2. Change the line

   ```
   set oAccountFolder = oNS.Folders("ExBook").Folders( _
      "Account Tracking")
   ```

   so that it reflects the location of the Account Tracking folder. For example, if you copied the Account Tracking folder into the main tree of your Public Folder hierarchy, the code would look like this:

   ```
   set oAccountFolder = oNS.Folders("Public Folders").Folders( _
      "All Public Folders").Folders("Account Tracking")
   ```

3. Modify the following line so that it reflects the location where you copied the Account Tracking folder:

   ```
   set oFolder = oNS.Folders("ExBook").Folders(" _
      Account Tracking")
   ```

4. Save the file to your hard drive to keep your changes.

NOTE The general instructions for modifying the Registry are given here, but for detailed instructions, follow steps described on page 190 in Chapter 7, in the section titled "Modifying the Registry." When the procedure asks you for the URL, specify the location where you saved the Account.htm file.

To modify the Registry for the customized Outlook Today page, add the following key to the Registry:

```
HKEY_CURRENT_USER\Software\Microsoft\Office\9.0\Outlook\Today
```

Add a new string value to the Today key named *Url*. For its value data, type the path to the Account.htm. For example, file://C:\Account.htm. When finished, click the Outlook Today icon in Outlook to display your customized Outlook Today page.

Chapter 7

Outlook and the Web

This chapter looks at how Microsoft Outlook ties into the World Wide Web and is broken into five main sections: Outlook Today, Active Server Pages (ASP), Outlook Web Access (OWA), Windows 2000 and IIS 5.0, and the Outlook HTML Form Converter. We start by examining Outlook Today, which is an HTML feature introduced in Outlook 2000. Then we'll cover the basics of ASP—the foundation for Outlook and the Web. We'll move on to an overview of installing and configuring OWA. OWA technology is based on Active Server Pages and Microsoft Collaboration Data Objects (CDO), and it's used to access information on Microsoft Exchange Server and Microsoft Internet Information Services (IIS). We'll finish with a discussion of the Outlook HTML Form Converter. The Form Converter is a tool to help you convert your Outlook forms into a Web-based format that integrates OWA.

OUTLOOK TODAY

As you learned at the end of Chapter 6, Outlook 98 and Outlook 2000 include a feature that takes advantage of the HTML support in Outlook—Outlook Today. Outlook Today allows users to view information in an HTML window, as shown in Figure 7-1, rather than as separate modules. You can also customize Outlook Today's HTML code. For example, a customized Outlook Today page might include hyperlinks to Public Folder favorites, intranet sites, or Internet sites. You can even link to other applications, such as a Web-based Microsoft SQL Server application. You can also create

multiple Outlook Today pages for the different types of users of your application. Let's take a look at how the Outlook Today page functions and how you modify the Outlook Today page for your specific application needs.

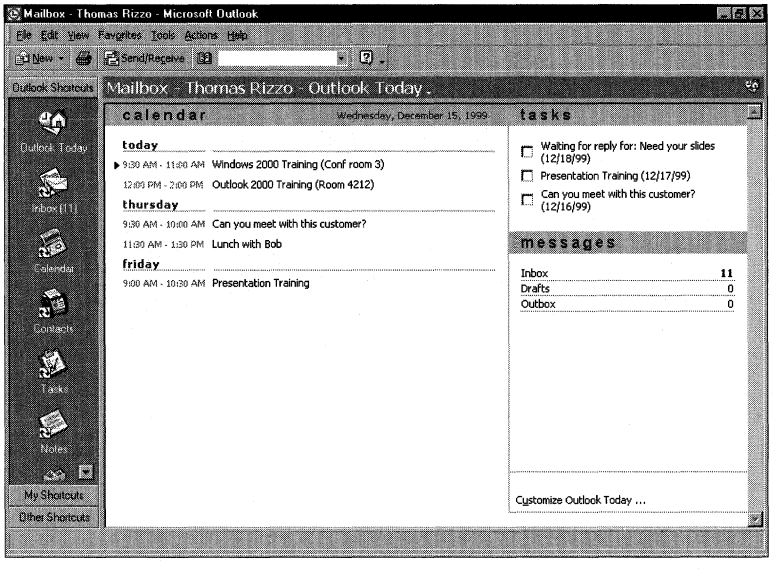


Figure 7-1. *Outlook Today is an HTML feature that was added in Outlook 98. Users can see different types of information in a single window.*

Outlook Today Technologies

The standard Outlook Today page takes advantage of a feature in Microsoft Internet Explorer called data binding. Data binding allows Outlook to quickly display the user interface for the Outlook Today page while asynchronously downloading data from the Exchange server (that is, the computer running Exchange Server) to the Outlook client. Once the data is downloaded, Outlook can modify the data without making additional trips to the server. On the default Outlook Today page, there are three separate data binding tables, one each for your Calendar, Tasks, and Mail. To modify the location of these tables or to add new functionality to the Outlook Today page, you need to know how to modify HTML. Since Outlook Today leverages Internet Explorer, you can even use Dynamic HTML (DHTML) in your customization. However, remember these limitations when customizing the Outlook Today page:

- Modifying the page might slow performance because Outlook has to retrieve information from other data sources. Try not to build complex applications in the Outlook Today window; instead, build either an Outlook form or a standard HTML application, and then add the HTML application to Outlook Web Access.
- Although you can add external links to Internet sites on your Outlook Today page, Outlook Today will not verify the security of the site. To use the security capabilities you have in your browser, add a link in Outlook Today that launches a separate browser, such as Internet Explorer, to render the page. If you are sure of the content that you are linking to, you do not have to browse the link in a separate browser.
- You cannot add the Outlook Today page as an Active Desktop item. Currently, the Outlook Today functionality works only when hosted in the Outlook window.

NOTE The standard Outlook Today pages are hosted in a resource dynamic link library (DLL), which improves the performance of the Outlook Today application. When modifying your Outlook Today pages, you have the option to place your custom HTML pages and images in a resource DLL. However, this book covers only customizing Outlook Today and saving these customizations as HTML files. To learn how to compile your files into a resource DLL for Outlook Today, visit *http://www.microsoft.com/office/ork/*.

Outlook Today in Outlook 2000

There are some differences between the Outlook Today page in Outlook 2000 and the Outlook Today page in previous versions of Outlook. In this section, we'll look at the most significant changes made to Outlook Today in Outlook 2000.

User Interface Changes

One of the major differences in the new version of Outlook Today is the user interface. In Outlook 2000, the ability to change both the folders that are listed and the style of the Outlook Today page is integrated directly into the user interface. Figure 7-2 shows the new interface for the Outlook Today page, and Figure 7-3 shows the Customize Outlook Today page.

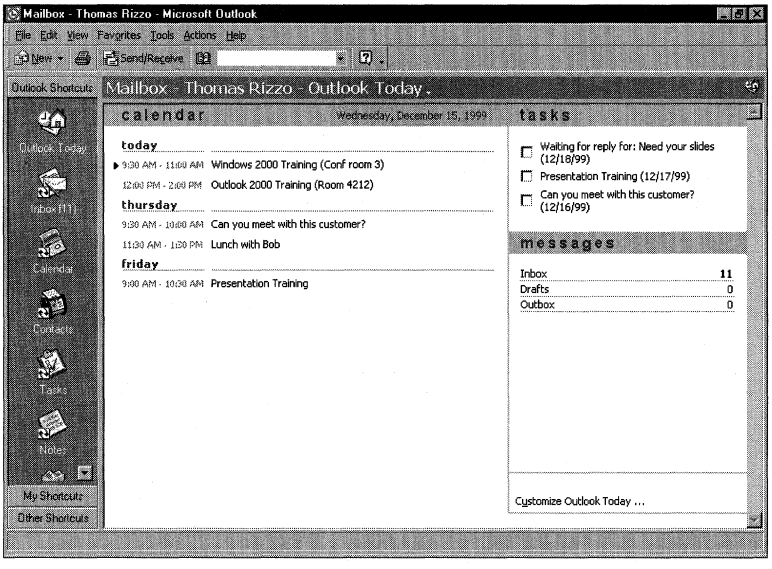


Figure 7-2. *The new interface for Outlook Today in Outlook 2000.*

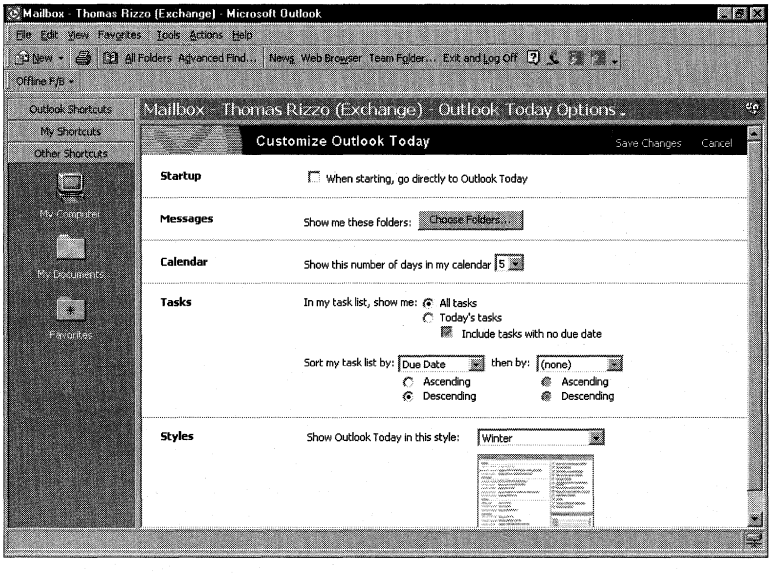


Figure 7-3. *The customization page for Outlook Today in Outlook 2000.*

Security Changes

The new Outlook Today page has two security changes from previous versions. First, browsing security is disabled for the default Outlook Today page. When the user clicks on links or navigates to other pages, the browser turns on security for those pages. Therefore, you need to make sure that the controls or scripts you use in your default

Outlook Today page are secure. However, you also need to make sure any links to which the user tries to navigate are not affected by the browser security being on.

Second, if the default page is a frameset, browser security is disabled when the parent frame is the page displayed in Outlook Today. This means security will not be enabled when a user makes any changes to or performs any navigation in child frames. Security will be enabled only when the user changes the parent frame to navigate to a different location. For this reason, you should force the links in the child frames of a frameset to open in the parent frame.

Browser Issues

Since the new version of Outlook Today implements a subset of Internet Explorer technology, there are two browser issues that you should be aware of. First, the frameset or Iframe HTML elements might not work as you expect. For example, when you set the target to be a blank frame, the new window will open in Outlook rather than in a separate Web browser. Second, you might find that HTML forms or tabular data controls do not work correctly.

To remedy these problems, you will have to modify Outlook Today to use a full version of Internet Explorer. Note that when you make this change you will lose the performance enhancements of Outlook Today, such as the <RENSTATICTABLE> elements. To modify Outlook Today to use the full Internet Explorer version, add two settings to the registry in the following location:

```
[HKEY_CURRENT_USER\Software\Policies\Microsoft\Office\9.0\Outlook\Webview\mailbox]
"url"="http://YOURLOCATION/default.htm"
"navigation"="yes"
```

You also can make this change programmatically on a folder-by-folder basis by setting the *WebViewAllowNavigation* property to True on the MAPIFolder object that represents the folder. By default, folder home pages, which you'll learn about later in the chapter, use the full version of Internet Explorer to display content.

Customizing Outlook Today

Customizing the Outlook Today page and saving the result as an HTML file involves a few steps, which are outlined in the following sections.

Retrieving the Outlook Today Source

To view the source of an HTML page in Internet Explorer, you can right-click on the page and select View Source from the context menu. If you right-click on the Outlook Today page to view the source of the HTML page, however, you will notice that the View Source option is not available—Outlook disables this option. To retrieve the HTML source for the Outlook Today page, you can use Internet Explorer. Follow the steps listed on the following page.

1. Start Internet Explorer. In the address box, type the following URL, adjusting the path as necessary:

   res://c:\Program Files\Microsoft Office\Office\1033\Outlwvw.dll/Outlook.htm

   NOTE The *1033* in the path above specifies the locale for your language. If you are using a version of Microsoft Office with a language other than English, you'll need to change the 1033 to the correct value for your language.

2. After you press Enter, you will get a script error. Select No to decline continuing running scripts and debugging the current page.
3. From the View menu, select Source to display the source in Notepad.
4. From the File menu in Notepad, select Save As and save the file to your hard disk as Outlook.htm.
5. Perform a search in Notepad for the three instances of *display:none*, and replace them with *display:*.

Modifying the Registry

You need to modify your registry settings so that Outlook knows you want Outlook Today to point to a new file. You could write a program that performs this step for your customers:

1. Start the Registry Editor (Regedit.exe).
2. Find the following registry key:

   HKEY_CURRENT_USER\Software\Microsoft\Office\9.0\Outlook\Today

   If the Today key does not exist, create it. Right-click on the Outlook key, point to New, and select Key. Type *Today* for the name of the key.
3. Add a new string value to the Today key by right-clicking on the Today key, pointing to New, and selecting String Value. Type *Url* for the value name.
4. Double-click on the URL string icon to edit it. For its value data, type the path to the Outlook.htm file that you saved previously. For example, file://C:\Outlook.htm. When finished, click OK.

Customizing the HTML File

The final step is to customize the HTML file. The easiest way to do this is by using a text editor, because the HTML code in the page contains special formatting that will make the file appear incorrectly in Microsoft FrontPage. Let's review some of the ways you can customize the Outlook Today HTML page.

Changing fonts

Since Outlook Today uses cascading style sheets, you can easily change fonts and styles by modifying the style sheet. For example, you could change the font for

important items by changing the *.itemImportant {color:red}* line in the style sheet to the desired font and color.

Adding text, images, and hyperlinks

Using HTML, you can add new text, images, or hyperlinks to your Outlook Today page. Remember that if you link to an external Web site, Outlook will not implement the security that you set in your standard Web browser. So use the following code when placing external links in the Outlook Today page:

```
<a style="cursor:hand"" class="itemNormal"" onclick=
"window.open('http://www.microsoft.com/exchange/','_blank');">
Exchange web site</a>
```

Adding components

Since Outlook Today uses Internet Explorer, you can place any components on your page as long as Internet Explorer supports them. These components can include ActiveX controls as well as Java applets. However, make sure you trust the source of the component because Outlook does not check the component's security credentials.

Adding script

Outlook Today supports both JScript as well as Microsoft Visual Basic Scripting Edition (VBScript). From script, you can access the Outlook object library and use its functions in your Outlook Today page. You can see an example of a customized Outlook Today page for the Account Tracking application in Chapter 6.

Using the Outlook Databinding controls

The Databinding controls provided in Outlook can quickly bind and display information contained in Outlook folders. The Databinding control places the information dynamically bound from Outlook data into the sections of the Outlook Today HTML page designated by the RENSTATICTABLE elements. You can point these controls at public folders rather than at default folders such as Calendar or Tasks. We'll look at the Databinding control in detail in Chapter 11, when we discuss the Digital Dashboard.

ACTIVE SERVER PAGES

In this section, we'll explore Active Server Pages technology. You should know about ASP for several reasons. First, the Outlook HTML Form Converter, a conversion tool that migrates Outlook forms to HTML forms, utilizes this technology. We'll examine the converter later in this chapter. Second, ASP is used in other areas, including Collaboration Data Objects (CDO) and Active Directory Services Interfaces (ADSI). We look more closely at CDO in Chapter 12 and ADSI in Chapter 15.

ASP Fundamentals

Active Server Pages are standard text files that contain HTML and script. The script can be written using any ActiveX scripting language, such as VBScript or JScript. The HTML files that most Web developers write differ from ASP files in two significant ways. First, instead of having an .htm or .html file extension, ASP files have an .asp file extension. When you install IIS, as a part of your installation you also install an Internet Server Application Programming Interface (ISAPI) component that processes all files with an .asp extension. This ISAPI component parses the ASP file and executes the appropriate script. Second, the actual script is processed on the Web server. The processed results can include client-side scripting code but for the most part is just simple HMTL. Returning only HTML has two benefits: any modern Web browser can view the results of an ASP application and the additional capabilities of the browser is less of an issue.

Since Active Server Pages supports VBScript, you can easily move from developing Outlook forms to developing ASP pages. The only difference in the development process is that you should use the CDO library to write your Active Server Pages application rather than the Outlook object library, because CDO was designed to be multiuser and server-based.

The following code is an example of an ASP application. This example uses the VBScript function *Now* to print the date and time that the ASP application ran on the Web server.

```
<%@ LANGUAGE="VBSCRIPT"%>
<!DOCTYPE HTML PUBLIC ""-//IETF//DTD HTML//EN">

<HTML>
<HEAD><TITLE>ASP Example</TITLE></HEAD>
<BODY>
<H1>I was created on <%=Now()%></H1>
</BODY>
</HTML>
```

As you can see, the syntax of the ASP script is a little bit different from the syntax for Outlook code. To tell the Web server that you want to run a script on the server, you must enclose it in special characters: <% and %>. Active Server Pages supports placing your script directly in your HTML code—the script does not have to be in a separate section of the HTML file.

Take a look at the first line of the code:

```
<%@ LANGUAGE="VBSCRIPT"%>
```

ASP assumes that the default language for server-side script is VBScript. If you replace VBSCRIPT with JSCRIPT, you can write server-side JScript code.

> **NOTE** You can specify the default ASP language for an application in the Management Console for IIS. Open the Properties window for an application, and in the Applications Settings area, click the Configuration button. On the App Options tab, type the desired default language in the Default ASP Language text box.

You might be wondering what the <%=*Now*()%> code does in this example. The equal sign (=) indicates that the code should evaluate the expression, which in this case returns the current date and time. As you will see, the equal sign in ASP is a shortcut for calling the *Write* method of the Response object.

Global.asa

If you've viewed the actual directories that contain .asp files, you might have noticed a certain file with the .asa extension: Global.asa. This is a special file in ASP applications that allows you to include global code that executes when an application starts and ends and also when a session starts and ends. One thing to remember is that the Global.asa is an optional file for your Web applications. A skeleton Global.asa file is shown here:

```
<SCRIPT LANGUAGE="VBSCRIPT" RUNAT="Server">
   Sub Session_OnStart
      'Put your session startup code here
   End Sub
   Sub Session_OnEnd
      'Put your session termination code here
   End Sub
   Sub Application_OnStart
      'Put your application startup code here
   End Sub
   Sub Application_OnEnd
      'Put your application termination code here
   End Sub
</SCRIPT>
```

The Global.asa file contains stubs for your session and application start and end subroutines. To understand when these subroutines are called, you must understand what exactly constitutes a session and an application inside ASP.

Normally when you browse Web pages, the Web server does not remember who you are or where you have been, and it does not store any values associated with you. One of the features of ASP is that it transforms the applications you can build on the http protocol from being stateless to being able to track the state of users. This ultimately lets you create global variables that are maintained for users throughout an application.

An ASP application consists of a virtual directory and associated files. But to understand when an ASP application starts and ends, you'll need a little bit more

explanation of how ASP works. For your Application_OnStart subroutine to be called, the first user must request an .asp file from the virtual directory of your ASP application. The user can request an HTML file or other types of files from that directory. However, these requests will not cause the Application_OnStart subroutine to be called. The user must explicitly request an ASP file. This is the only time this subroutine will be called, unless you restart the application. Restarting the application usually consists of restarting the Web service.

You should use the Application_OnStart subroutine to initialize global variables across the lifetime of the Web application. A good example of a variable to initialize or set in your Application_OnStart subroutine is one that counts the number of users who have used your application. To improve performance, for every user in your ASP application, you should initialize in the Application_OnStart subroutine any server components that you will use. Figure 7-4 illustrates a Web browser sending a request to an ASP application for the first time.

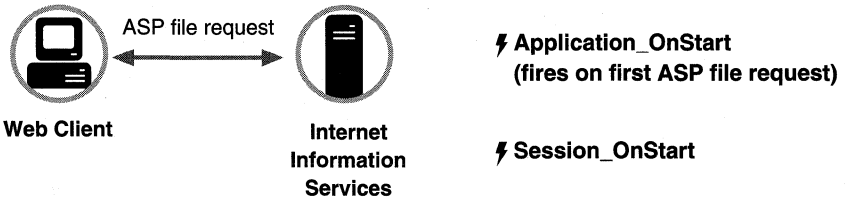


Figure 7-4. *When the first user of an application requests an .asp file, the Application_OnStart event is fired and then the Session_OnStart event fires.*

When the user who first requested the ASP page also browses an .asp file in your application, the Session_OnStart event is called. Unlike the Application_OnStart event, the Session_OnStart event is called for any user who makes an application file request. With ASP, each user of your application is considered to have a distinct session with the Web server. As a user browses Web pages in your ASP application, ASP implements and maintains state in a session by using cookies—whenever a user connects to your application, a file containing information (a cookie) is saved on the user's machine. When his session ends and he closes his Web browser, the cookie is removed and the session is invalidated. If he reconnected to your application, his machine would receive a new cookie and a new session would be started. For this reason, the users of your application must support and accept cookies; otherwise, your ASP applications will not fully function. You can still use the server-side script of ASP, but you cannot maintain state information for any of your users.

The Session_OnStart event is best used to initialize session variables for individual users. Session scope variables include a connection to Exchange Server for an individual user and personalized information that a user has set in your application—

for example, a user could specify a background color for Web pages that is stored in a session variable. Then, each page the user accesses from your site during a session could be displayed in her personalized background color. Figure 7-5 shows each Web browser starting a new session when accessing an ASP application.

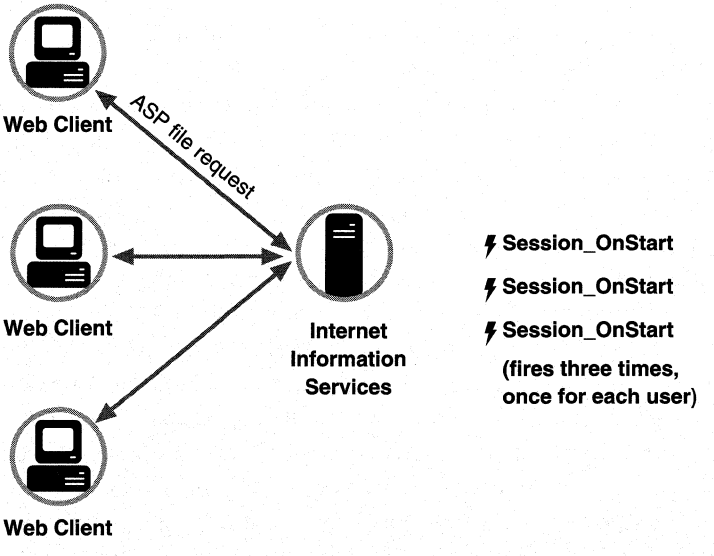


Figure 7-5. *Whenever a new user accesses your ASP application, the Session_OnStart event fires. Application_OnStart fires only when the first user accesses your application.*

The Session_OnEnd event is called when the session with the Web server ends. This end state can be reached in two ways:

- When the user has not requested or refreshed a Web page in the application for a specified amount of time
- By explicitly calling the *Abandon* method on the Session object

By default, IIS sets the timeout interval at 20 minutes. You can change this interval either through the administration program for IIS or by setting the *TimeOut* property on the intrinsic Session object in ASP. For example, to set a particular script timeout to 10 minutes, you would write the following code in your ASP application:

```
<% Session.TimeOut = 10 %>
```

The second way to reach the end state—by explicitly calling the *Abandon* method on the Session object—immediately ends the session and calls the Session_OnEnd event.

> **NOTE** Applications discussed in later chapters (CDO Helpdesk, Event Scripting Expense Report, Routing Objects Expense Report) provide a logout menu option. This option calls another ASP file, which calls the *Abandon* method on the Session object to end the session.

One final note about sessions: you have to be careful when you redirect people to other virtual directories in your application. Developers, including me, commonly make the mistake of redirecting users to another virtual root and forget that this is considered by ASP to be an application. When you do this, the session variables you establish in one application will not transfer to the other application. If you want to share session variables between the two applications, you should place the second application under the same virtual directory in IIS as the first application.

When a Web application ends, the Application_OnEnd event is called. You end a Web application in one of two ways: by shutting down the Web server, or by stopping your application by using the Unload button in the IIS administrator. To use the Unload button, you must be running your Web application in a separate memory space. So make sure you save any application scope variables to a persistent medium, such as to your Exchange server or to a database, so that when your application restarts, the Application_OnStart event can reload the values. For example, you don't want a user-counter variable to restart at zero every time your application restarts. You should also destroy any server objects that you have created with an application scope. This will eliminate potential memory leaks on your server.

Built-In ASP Objects

The real power of ASP applications is that you can write server-side scripts and use their intrinsic objects. ASP and its built-in objects enable you to generate custom responses and maintain state information. The following section describes, in detail, five built-in objects in ASP: Application, Session, Request, Response, and Server.

> **NOTE** The only object not covered here is the ObjectContext object, available in IIS version 4.0. This object can be used for creating ASP applications with transaction capabilities. For more information on the ObjectContext object and transactions, consult the IIS product documentation.

Application Object

The Application object is used to store global data related to an application that can be shared among all users. By using the methods and properties of this object in your application, you can create and set variables that have an application scope. To make sure that you do not run into concurrency issues when setting your application-level variables, since multiple users can be using the same application simultaneously, the Application object provides two methods: *Lock* and *Unlock*. These methods serialize the access to application-level variables so that only one client at a time can read or modify the values. The following example shows how to use the *Lock* and *Unlock*

methods to increment a user-counter variable whenever a user accesses the application. The example also shows you how to set and retrieve application-level variables by using the *Application"("VariableName")* syntax:

```
<HTML>
<HEAD>
<TITLE>Example: Application Object</TITLE>
</HEAD>
<BODY>
<%
   Application.Lock
   Application("NumVisitors") = Application"("NumVisitors") + 1
   Application.UnLock
%>
Welcome! You are visitor #<%=Application"("NumVisitors")%>.
</BODY>
</HTML>
```

The Application object also contains two other collections beyond the variables collection—Contents and StaticObjects—which allow you to browse through the application-level objects and variables you have created. You probably won't use either of these collections in your final applications, but both of them provide great debugging functionality. For example, the Contents collection enables you to list all the items that have been added to your application through a script command, and the StaticObjects collection enables you to list all the items with an application scope that have been added using the <OBJECT> tag. By adding debug code to your application at design time, when you run into application object problems, you can make ASP list all the objects you have created with an application scope. The following code illustrates creating debug code for both the Contents and StaticObjects collections. You can see the code output in Figure 7-6.

```
<HTML>
<HEAD>
<TITLE>Debugging Application Objects</TITLE>
<%
   'Create some application variables
   Application.Lock
   Set Application"("oCDOSession") = _
      Server.CreateObject"("MAPI.Session")
   Application("counter") = 10
   Application.UnLock
%>
<P>Objects from the Contents Collection<BR>
<%
   for each tempObj in Application.Contents
      response.write tempObj & "<BR>"
```

(continued)

```
   next
%>
<P>Objects from the StaticObjects Collection<BR>
<%
   for each tempObj in Application.StaticObjects
      response.write tempObj & "<BR>"
   next
%>
</BODY>
</HTML>
```

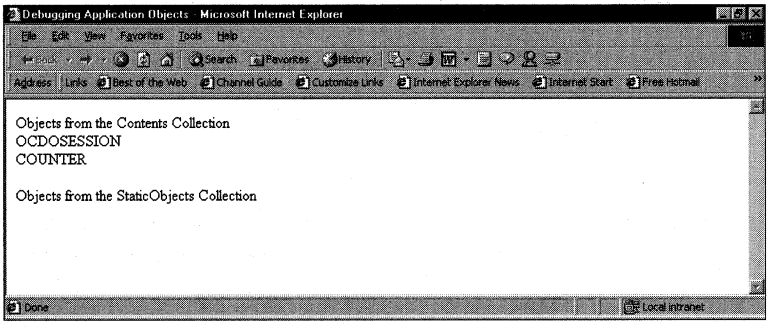


Figure 7-6. *The debug output for the Contents and StaticObjects collections. As you can see, objects and variables both can have an application scope.*

Session Object

The Session object is one you'll use a lot in your Web applications. It holds the variables for individual users across the Web pages in your application. When you place a variable in the Session object, that variable is valid only for the current user and cannot be shared among users in the same way that an Application variable can.

Like the Application object, the Session object contains the Contents and StaticObjects collections. You can also create session variables in the same way you create *Application* variables, by using the syntax *Session("VariableName")*.

The properties for the Session object include CodePage, LCID, SessionID, and TimeOut. The *CodePage* property represents the language code page that will be used to display the content for the HTML page. The Outlook HTML Form Converter, which you'll learn about later in this chapter, uses this property in its converted forms, as shown here:

```
<% @LANGUAGE=VBSCRIPT CODEPAGE = 1252 %>
```

You can use the *LCID*, or locale identifier, property in conjunction with the *CodePage* property. The *LCID* property stores a standard international abbreviation that uniquely identifies a system-defined locale.

The *SessionID* property returns to you the unique session identifier for the current user. You should remember, however, that this ID is unique only during the lifetime of the ASP application. If you restart your Web server and therefore restart

your Web applications, the Web server might generate the same IDs it already generated for the users before the Web application was restarted. For this reason, you should avoid storing these IDs and attempting to use them to uniquely identify a user of your application. If you always need to uniquely identify your users whenever they access your application, you should use globally unique identifiers (GUIDs) in cookies, which are saved on the users' computers.

The fourth property of the Session object is the *Timeout* property. This property enables you to change the timeout period associated with a particular ASP session. Remember that by default, the timeout is set to 20 minutes. If you know that your application will be used for less than 20 minutes, you might want to decrease the duration of the timeout so that sessions end more quickly and resources are returned to the Web server at a faster rate.

The only method of the Session object is the *Abandon* method. As mentioned earlier, by calling this method, the user's session with the Web server as well as any associated objects and variables for that session are destroyed. If the user attempts to reconnect to the Web application, a new session starts on the server.

Request Object

The Request object allows you to access the information that was passed from the Web browser to your Web application. The Request object is crucial in ASP applications since it enables you to access user input for your server-side scripts. For example, suppose a user fills out an HTML form that you created. Once the user clicks the Submit button on the form, the Request object contains the form information that was passed to the server. By using the collections of the Request object, you can retrieve that information and design your application to respond based on the user's input.

Request object collections

The Request object collections are created when the user submits a request to the Web server either by requesting an ASP file or by submitting an HTML form via clicking the Submit button. The three collections of the Request object that you'll primarily work with in your ASP applications are the Form, QueryString, and ServerVariables collections.

> **NOTE** For information on the other two collections, ClientCertificate and Cookies, refer to the IIS documentation.

To understand when to use these collections, you first need to know about the different ways information can be passed from the Web browser to the Web server. Normally in your Web applications, you use HTML forms to gather input from the user so that you can use it in your calculations or store it in a data source. There are two main ways input can get passed to the Web server from the client browser: via the *Get* method and via the *Post* method. The example that follows shows an HTML page that contains both methods on the same page.

```
<html>
<head>
<title>Forms Galore</title>
<meta name="GENERATOR" content="Microsoft FrontPage 3.0">
</head>
<body>
<form method="GET"" action="getinfo.asp"" name="GetForm">
   <p>What is your e-mail address?</p>
   <p><input type="text" name="e-mail" size="20"></p>
   <p><input type="submit" value="Submit" name="GetSubmit"> </p>
</form>

<form method="POST" action="getinfo.asp" name="PostForm">
   <p>What is your first name?</p>
   <p><input type="text" name="firstname" size="20"></p>
   <p><input type="submit" value="Submit" name="PostSubmit"> </p>
</form>
</body>
</html>
```

The Action attribute for each of the HTML forms specifies the same ASP file, getinfo.asp. The getinfo.asp file is shown here:

```
<HTML>
<HEAD>
<TITLE>Post and Get Methods Example</TITLE>
</HEAD>
</BODY>
<%txtRequestMethod = Request.ServerVariables"("REQUEST_METHOD"")%>
You selected to use the <B><%=txtRequestMethod%></B> Method.
<P><% if txtRequestMethod="GET"" then %>
You entered your e-mail address as:
<B><%=Request.QueryString"("email")%></B>
<% else %>
You entered your first name as: 
<B><%=Request.Form"("firstname")%></B>
<% end if %>
</BODY>
</HTML>
```

This ASP code uses the ServerVariables collection of the Request object to check whether the form's *Request* method was a *Post* or *Get* method. Once the file determines which method was used, it displays the correct information for that particular type of form. Figure 7-7 shows a sample of the *Get* method.

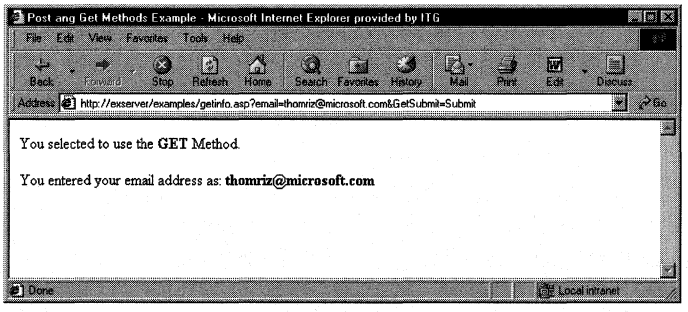


Figure 7-7. *When a user types an e-mail address and submits the form, the* Get *method is used to pass the information to the Request object.*

NOTE You can also retrieve other server variables such as *HTTP_USER_AGENT*, which returns information about which browser the client is using; and *LOGON_USER*, which represents the Microsoft Windows NT account the user is currently logged on to. For a complete list of server variables, see the IIS documentation.

As you can see in Figure 7-7 with the *Get* method, the information from the form is actually appended to the URL—for example:

http://exserver/examples/getinfo.asp?email=thomriz@microsoft.com&
GetSubmit=Submit

When data is appended to the URL using the *Get* method of a form, you use the QueryString collection of the Request object to retrieve the data. When using the QueryString collection, follow this format to retrieve the information:

```
Request.QueryString"("VariableName"")
```

Because the information that is passed to your application appears in the address of the user's browser, the user can see it, so you might want to limit when you use the *Get* method. Instead, consider using the *Post* method.

The *Post* method places the form information inside the HTTP header, hiding the information from the client. However, when the *Post* method is used to submit form variables, you cannot use the QueryString collection. Instead, you need to use the Forms collection of the Request object. In the preceding example, the line

```
Request.Form"("firstname")
```

retrieves the information the user typed into the First Name text box on the form. You can use this same syntax in your applications to retrieve information from an HTML form.

Response Object

The Response object is used to control the content that is returned to the client. For example, when you calculate a value on the server, you need a way to tell the ASP engine that you want to send the information back to the client. You do this by using the *Write* method of the Response object.

The *Write* method of the Response object will be the most commonly used method in your ASP applications. Even though you have not seen any explicit statements using the *Response.Write* method in the examples, they are there. The syntax <%=*Variant*%> is equivalent to <% *Response.Write Variant* %>. The shorthand version makes it easier for you to put these statements in your code quickly.

The Response object has a number of other collections, properties, and methods that you can use, such as the *Expires* property, which tells the Web browser how long to cache a particular page before it expires. If you do not want your clients to cache your Web pages, you would add the following line to your ASP files to cause your Web page to expire immediately on the user's local machine:

```
<% Response.Expires = 0 %>
```

The Response object allows you to buffer the output of your ASP page. This is useful if you want to hold back the output of your ASP code until the script completes its processing. The best example for using buffering is to capture errors in your code. For example, by turning buffering on using the command *Response.Buffer = True*, you can check throughout your ASP code whether an error has occurred. If one has, you can clear the buffer without sending its contents by using the *Response.Clear* method, and then you can replace the output with new output such as *Response.Write "An error has occurred. Please contact the administrator."* Finally, you can call the *Response.End* method, which sends the new contents of the buffer to the client and stops processing any further scripts in the ASP.

Server Object

The Server object provides you with utility methods and properties to modify the information on your Web server. This object is used extensively in ASP applications because it contains both the *CreateObject* method and the *ScriptTimeout* property.

The *CreateObject* method allows you to create an object on the Web server by passing in the ProgID for the object. Let's look at an example. To create a CDO object, you would type this in your ASP file:

```
Set oSession = Server.CreateObject("MAPI.Session")
```

ASP creates an object and passes that object to you in the *oSession* variable. By default, when you do this on an ASP page, the object has page-level scope. This means that when ASP is done processing the current page, the object is destroyed. Therefore, you might want to create objects on a page and then store them by assigning them to either session variables or application variables, as shown in this code snippet:

```
<%
    Set oSession = Server.CreateObject("MAPI.Session")
    Set Session("oSession") = oSession
%>
```

As you learned earlier, an object that is assigned either a session or an application scope will be destroyed when either the session or the application ends, respectively. The one issue to watch out for with the *CreateObject* method and some objects is potential performance loss. You can instantiate almost every object on your Web server as an ASP object, but some objects are specifically designed to run in a server-based, multiuser environment such as CDO. When you instantiate an object that was not designed for an ASP environment, the application performance might suffer if many people hit the page containing that object at the same time.

The *ScriptTimeout* property of the Server object allows you to specify how long a script should run before it is terminated. By default, an ASP script can run for 90 seconds before it is terminated, but this might not be enough time to retrieve data from a data source. By using the following syntax for this property, you can increase or decrease the amount of time the script will run before termination:

```
Server.ScriptTimeout = numseconds
```

Avoid increasing this number much beyond 90 seconds, because users who are waiting for long periods of time might assume the page did not load correctly, and they might click their Stop and then Refresh buttons continuously, flooding your Web server with requests.

Server-Side Include Files

One other powerful feature beyond the intrinsic objects of ASP is the ability to use server-side include files in your ASP files. Include files are just text files containing script or HTML that you want to add to your ASP page. Outlook Web Access, which you will learn about later in this chapter, relies heavily on server-side include files for common code libraries in its ASP files. Here are some examples of server-side include files:

```
<!-- #include file="library/vbsfunctions.inc" -->

<!-- #include virtual="/library/vbsfunctions.inc" -->
```

Server Components

ASP can take advantage of built-in objects and also use server components to add functionality to ASP. An example of two such components are Microsoft ActiveX Data Objects (ADO) and CDO. ADO allows you to connect to many types of databases; CDO allows you to connect to Exchange Server and other messaging servers. You can also write your own components using any COM-based development tool.

NOTE There are a number of other components packaged with ASP that you can use in your applications, including Ad Rotator, Browser Capability, Content Linking, Content Rotator, File Access, Page Counter, and Permission Checker. If you want to learn more about these components, you should refer to the documentation that ships with IIS version 4.0.

OUTLOOK WEB ACCESS

Outlook Web Access is an ASP application that Microsoft ships with Exchange Server version 5.5. This ASP application allows you to access your mailbox, calendar, and contacts as well as directory information using any standard Web client. The Outlook Web Access application is built on CDO and is one of the best tools for learning CDO.

In this section, you'll learn how to install Outlook Web Access on your Web server, which also installs the CDO library. You'll also learn about security when using Outlook Web Access. This security architecture is important since it also applies to any custom CDO applications you develop using ASP.

Installing Outlook Web Access

Before installing Outlook Web Access, you must have installed IIS version 3.0 or a later version with Active Server Pages. IIS 4.0 and Exchange Server 5.5 both require at least Windows NT 4.0 Service Pack 3, but it is recommended that you install Service Pack 4 or later, or better yet, use Microsoft Windows 2000 Server. You can download Windows NT 4.0 Service Pack 4 from *http://www.microsoft.com/windows/downloads/*. If you don't install Service Pack 4, you will need to install the Windows NT related fixes required for Outlook Web Access. You can download these hot fixes from the following:

ftp://ftp.microsoft.com/bussys/winnt/winnt-public/fixes/usa/nt40/hotfixes-postsp3/roll-up/

You can install Outlook Web Access on your Exchange server or on a separate server. Be aware that if you do install Outlook Web Access on a separate server, you cannot use Windows NT Challenge/Response as your authentication method. You'll learn more about security implications later in the chapter.

The architecture for your Web servers and Exchange servers can vary depending on the topology of your network environment and the requirements of your applications. For example, if few users will be accessing Outlook Web Access but you have a number of Exchange servers and you do not want to set up multiple Outlook Web Access servers for each Exchange server, you can set up just one Outlook Web Access server to talk to multiple Exchange servers. The opposite is true as well. You

can have multiple Outlook Web Access servers talking to just one Exchange server. Think of it as a Web farm of Outlook Web Access servers. This will work as long as you make sure that when a user starts a session with an Outlook Web Access server in a Web farm, that user stays with the same Outlook Web Access server until her session expires or she logs out. Remember that ASP sessions do not span separate ASP applications. If you use DNS round-robin techniques to farm a user out to multiple Outlook Web Access servers, that user's session will be lost when she changes to a different server.

To install Outlook Web Access, follow these steps:

1. Insert the Exchange Server version 5.5 CD in your CD-ROM drive.
2. If the Exchange Server welcome screen does not start automatically, launch it by double-clicking on Launch.exe.
3. Click on Server Setup And Components.
4. Click on Microsoft Exchange Server 5.5.
5. When setup starts, click Complete/Custom installation. (If you already have Exchange Server installed, click Add/Remove.)
6. In the Options list, check Outlook Web Access and click Continue. If you don't have Windows 2000 Server, or the Windows NT Service Pack 4 (or the Windows NT related fixes) and IIS installed, a message box will be displayed and you won't be able to continue the installation of Outlook Web Access.
7. The setup program will prompt you for the name of an Exchange server that Outlook Web Access should connect to. Type a name of a server that contains an entire replica of the Exchange Server directory. This sets up Outlook Web Access so that it automatically redirects itself to the Exchange server where the mailbox of the user resides. It also allows you to set up one Outlook Web Access Web server that talks to multiple Exchange servers.

After completing the Outlook Web Access installation, you need to update it by installing Service Pack 3 for Exchange Server 5.5. You can download or order this service pack from *http://www.microsoft.com/exchange/*. This service pack includes a number of enhancements for the Outlook Web Access client, such as the ability to access Outlook contacts from the Web.

After running the update, you need to set the proper permissions in the User Manager For Domains. Ensure that the Exchange users who will use Outlook Web Access have the following rights: Log On Locally and Access This Computer From Network.

Access your new Outlook Web Access server by typing the URL *http://OWAServer/exchange* in your browser, replacing *OWAServer* with your Outlook Web Access server name. From the displayed page, you can log in as an Exchange user or log in with anonymous access. (Anonymous access is discussed in more detail in Chapter 12.) Figure 7-8 shows how Outlook Web Access looks when you log on as an Exchange user.

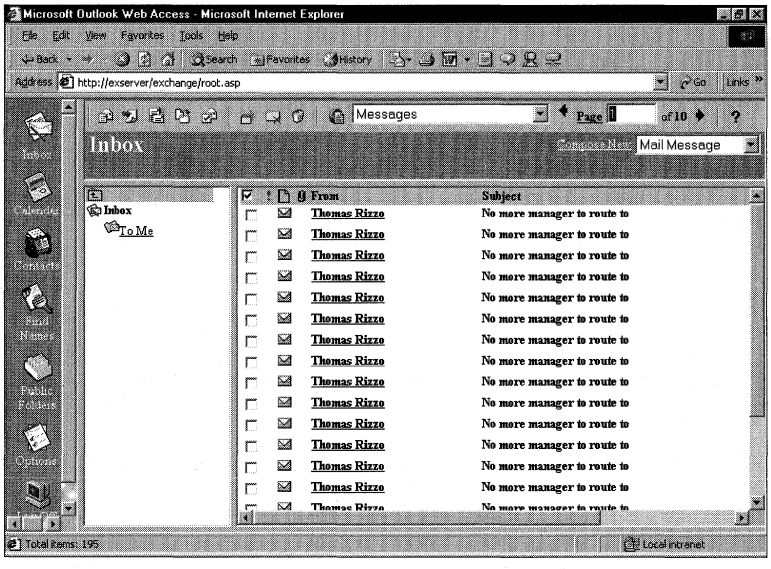


Figure 7-8. *Outlook Web Access.*

Outlook Web Access and ASP Security

Since the Outlook Web Access application utilizes Active Server Pages, you need to understand the ASP security architecture and how best to configure it for your environment. If you configure the environment incorrectly, you will run into problems when attempting to use authenticated access to the Outlook Web Access application or to any of your CDO Web applications requiring authenticated access. This section describes how ASP security works and how you should set up Windows NT to support the type of security you want for your Web applications.

ASP Security

When IIS is first installed, it creates a Windows user account named IUSR_*computername*, where *computername* corresponds to the current computer name. This account is assigned to the Guests account group, is given a random password, and is granted the right to Log On Locally. Whenever a user browses a Web page, this account attempts to access the page on behalf of the user. If the IUSR_*computername* account does not have the proper permissions to access the page, the request is rejected with this error message: "401 Access Denied". The Web server then informs the Web

browser which authentication methods the Web server will support—either Basic authentication or Windows NT Challenge/Response authentication—depending on your settings for your IIS server.

Basic authentication

Basic authentication is supported across all Web browsers. When the Web server informs the client that it supports Basic authentication, the Web browser displays a message box asking the user for a user name and password. Once the user types this information in, the Web server tries to invoke the request using the identity of the supplied user rather than the IIS anonymous account. It is a good idea to pass in your domain name as well as the user name in the authentication dialog box in the Web browser using the syntax *domain\username*.

Basic authentication, if used over Internet connections, can present some security concerns, because the user name and password typed into the authentication dialog box is transmitted to the server as clear text. If you do use Basic authentication over Internet connections, use it in conjunction with Secure Sockets Layer (SSL). SSL will encrypt the connection between the Web browser and the Web server so that any information passed between the two cannot be viewed by unauthorized individuals.

For the Web server to impersonate the user whose name is typed into the authentication dialog box, the Web server must log on as that user. By default, Windows NT does not give regular users the Log On Locally right on the server computer. For this reason, you must give all the users you expect will use your Web application with Basic authentication enabled the Log On Locally right on your Windows NT server, which runs your Web server. The easiest way to do this is to grant all your domain users the Log On Locally right in the User Manager for Domains.

Windows NT Challenge/Response authentication

Windows NT Challenge/Response, or NTLM, authentication is the most secure form of authentication because the user name and password are not sent from the Web browser to the Web server. Instead an encrypted challenge/response handshake mechanism is used. Unlike Basic authentication, NTLM typically does not prompt the user for a name and password. The Windows NT security credentials of the Web user currently logged on are sent to IIS and are used to access the requested resource. IIS then changes to the context of the specified user and attempts to access the resource. If this fails, the user will be prompted for a user name and password.

> **NOTE** For Windows NT Challenge/Response to work correctly, the users you are trying to authenticate must have Access This Computer From Network rights in the User Manager for Domains. This is normally enabled for users by default.

A one-way encryption method is used, meaning the mechanism validates the user without sending the password to IIS. IIS doesn't know the user information and cannot use it to access other resources on other machines. Essentially, this is a problem

of delegation. When IIS attempts to access a resource on another machine, the other machine will prompt IIS for user credentials. Since IIS does not have the password for the user, it cannot return the correct information to the other machine. For this reason, you cannot use the Windows NT Challenge/Response authentication method with Outlook Web Access when your Outlook Web Access server is on a server different from your Exchange server. IIS cannot remotely send the authentication to the Exchange server when the Windows NT Challenge/Response method is used. If you're using IIS 4.0, this is a gotcha. If you're using IIS 5.0, however, you can solve this problem with the new features of IIS 5.0. For more information on Windows 2000 and IIS 5.0, see the section titled "Windows 2000 and IIS 5.0" later in this chapter.

A second gotcha of the Windows NT Challenge/Response method is that you cannot use it over proxy connections for the same reasons just discussed. So when setting up your Web server, consider NTLM's security advantages as well as its limitations. One way to solve some of these problems is to upgrade to Windows 2000 and IIS 5.0.

A third gotcha for NTLM is that at the time of this book's publication, NTLM is supported only by Internet Explorer. This means that if you have a mixture of Web browser clients accessing your application, you might want to enable both Windows NT Challenge/Response and Basic authentication. If you enable only Windows NT Challenge/Response, when your Netscape Navigator users attempt to access a secure resource or page, they'll receive a message denying them permission. With both security methods set up, if Windows NT Challenge/Response fails, Basic authentication will be used.

ACLs

Another way to restrict access to your Web pages is by setting NTFS file permissions, or access control lists (ACLs), on your actual ASP files and directories. Doing so controls who can and cannot read the files. IIS respects the ACLs on the files, and if you have authentication enabled on your IIS server, IIS will use it to attempt to verify users and their individual permissions on the files. Be careful when setting permissions on files, however, because if the permissions you set are too restrictive, users will not be able to use your application.

Special Considerations for Setting Up Outlook Web Access

ASP security and Outlook Web Access security work in the same way when a user is being authenticated, but when you set up Outlook Web Access on your Web server, you need to keep some access issues in mind. The following section describes file permission issues when users try to access Outlook Web Access on a Web server, problems that could ultimately cause trouble in your CDO applications.

File Permissions for the Outlook Web Access Files

Outlook Web Access is installed by default in a subfolder named Webdata, located under Exchsrvr. If you change the NTFS file permissions for this folder, at the minimum you should enable Read access on it and its subfolders. For temporary work, Outlook Web Access uses another subfolder under Exchsrvr, named WebTemp. Make sure that the permission for this folder is set to Change because Outlook Web Access needs to create and delete items in it.

Exchange Server Search Permissions

If you are using the Search permissions in Exchange Server version 5.5 to restrict access to information in the Exchange Server directory for your users, you need to make sure that the Everyone and Directory Anonymous accounts have Search permissions at the Exchange Site or Configuration container level. If you do not grant these permissions, a user might get an error message stating that the Exchange server is down or that HTTP access has been disabled. Your CDO applications could fail as well. For more information on this error, be sure to check out the following Knowledge Base articles in MSDN: Q173455 "OWA Returns Exchange Server Down Error Message"; Q175892 "Permissions Required for Outlook Web Access"; Q180417 "Error Msg: Sorry! The Microsoft Exchange Server Is Down."

Installing Outlook 8.03 on Your Outlook Web Access Server

If you install Outlook 8.03 on your Web server after installing Outlook Web Access, Outlook will register an older version of the CDO library. Most commonly, users won't be able to access or render calendar information in Outlook Web Access or CDO applications because the older version of the library did not support this. To fix this problem, type the following at the Run command, which is accessed from the Start menu on your Outlook Web Access server: *regsvr32 cdo.dll.*

WINDOWS 2000 AND IIS 5.0

Whereas IIS 4.0 and Windows NT 4.0 provided an excellent platform for building Web applications such as OWA, Windows 2000 and IIS 5.0 provide an even better one. For this reason, Microsoft supports running OWA on IIS 5.0 and Windows 2000. To run OWA on IIS 5.0 and Windows 2000, you must upgrade your OWA server to at least Exchange 5.5 Service Pack 3. Using Windows 2000 and IIS 5.0 offers a number of benefits, especially for ASP developers, and I'll discuss them in the next few sections.

Improved ASP Support

In versions of IIS prior to 5.0, the scope of a Web application was a virtual directory. So if you accessed an application in the virtual directory *sales,* and then you transferred the user to the virtual directory *contacts,* you lost all the application information

from the *sales* application. Why? IIS considered these two Web applications to be separate because they resided in distinct virtual directories. However, with IIS 5.0, which uses the *Server.Transfer* method, this directory limitation is removed. With the *Server.Transfer* method, you can specify another ASP file to execute in a different ASP application without losing all your existing variables in the calling ASP application.

IIS 5.0 also supports much better error-handling than previous versions. When IIS detects an error in your application, it passes you an ASPError object. Using this object, you can determine the error number and the line of the source code as well as the number of the code line that caused the error. From this information, you can display custom error information to your users.

Improved Scripting Support

IIS 5.0 includes new versions of both VBScript and Jscript. While I can't cover all the new features of both languages, I do want to make you aware of two key enhancements in VBScript. The first one is support for the With statement. This support makes it easier for you to instantiate an object and then call methods and set properties on the object without rewriting a bunch of code. The second enhancement is support for regular expressions, which allows you to perform complex evaluation and manipulation of string variables.

Improved Security Features

IIS 5.0 supports the standard Digest authentication. Digest authentication is similar to Basic authentication, but Digest authentication does not send the user's password over the wire. Instead, Digest authentication uses a hashing algorithm to form a hexadecimal representation of a combination of user name, password, the requested resource, and the HTTP method.

WebDAV Support

IIS 5.0 supports WebDAV. WebDAV is a set of extensions to HTTP that allows you to send to your Web server commands that will open, edit, move, search, or delete files. Exchange 2000 supports WebDAV, so you'll learn more about it when I discuss Exchange 2000.

THE OUTLOOK HTML FORM CONVERTER

In previous chapters, you learned how to develop Outlook solutions using the features of Outlook, such as forms. These forms, however, work only with Outlook on machines running Microsoft Windows 95 or later versions, or Microsoft Windows NT 4.0 or Windows 2000 Server. There are still many 16-bit, UNIX, and Macintosh clients for whom developers need to design collaborative solutions. To provide cross-platform

support for forms, Microsoft offers the Outlook HTML Form Converter. This converter allows you to take your Outlook solutions and turn them into HTML, ASP, and CDO-based applications that can be viewed by any standard browser such as Internet Explorer and Netscape Navigator. Once you convert your application to HTML, you can use any standard Web development tool—for example, FrontPage—to edit the HTML output of the converter. Once you convert the form, your users can work with either the Outlook version of the application or the HTML version of the application.

While this technology is a great step forward for cross-platform collaborative solutions, the HTML environment has some limitations and does not provide the same level of functionality as Outlook. This section describes what the Outlook HTML Form Converter is, how the converter works, and what the Web forms library for Outlook Web Access is. I also provide tips for developing Outlook solutions that can be more easily converted to Web solutions.

Software Requirements of the Converter

Before you attempt to convert your forms, you must meet a few software requirements. First, you must have Outlook Web Access installed on one of your servers running Internet Information Services. Installing Outlook Web Access was discussed in the previous section. Second, you must have either Outlook 97 (version 8.03 or later), Outlook 98, or Outlook 2000 installed on the machine on which you are going to convert the forms. Make sure that you install the converter *after* you install one of these versions of Outlook. Third, you need to have Exchange 2000 or Exchange Server 5.5 with Service Pack 3. The service pack includes the Outlook HTML Form Converter as well as some improvements to Outlook Web Access that allow you to view Outlook contacts from any standard Web browser. To install the Outlook HTML Form Converter, run Fcsetup.exe in the Formscnv folder of the Service Pack 3 CD for Exchange 5.5. Finally, on the client, users of converted forms can use any version of Outlook or no version of Outlook—that is, they don't even need to have Outlook. They need only a Web browser to use the forms.

Components of the Converter

The Outlook HTML Form Converter's architecture consists of a number of components:

- *Conversion wizard.* This is the user interface that walks you through converting the form.
- *OFT-HTML COM object.* This reads the layout and data-binding information from the Outlook form and writes the corresponding HTML code to one or more files on the Web server.

- *Form Converter templates.* These templates are used as base templates for the converted file.
- *Template processor object.* This object customizes the base templates to create the converted form.

Features of the Converter

Before stepping through the actual Outlook HTML Form Converter, you should know about its features. The Form Converter does provide a large feature set that you can take advantage of, but it also has some limitations.

Form Locations

The Form Converter allows you to convert forms from multiple forms libraries as well as forms from the file system that are saved as .oft files. The types of forms libraries that the Form Converter supports are the Personal Forms Library, the Organizational Forms Library, and the Folder Forms Library. Using the Form Converter wizard, you can specify either the forms library or the specific .oft files you want to use. The Form Converter also supports selecting multiple forms for simultaneous conversion.

Form Types

The Form Converter currently supports forms based on the following types:

- *IPM.Note.* Mail message
- *IPM.Post.* Post form
- *IPM.Contact.* Contact form

IPM.Task, IPM.Appointment, and IPM.Activity (journal entry) are not supported by the Form Converter, but if you need to convert the user interface for any of them, you can copy their controls to a supported form type. You can then convert the supported form type with the copied controls and customize the converted form using an HTML development tool. If you try to convert an unsupported form type, the Form Converter will display an error message.

Convertible Features

The Form Converter can convert many of the Outlook controls with their corresponding layouts. Following is a list of features that the Form Converter can convert to HTML. Limitations are described.

- *Label control.*
- *TextBox control.*
- *ComboBox control.* If the Outlook form contains an editable ComboBox control, the Form Converter changes it to a noneditable ComboBox.

- *ListBox control.*
- *CheckBox control.*
- *OptionButton control.*
- *Frame control.*
- *CommandButton control.* Any images placed on the CommandButton control are lost since HTML does not support images on buttons.
- *MultiPage control.*
- *Image control.* Images that are bitmaps are converted to GIF files automatically by the Form Converter. In addition, any images that are clipped in Outlook by the Image control are shrunk automatically by the Form Converter for the HTML version of the form.
- *Background images on a form.*
- *ActiveX controls.* The Form Converter adds a commented out Object tag to the HTML form for the ActiveX control. However, the Form Converter does not package the ActiveX control as a CAB file nor does it add a CodeBase statement to the Object tag to point to the control's CAB file. To make the control appear on the form, you can package the control, add the CodeBase statement, and then remove the comments generated by the Form Converter.
- *Initial values.*
- *Required fields.* If a user attempts to change a tab in the HTML version of the form, the form will display a warning message that one of the fields is required on the form and also display the identifier text of the field. This text might appear in a format like SUBJECT_41_0_G, which might not be meaningful to your users. You can modify the error code to make it display the friendly name of the control rather than the identifier text.
- *Type checking and formatting.*
- *Read and compose layout.*
- *Hidden controls.* In most cases, the initial values of hidden controls are not maintained.
- *Built-in and custom actions.* If your actions call a custom form, be sure to also convert these forms, or the HTML version will return an error when the user invokes the custom action. The HTML version of the form can use only two rows of buttons to invoke custom actions. Since the width of the button is based on the amount of text on the button, and you're allowed only two rows of buttons, keep the length of the names of custom actions to a minimum.

- *Limited support for non-English forms.* The Form Converter provides limited support for non-English forms. It generates the ASP files and places them on the client machine in the correct subfolder for the language. For example, the output of a German form will be placed under the GER folder in Outlook Web Access, not under the USA folder. The Form Converter also places in the Form.ini file the appropriate code page for the language in which the form must be rendered. You must have installed on Outlook Web Access the language pack for the character set of the form you want to convert before running the Form Converter wizard. If you do not, you will not get the international options in the wizard.

Unconvertible Features

Following is a list of features that are not supported by the Form Converter. Details are provided for a few of these.

- *ScrollBar control.*
- *SpinButton control.*
- *TabStrip control.*
- *ToggleButton control.*
- *Formulas.* Even though the Form Converter does not convert formulas, it places the code for the formulas in commented out text in the HTML file. You can then uncomment and modify the formulas according to the needs of your application.
- *Script code.* The VBScript behind an Outlook form is not converted. Instead, it is placed in a text file, named Script.txt, which is in the folder that contains the ASP files for the converted form. The reason VBScript is not included in the HTML form is to accommodate cross-browser support. Since Netscape browsers do not support VBScript, you can either change the script in your form to server-side VBScript or rewrite the script as client-side JavaScript code.
- *Overlapped controls.* Since HTML does a poor job of supporting overlapped controls and layouts, the Form Converter does not convert overlapped controls. Instead, it places the controls as close as possible to one another on the form.
- *Calculated fields.* In a Contact form in Outlook, there are calculated fields such as FullName whose values are derived from other fields, such as FirstName and LastName. The Form Converter will convert forms that contain calculated fields, but these fields will become static fields. For example, the FullName field will not automatically change in an HTML form when either the FirstName or LastName field is changed.

Stepping Through a Conversion

Before you attempt to convert a form, you must first share the Webdata folder on your Outlook Web Access Web server. For a default installation of Outlook Web Access, this folder is located at C:\exchsrvr\Webdata. You must give yourself and other developers in your organization who will use the Form Converter at least Read and Write access to the share. Also, be sure to name the share *Webdata*. If you do not share this folder, the Form Converter will not allow you to finish the wizard.

To start the Form Converter, click the Start button, point to Programs, and select Microsoft Outlook HTML Form Converter. This will display a starting screen for the Form Converter. Click Next to begin the conversion process.

Selecting a Form Location

On the second screen of the Form Converter, shown in Figure 7-9, you can select the type of form you want to convert. As mentioned earlier, you can select forms from the Personal Forms Library, the Organizational Forms Library, and the Folder Forms Library or Outlook templates from the file system. On the second screen, you also specify the name of the Outlook Web Access Server where the ASP file will be placed after the conversion.

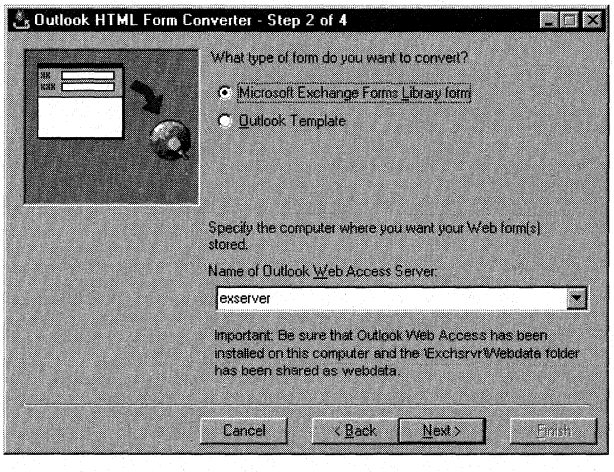


Figure 7-9. *Selecting the type of form and specifying the Outlook Web Access server in the Form Converter wizard.*

Selecting Specific Forms

Clicking Next might display a Choose Profile dialog box. If so, choose a profile or create a new profile, and click OK. Depending on the type of form you selected in the second screen, the Form Converter wizard will present you with a forms library view, shown in Figure 7-10, or the Open Outlook Template dialog box, shown in Figure 7-11. You can select multiple forms from either of these interfaces. The forms library view enables you to display form categories rather than form names in the

Outlook Forms list box. For forms in which these category properties were specified, this can make finding forms easier.

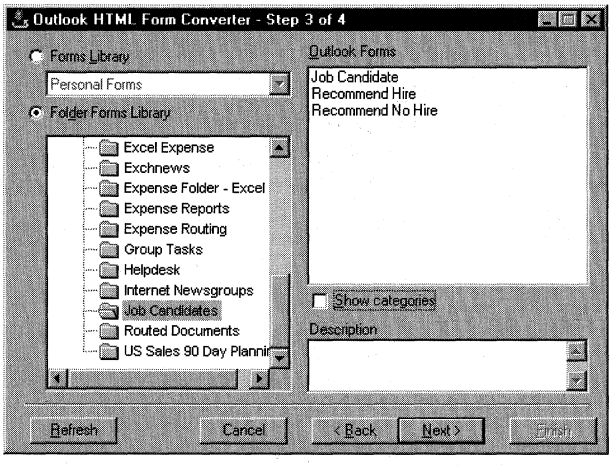


Figure 7-10. *The Forms Library view of the Form Converter wizard. You can view your forms by category or by display name.*

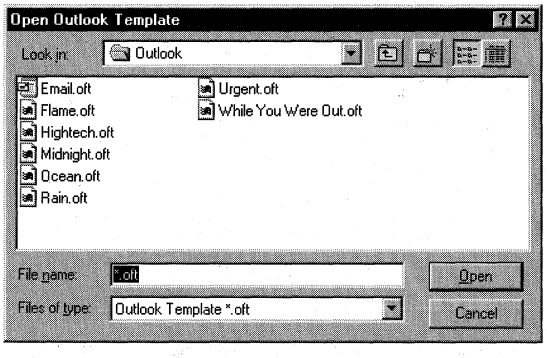


Figure 7-11. *The Open Outlook Template dialog box presented by the Form Converter wizard. You can select multiple forms to convert from the dialog box by using the Ctrl key.*

Choosing Conversion Options

In the final step of the conversion process, shown in Figure 7-12, you can choose how you want the forms converted. If there are international language packs installed on the Outlook Web Access server, this screen also provides a drop-down list from which you can select the language for the converted form. This final step of the wizard also gives you the option of always overwriting your existing form. If you do not check this option, the Form Converter will prompt you during the conversion about

overwriting the existing form. Checking the Layout Debug Mode check box enables debug mode on the converted form. If debug mode is enabled, the table borders for the HTML version of the form will be made visible so that you can easily see where and how the tables are laid out. You can use the borders to adjust the size and placement of controls on the converted form.

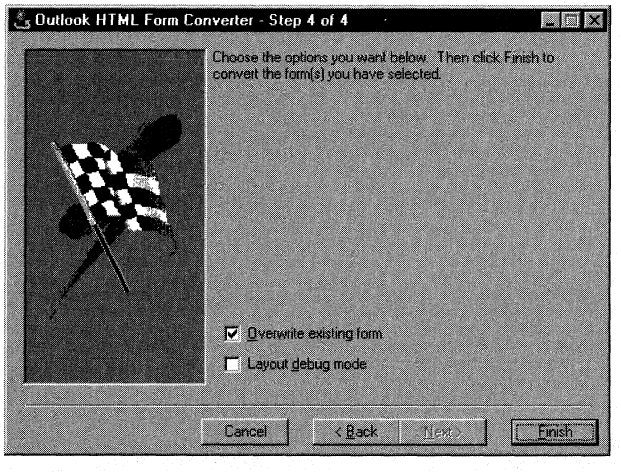


Figure 7-12. *On the last page of the conversion process, you choose options for conversion: you can overwrite your existing forms or enable the debug mode for your converted forms.*

Results of the Conversion Process

When you click the Finish button, the wizard converts your form. At the end of the conversion, the wizard displays its results. There can be three results:

- *Successful conversion.* This result means that the form was converted and the Form Converter has no suggestions for improving the layout and functionality of the form through post-conversion edits. A successful conversion is shown in Figure 7-13.

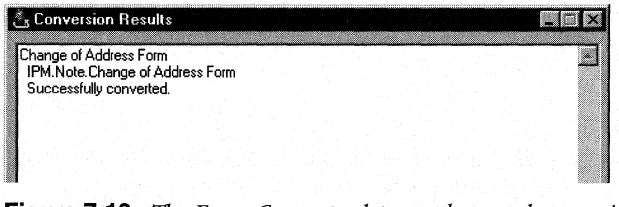


Figure 7-13. *The Form Converter lets you know when you've converted a form successfully.*

- *Successful conversion with To-Do list.* This result means that although the form was converted successfully overall, the Form Converter describes some post-conversion enhancements that you can make or some functionality on the form that did not get converted. The Form Converter creates a text version of the To-Do list it displays, named ToDo.txt, and copies it into the same folder it places the ASP files for the converted form. Figure 7-14 shows a successful conversion with a To-Do list.

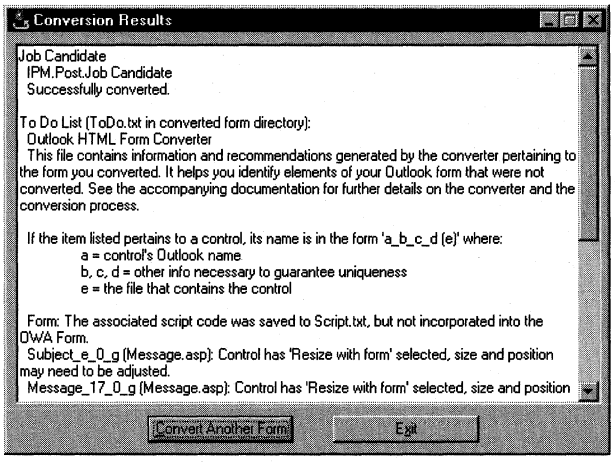


Figure 7-14. *A successful conversion with a To-Do list offers suggestions.*

- *No conversion.* This result means that the Form Converter could not convert the specified form. This unsuccessful conversion could be due to a number of issues, the most common being that the form you are trying to convert does not fall into one of the three supported form types. Figure 7-15 shows an example of an unsuccessful attempt to convert a Task form from an .oft file.

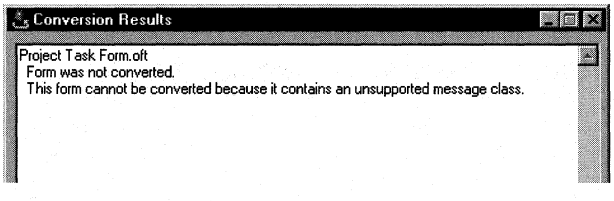


Figure 7-15. *Attempting to convert a task form results in the Form Converter returning a No Conversion result.*

Viewing the Results

The most common way to view the results of your conversion is to launch your Web browser and type *http://*OWAServer*/exchange/* for the URL, replacing *OWAServer* with your Outlook Web Access server name. When the Outlook Web Access Log On page is displayed, log on. Select Custom Form from the Compose New drop-down list as shown in Figure 7-16, and click the Compose New link.

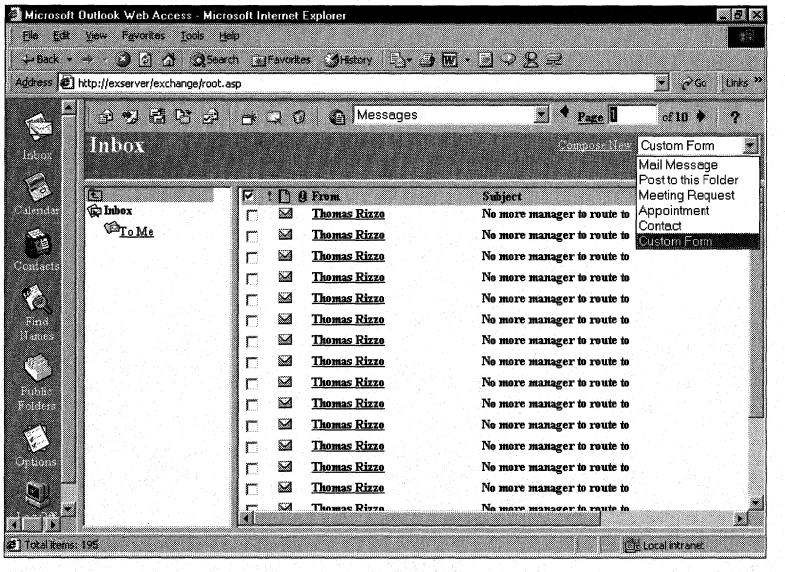


Figure 7-16. *Selecting Custom Form from the Compose New drop-down list in Outlook Web Access to view forms converted with the HTML Form Converter.*

After you select Custom Form and click the Compose New link, the Launch Custom Forms window is displayed, as shown in Figure 7-17. This window lists the custom forms in your Web forms library; the forms you converted using the Form Converter will be listed here. Click the link of the custom form that you want to test.

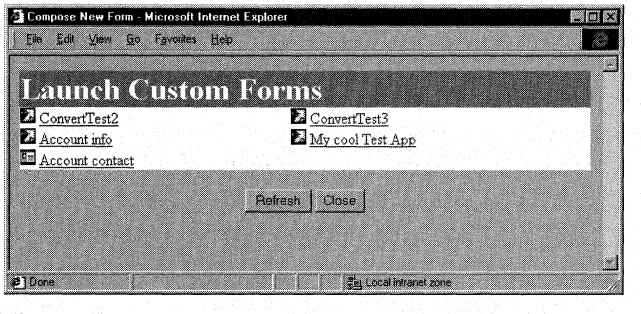


Figure 7-17. *The Launch Custom Forms window lists the forms in your Web forms library.*

Examples of Conversions

Now let's take a look at a few figures showing Outlook forms before and after being run through the Form Converter. Some of the figures illustrate the limitations mentioned earlier.

Figure 7-18 shows the converted Compose form for the Account Tracking application discussed in Chapter 6. The Account Tracking application was not designed with conversion to the Web in mind; therefore, a large amount of dynamic user-interface script was generated for the form. Typically, when forms use a lot of script to change user interface elements (for example, when you dynamically disable controls on your form based on the values a user types in another control), you will need to manually code many of the changes on the converted HTML form.

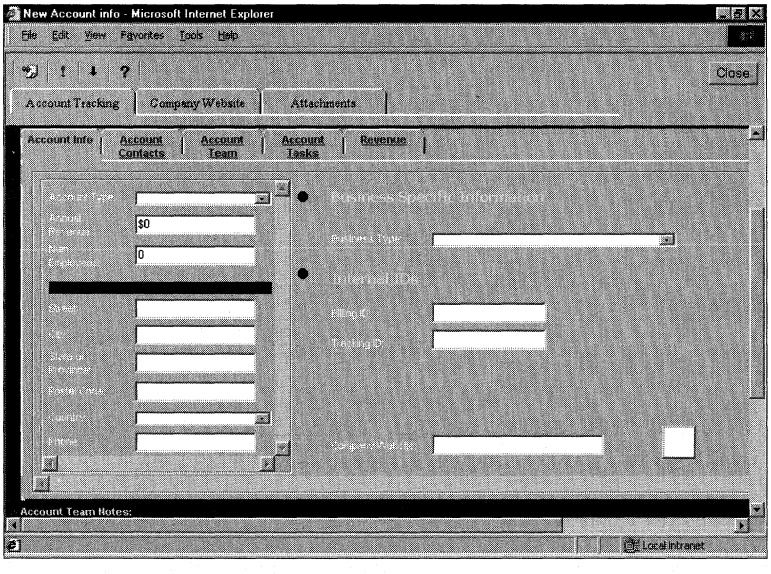


Figure 7-18. *The Compose form for the Account Tracking application. Notice how the picture does not come across on the CommandButton control.*

Figure 7-19 shows the Read form for the Account Tracking application. The form's custom actions display as buttons in the HTML version. Also notice that the Form Converter automatically converts the Read form for your Outlook application. The CDO rendering library, which you will learn about in Chapter 12, also automatically recognizes that the user is reading information for an application and thus launches the Read form when that user clicks on data in your application. All the binding of data in the form is generated by the Form Converter. You should be aware, however, that the HTML version of this form does not have the full functionality of the Outlook version. For example, the HTML version cannot display the embedded Tasks or Contacts for the account because these are populated by using VBScript code in the Outlook form.

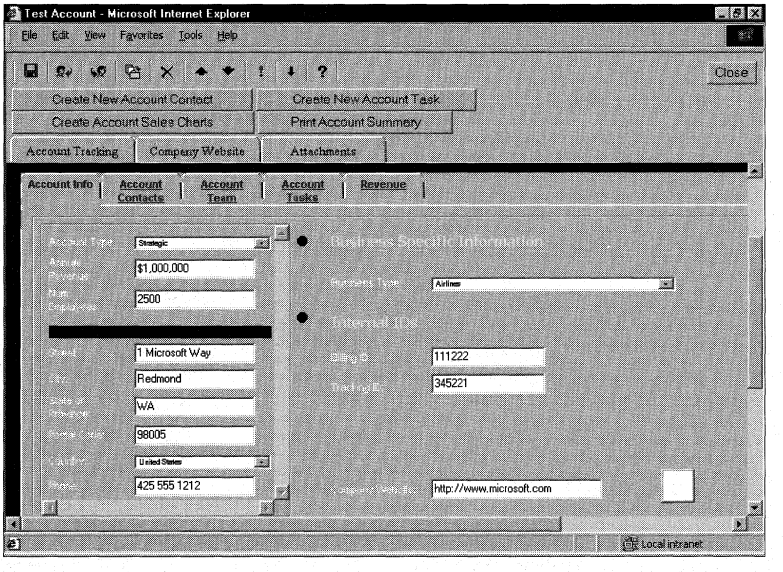


Figure 7-19. *The Read form for the Account Tracking application. Notice that the custom actions come across as buttons on the converted form.*

Figure 7-20 shows a helpdesk application before conversion to HTML, and Figure 7-21 shows the application after conversion. Notice the Opened By text box is automatically filled for the Outlook version but not filled for the Web version. However, the Web version does have text boxes where the default values are specified.

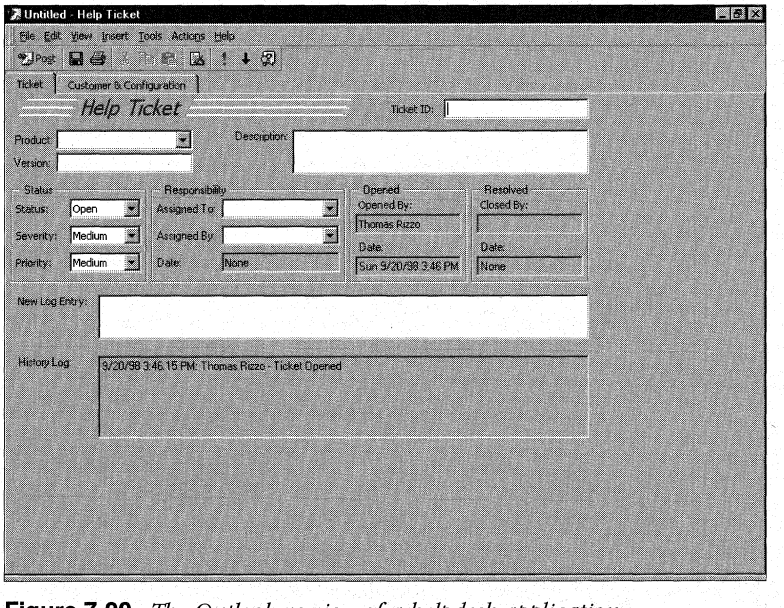


Figure 7-20. *The Outlook version of a helpdesk application.*

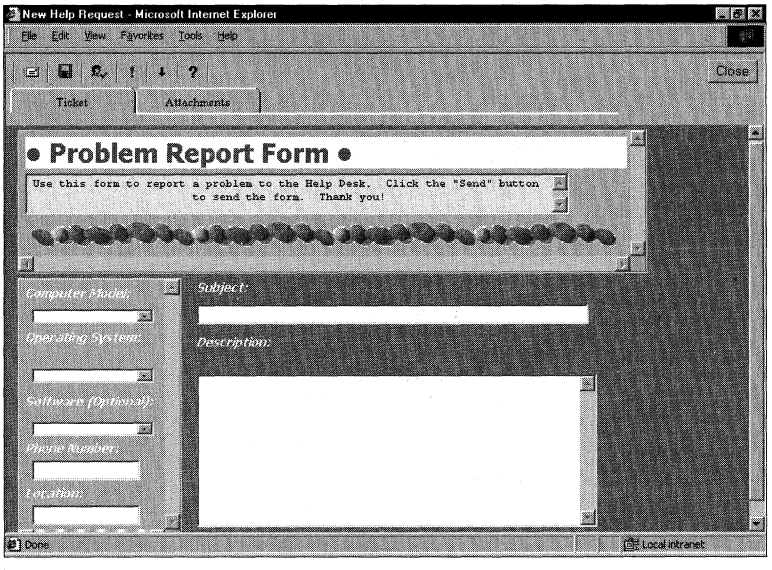


Figure 7-21. *The Web version of a helpdesk application.*

Files Created for Converted Forms

Forms converted by the Outlook HTML Form Converter consist of a number of files that you can customize. These files are placed in the Web forms library, which we will discuss in the next section. Let's look at the set of HTML and ASP files that comprise the core architecture of the Form Converter.

FrmRoot.asp

This file is the entry point of the converted or custom form. For the Form Converter, this file includes script and an HTML frameset that displays the other components of the form. This file is discussed more in the next section on the Web forms library.

Posttitl.asp

This file is the frame of the form, located near the top of the screen. This frame contains the toolbar and the tab strip. The main purpose of this file is to handle form commands generated by clicking the toolbar buttons and the tab strip.

Page_N.asp and Page_N-Read.asp

Every page in a converted form is represented by a separate .asp file. Certain pages of a form have predefined names—for example, the Options tab is named Options.asp. Custom pages of a form are assigned system-generated names, such as Page_3.asp. These custom form pages are generated by the Form Converter. If you created a separate compose and read layout for a particular form page, its file will have two different versions: Page_N.asp for composing the custom form, and Page_N-Read.asp for reading the custom form.

Commands.asp

Commands.asp is used to implement utility functions and event handlers for the converted form. Commands.asp is never seen by the user but rather is a hidden .asp file that is called to handle functions such as standard actions (*On_Send*, *On_Reply*, and so on) and also custom action event handlers.

Form.ini

The Form Converter automatically creates and publishes a Form.ini file for your application. This file contains the form's display name and code page, and indicates whether the form should be hidden in the Launch Custom Forms window. This file is discussed more in the next section on the Web forms library.

Web Forms Library

After the Outlook HTML Form Converter wizard is run and the necessary files are created, the files are copied to the Web forms library in Outlook Web Access. You might be wondering what the Web forms library is—well, it's nothing more than a folder, named Webdata, on the server where Outlook Web Access and some associated files were installed. CDO uses the Webdata folder to allow developers to publish custom forms for the Outlook Web Access client. By default, the Webdata folder is located at C:\exchsrvr\Webdata. Within the Webdata folder are additional subfolders that contain converted forms and other files.

If you browse the Webdata folder, you will see a folder structure similar to that shown in Figure 7-22. In the Webdata folder is a subfolder named for the language pack you installed for Outlook Web Access. Open it to find the Forms subfolder. This is where all the default forms for this particular Web server are stored.

All folders under the Forms subfolder correspond to custom message classes in Outlook. For example, if you have a custom form in Outlook with the message class IPM.Post.Project, the Form Converter creates a subfolder under the IPM\Post folder named Project, and copies all ASP and related files for that converted form into the Project subfolder. When a user views the custom forms in Outlook Web Access, she will see a new form named Project. A user can refresh the custom forms listing in the Launch Custom Forms window because the CDO object library allows this. Once the user clicks on the Project link in the Launch Custom Forms window, the converted Outlook form appears in her browser.

The Web forms library does not have to include converted Outlook forms only. You can place your own Web-based applications there. The applications you place in the library do not even have to be Exchange Server applications. You can add any application that can be used on a Web server. For example, you can place a SQL-based Web application in the Web forms library. This can make it easier for users because they can find the Web-based enterprise applications they need in a single location. Furthermore, since Outlook Web Access can authenticate users in

your ASP session, you can use this authentication in other Web-based applications stored in the Web forms library.

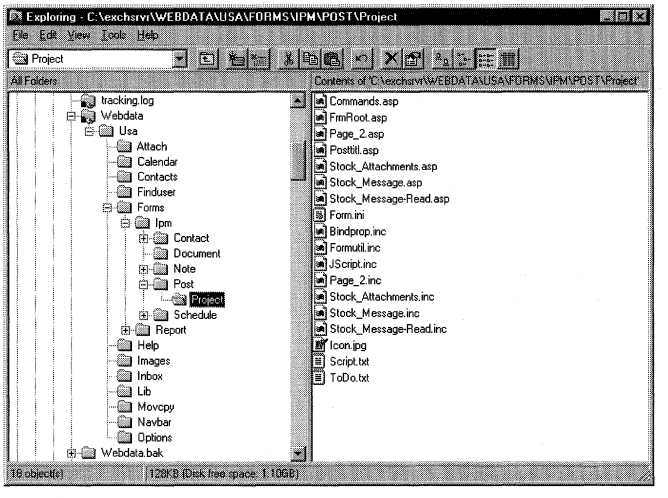


Figure 7-22. *The folder structure of the Web forms library. This structure is used by Outlook Web Access to show available Web applications to users.*

Adding Web-Based Applications to the Web Forms Library

To add a Web-based application to the Web forms library, you first need to create a subfolder for your application under the Forms folder. Then you need to specify certain names for your ASP files so that Outlook Web Access can recognize your application. You need to place three files in the form's subfolder: Form.ini, FrmRoot.asp, and Icon.jpg.

Form.ini

Form.ini includes the following three values:

- The friendly display name of a form, which avoids display of the message class name, such as IPM.Post.Project.
- The language code page of the form.
- Whether the form should be hidden in the Launch Custom Forms window. (Hide it if you want to make it available only as a response form.)

This is an example of a Form.ini file:

```
[Description]
DisplayName=My Custom Exchange and SQL Application
CodePage=1252
Hidden=0
```

Setting the Hidden variable to *0* makes a hyperlink to this application available in the Launch Custom Forms window. Setting Hidden to *1* hides the hyperlink. If your form is going to appear correctly when a user clicks on its name in the Launch Custom Forms window, you need to create a file named FrmRoot.asp in the form's subfolder. When the CDO Rendering Objects trace the Webdata folder tree, they recognize the Form.ini file and automatically add a hyperlink in the Launch Custom Forms window to the file FrmRoot.asp. If you do not create this file, your application name will appear as a hyperlink, but the link will be broken. In your FrmRoot.asp, you can redirect users to the Web server that contains your application or place your application code directly in the file. Include any necessary files in the subfolder you created for the application. If you place your application code in the same subfolder as Outlook Web Access, you can take advantage of the authentication and Exchange Server ASP Session objects that Outlook Web Access creates for you. If your application requires authentication and you redirect the browser to a separate Web location, you will have to create your own authentication code and CDO session for the Exchange server.

Icon.jpg

To make an icon appear to the left of the hyperlink for your Web-based application in the Launch Custom Forms window, you must create a JPEG file containing the image you want displayed, name it Icon.jpg, and place it in the form's subfolder. If you do not do this, the image will be displayed as missing.

Making HTML Forms Available in Outlook

You might want to make your converted forms and Web applications in the Web forms library available to users who are running Outlook 97, Outlook 98, Outlook 2000, Outlook for the Macintosh, or Outlook for Windows 3.1. To do this, you need to set some options in the Outlook client by following these steps:

1. In Outlook, from the Tools menu, select Options.
2. For Outlook 98 and Outlook 2000, click on the Other tab, click the Advanced Options button, click the Custom Forms button, and then click the Web Services button. This displays the Web Services dialog box. A configured Web Services dialog box for Outlook 2000 is displayed in Figure 7-23.

NOTE The version of Outlook that your users have determines the location of the Web Services button. Users can always find it, however, by clicking the Custom Forms button.

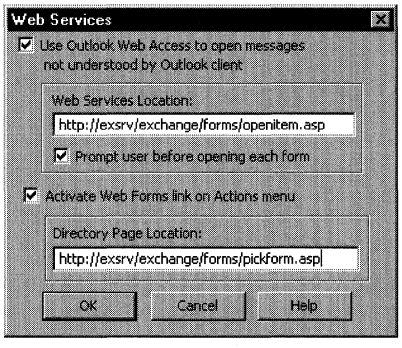


Figure 7-23. *A fully configured Web Services dialog box in Outlook 2000.*

3. Check the Use Outlook Web Access To Open Messages Not Understood By The Outlook Client check box.

4. In the Web Services Location text box, type the following URL, replacing *OWAServer* with the name of your Outlook Web Access server:

   *http://*OWAServer*/exchange/forms/openitem.asp*

   This option causes Outlook to search the Web forms library if it receives an item with a message class that does not have a corresponding Outlook form. Outlook looks in the Web forms library for a Web-based form that matches the message class.

5. If you want to prompt the user before Outlook opens the Web form, check the Prompt User Before Opening Each Form check box. If you enable this option, users will see a message similar to Figure 7-24.

Figure 7-24. *A prompt telling the user that Outlook is going to look for the custom form in the Web forms library.*

6. Check the Activate Web Forms Link On Actions Menu check box.

7. In the Directory Page Location text box, type the following URL, replacing *OWAServer* with the name of your Outlook Web Access server:

   *http://*OWAServer*/exchange/forms/pickform.asp*

   This adds the Web Form option under Actions menu to the Outlook client. If a user selects this option from the menu, Outlook will automatically launch the user's Web browser and display the Launch Custom Forms window. Users can then pick the desired Web-based form.

8. Click OK four times.

You should now have an option named Web Form on the Actions menu. If you select this option, the Launch Custom Forms window should appear when you log on.

Tips for Developing HTML-Ready Outlook Applications

This section discusses a few techniques that will simplify the process of converting your Outlook applications to HTML.

Align Your Controls on the Form

The Outlook HTML Form Converter uses HTML tables to position controls, so always use the layout capabilities of the Outlook design environment to line up your controls when you can. HTML tables are not as sophisticated as Outlook's layout tools, so relying on them for composition is risky. Using the layout tools helps the Form Converter figure out the best layout for your form in HTML, such as where the controls on your form should be relative to one another.

Avoid Calling the Outlook Object Library Interface Objects

The Outlook object library can be divided into an interface library with objects like the Explorer and the Inspector and a data access library with objects like MAPIFolder or Items. If you want to enable the cleanest conversion for your application, avoid dynamically changing the user interface using the Outlook object library because that code cannot be ported directly to HTML. If you require this, keep the changes to a minimum so that you can manually code them easily in the Web version of the application.

Avoid Overlapping Controls

Standard HTML today does not support overlapping controls on a Web page. If any of the controls overlap, the Form Converter will automatically move one of the controls in the HTML version. If your controls completely overlap, the Form Converter will probably skip converting one of the controls altogether because it won't be able to resolve which should be converted.

Avoid Using Images on CommandButtons

Since HTML does not support placing images on buttons, your images on CommandButtons will not be converted to HTML.

Do Not Customize the First Page of a Contact Form

The Form Converter works with Outlook 97, Outlook 98, and Outlook 2000 forms. Because Outlook 97 did not support first-page customization for a Contact form, the Form Converter will not convert a modified first page of a Contact form to HTML. The Form Converter ignores your modifications.

Do Not Save the Form Definition with the Item

For your form to retain its correct message class, do not save the form definition with the item. If you do save the form definition with the item and convert it, Web browser users will see only the form that corresponds to the default message class for the item, such as IPM.Note.

Avoid Using Unsupported Controls

If you can, avoid using the unsupported controls listed in the section "Features of the Converter," which appears earlier in the chapter. If you find that you do need to use them, you can instantiate them as ActiveX controls on an HTML form. You would then have to write the script to enable interaction between the ActiveX controls and the ASP application.

Set *AutoSize* to True with Image Controls

When working with Image controls, always set the *AutoSize* property to True under the Advanced Properties for the control. This will force the control to change size based on the size of the image and help avoid unintended sizing when the converter converts the form to HTML.

Chapter 8

Outlook 2000 Development Features

Microsoft Outlook 2000, released with Microsoft Office 2000, offers a slew of enhancements you can employ in your collaborative applications. These enhancements include COM add-in support, an enhanced object model, folder home pages, and Microsoft Visual Basic for Applications (VBA) support. In addition to the enhancements, Outlook 2000 provides backward compatibility with any of the solutions you developed in previous versions of Outlook. This means that all the techniques and code we've looked at for Outlook 98 apply for Outlook 2000. And because the forms environment in Outlook 2000 has not changed, you use the same tools for both Outlook 98 and Outlook 2000.

This chapter discusses COM add-ins and the changes to the Outlook object model. We will also look at how to write applications by using VBA in Outlook, and explore the trade-offs of developing a COM add-in vs. developing a VBA program directly inside Outlook. In Chapter 10, we'll update the Account Tracking application by adding a COM add-in as well as two folder home pages, which will illustrate how to take advantage of the new objects in the Outlook object model, use the Outlook View control, and write COM add-ins.

OFFICE 2000 COM ADD-INS

When developing Office solutions, you probably want to extend existing Office applications—Outlook in this case—with new functionality. With Outlook 98, you could add new forms to your application's Outlook environment, but you could not easily add new toolbars or program your application to respond to events beyond the form events, such as Item_Open or Item_Read. Plus, if you really wanted to extend beyond forms, you had to write an Exchange Client Extension. Exchange Client Extension development involved strict requirements and coding practices, and any extensions had to be written using C/C++. This meant that as a Microsoft Visual Basic or VBA developer, you were stuck either hacking a solution together or not enhancing the functionality at all.

Office 2000 includes support for COM add-ins. A COM add-in is a dynamic-link library (DLL) that can be used in an Office 2000 application. COM add-ins are used to include additional functionality in an Office application. As you can guess by the name, a COM add-in can be built using any COM development tool, such as Visual Basic, Microsoft Visual C++, or even Microsoft Visual J++. Since COM add-ins are compatible with all Office products, you can design a COM add-in once and reuse it in other Office products. For example, you could write a COM add-in that customizes the toolbars in your applications by using the CommandBar object model, which is shared across all of the Office products.

In this chapter, we'll look at a COM add-in that cannot be used across all the Office applications since it does call specific Outlook functionality. However, the concepts required to build this COM add-in can be applied to any add-in designed for other Office applications.

COM add-ins are registered specifically to be loaded by Office 2000 applications. Since COM add-ins are designed as DLLs, they will run in the same process as the host application. One benefit of an in-process add-in is that it has efficient access to the object model of the host application, allowing the add-in to quickly call methods and properties or to receive events from the host application. One potential caution to running an add-in in the same process space as the host is that you're in danger of slowing down or even crashing the host application. Keep this in mind during development.

Deciding Whether to Write a COM Add-In

You need to consider a number of issues when deciding whether to develop a COM add-in. Some of the functionality COM add-ins provide in Outlook is similar to other

Microsoft Exchange Server and Outlook development technologies, such as the Event Scripting Agent, which we'll discuss in Chapter 13. For this reason, I've provided three test questions to help you determine whether to create a COM add-in or use another technology.

First, do you need to receive events when the Outlook client is not running? The life span of your COM add-in is controlled by Outlook. When the Outlook process is running, your COM add-in can run and receive events. When Outlook is not running, your add-in is also not running. If you need to receive events when the Outlook client is not running, you might want to consider using the Event Scripting Agent; because your agent runs on the server, it will always receive events while the server is running. In Chapter 10, we'll examine a COM add-in that notifies users when an item in a folder changes—functionality that might be better implemented by using the Event Scripting Agent.

The second test question to answer is this: is performance a big concern for your application? If so, you should use an add-in because it is loaded in-process with Outlook, but be aware that you must use defensive coding practices to prevent crashing Outlook. Don't create an add-in that performs expensive lookups or data retrievals when starting, because Outlook will wait for it to finish before continuing.

Third, is your application event-driven? Outlook will fire a number of new events that your COM add-in can implement and handle. These new events allow you greater control over the Outlook user interface and Outlook data.

Developing a COM Add-In

If your application passes my three test questions, start developing your COM add-in! It is actually quite easy as Visual Basic has some features that can get you up and developing in a matter of minutes. In this section, we'll take a look at how to start developing COM add-ins, and then we'll review the new features of the Outlook object model that you can employ in your COM add-ins.

Before you can begin creating an add-in, you must start Visual Basic 5.0 or a later version and select an ActiveX DLL project. After the new project loads, you must select Microsoft Add-In Designer from the Project/References dialog box, as shown in Figure 8-1. This library contains the necessary interfaces for your COM add-ins.

In your Visual Basic code, you will need to type *Implements IDTExtensibility2* to see the *IDTExtensibility2* interface's events in the Procedure drop-down list in the Visual Basic code window. Figure 8-2 shows the code window with all of the *IDTExtensibility2* event procedures added.

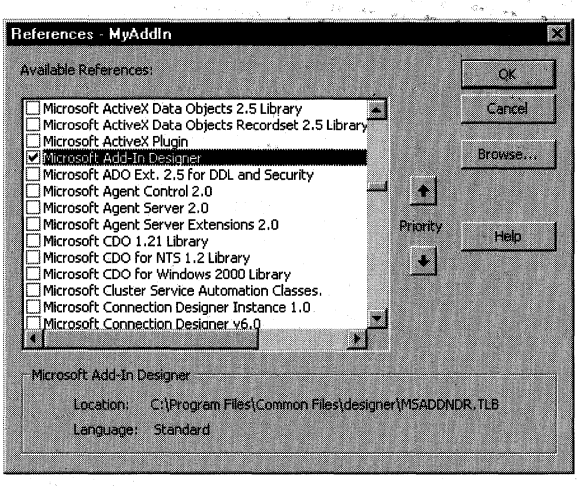


Figure 8-1. *Select Microsoft Add-In Designer from the Project/References dialog box.*

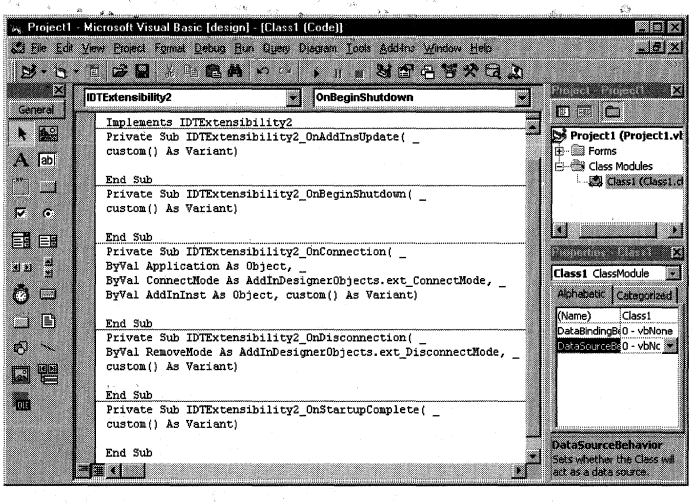


Figure 8-2. *The Visual Basic 6.0 code window with the five event procedures for the* IDTExtensibility2 *interface.*

The *IDTExtensibility2* Events

As you can see in Figure 8-2, *IDTExtensibility2* provides five events for you to use in your COM add-in: OnConnection, OnDisconnection, OnStartupComplete, OnBeginShutdown, and OnAddInsUpdate. Let's examine each of these events.

OnConnection event

The OnConnection event is called when your add-in is first loaded or connected to—for example, when Outlook starts or when the user selects to load your COM add-in. The user can select your add-in in the COM Add-Ins dialog box in Outlook 2000.

You can access this dialog box in Outlook by choosing Options from the Tools menu, clicking the Other tab, clicking the Advanced Options button, and clicking COM Add-Ins. The COM Add-Ins dialog box is shown in Figure 8-3.

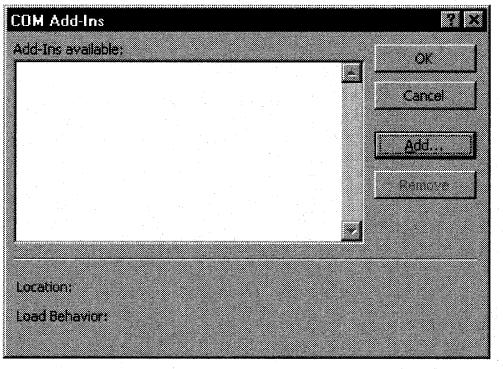


Figure 8-3. *The COM Add-Ins dialog box in Outlook 2000, where users can add or remove COM add-ins. Using the registry, you can force your add-ins to always load no matter what the user selects.*

The *OnConnection* event procedure is a great place to grab and store the Outlook Application object for use in your code later. When an OnConnection event occurs, the *OnConnection* event procedure is passed the following four parameters: *Application*, *ConnectMode*, *AddInInst*, and *Custom()*. The *Application* parameter is a reference to the Outlook Application object. The *ConnectMode* parameter describes the way in which the COM add-in was loaded. The *ConnectMode* parameter is a Long data type that can be set to one of the following four constants: ext_cm_AfterStartup, ext_cm_CommandLine, ext_cm_External, or ext_cm_Startup. The constants ext_cm_CommandLine and ext_cm_External do not apply to Office 2000 add-ins. The ext_cm_AfterStartup and ext_cm_Startup constants are subtly different from each other. The *ConnectMode* parameter is set to ext_cm_AfterStartup when the add-in is connected after Outlook starts or when the *Connect* property of the add-in is set to True. Usually, the *ConnectMode* parameter is set to ext_cm_AfterStartup when the user connects the add-in manually through the user interface. The *ConnectMode* parameter is set to ext_cm_Startup when your add-in is connected at the time Outlook starts up. The *AddInInst* parameter passes an object that refers to the current instance of your COM add-in. The *Custom()* parameter is an array of Variant data types that can hold user-defined data for your add-in. For Office 2000 add-ins, this parameter should be ignored.

OnDisconnection event

The OnDisconnection event occurs when your COM add-in is being disconnected from the application. The *OnDisconnection* event procedure is passed two parameters: *RemoveMode* and *Custom()*. The *RemoveMode* parameter, which is a Long data

type, specifies how your add-in was disconnected and can be set to these constants: ext_dm_HostShutdown or ext_dm_UserClosed. As you can guess by their names, ext_dm_HostShutdown indicates that the add-in is disconnected by the host shutting down, and ext_dm_UserClosed indicates either that a user is unchecking the add-in's check box in the COM Add-Ins dialog box or that the *Connect* property of the add-in is set to False.

The second parameter, *Custom()*, is an array of Variant data types that can hold user-defined data for your add-in. For Office 2000 add-ins, this parameter should be ignored.

Use the OnDisconnection event to restore any changes made to the application or to perform general cleanup for your application. Make sure you destroy any Inspector or Explorer objects that you create since Outlook will not properly close if any of these objects still exist.

OnStartupComplete event

In the case where a COM add-in connects at the time the host application is started, the OnStartupComplete event fires when the host has completed all of its startup routines. The OnStartupComplete event will not occur when a user selects to load the add-in from the COM Add-Ins dialog box after the application has already loaded. In that case, the OnConnection event will fire. The *OnStartupComplete* event procedure takes one parameter, *Custom()*, which you should ignore.

In this event procedure, place code that interacts with the application and should not be run until the application finishes loading. This event procedure is a good place to set some of your local and global variables to their corresponding Outlook objects. In the COM add-in example for Chapter 10, the *OnStartupComplete* event procedure searches the Outlook groups for a shortcut to the Account Tracking application and also has code to manipulate the command bars in the user interface.

OnBeginShutdown event

The OnBeginShutdown event is fired when the application is about to shut down and is called before the OnDisconnection event. Even after the OnBeginShutdown event fires, you still have full access to the Outlook object model, so you can save your settings to the registry or a file, or save any changes to your objects, before your objects are unloaded.

> **NOTE** If you are using Explorer or Inspector objects in your COM add-in, listen for the Close event on these objects. When your application receives this event, it should destroy all your open Explorer or Inspector objects because your Outlook COM add-in will not correctly shut down if any Explorer or Inspector objects are left open.

OnAddInsUpdate event

The OnAddInsUpdate event is fired whenever the list of COM add-ins is updated. When another add-in is connected or disconnected, this event occurs in any other

connected COM add-in. You can use this event to ensure that any other add-in upon which your add-in is dependent is connected. Once the dependent add-in is disconnected, you can disable your functionality or display a dialog box to warn the user to reconnect the other add-in. The *OnAddInsUpdate* event handler includes one parameter, *Custom()*, which your application should ignore.

Registry Settings for COM Add-Ins

Now that you know which events fire for add-ins, you need to know how to register and load the add-ins. Outlook decides which add-ins to load based on settings in the user's registry. If your add-in is not specified correctly in the registry, Outlook will not be able to load your add-in nor will your add-in appear in the COM Add-Ins dialog box.

Registering your add-in

For your add-in to work correctly, you must first compile and register the DLL that the add-in is based on. To do this, use the Regsvr32 command and specify the path to your DLL. This will register your DLL under the HKEY_CLASSES_ROOT subtree in the registry. If you are deploying your add-in to multiple machines, you will have to figure out how to install your DLL file on those machines. One way would be to use logon scripts to copy and register the DLL. Another way would be to deploy your add-in using either the Visual Basic deployment and setup tools or Microsoft Systems Management Server (SMS).

Once your COM add-in DLL is registered, you need to add some settings into the registry on the local machine. These settings include the add-in's name, description, target application, initial load behavior, and connection state.

Before writing this information to the registry, you must first decide how you want to deploy your add-in: you can either force all users to use your add-in or allow each user to decide whether he or she wants to load the add-in. The model you select determines where in the registry the information for your add-in has to be written. If you want to ensure the add-in is always loaded and that every user on a machine has access to it, you must register it under the key:

\HKLM\Software\Microsoft\Office<application>*\AddIns*

Then you must lock down the registry because the COM Add-Ins dialog box cannot unload add-ins registered there. If you want to give your users the option to specify whether they want the add-in loaded and to choose their own settings for the add-in, install your add-in under this key:

\HKCU\Software\Microsoft\Office<application>*\AddIns*

This location allows per-user settings for the add-in. An example of registering your add-in under this key is shown in Figure 8-4.

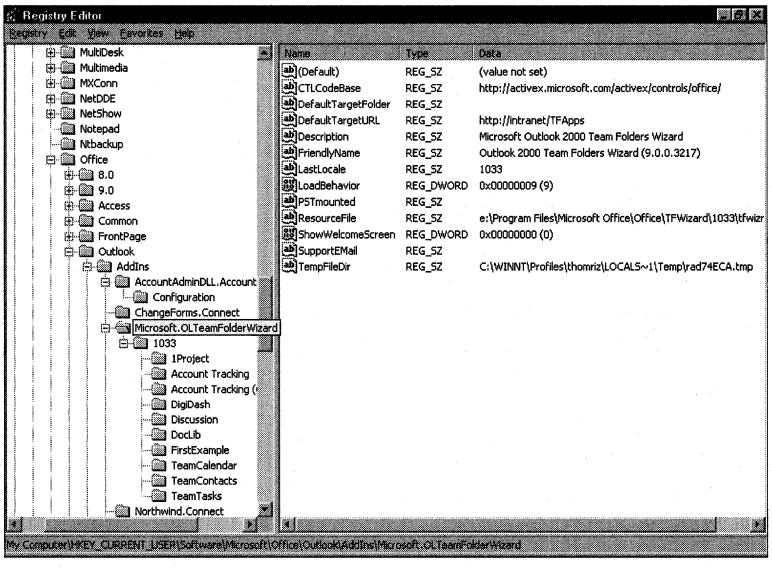


Figure 8-4. *This registry shows an add-in loaded under the key \HKCU\Software\Microsoft\Office\Outlook\AddIns. Registering your add-ins under this key will allow per-user settings.*

When you register your add-in under one of these registry keys, the information written to the key includes the following name/value pairs: *Description*, *FriendlyName*, and *LoadBehavior*. *Description* is a string type that provides a short description of the COM add-in. *FriendlyName* is a string that contains the name displayed in the COM Add-Ins dialog box. *LoadBehavior* is of type DWORD where the value is an integer that specifies how to load your COM add-in. This integer can have a value of *0* for Disconnected, *1* for Connected, *2* for load on startup, *8* for load on demand, or *16* for connect first time. You can combine these values to create different types of load sequences. For example, if you assign the value *3* to *LoadBehavior*, the add-in will be loaded on startup as well as connected. If you assign *9* to the add-in, the add-in will be connected and loaded when necessary, such as when the user clicks a button that uses code in the add-in.

The following code shows the content of a sample registry editor file (.reg) for a COM add-in:

```
REGEDIT4

[HKEY_CURRENT_USER\Software\Microsoft\Office\Outlook\
Addins\Sample.MyAddIn]
"FriendlyName"="My Sample Add-In"
"Description"="Sample Outlook COM Add-In"
"LoadBehavior"=dword:00000003
```

Trusting your COM add-ins

You can specify whether to trust all installed COM add-ins on a machine by setting the DWORD value *DontTrustInstalledFiles* under the following registry key:

```
\HKCU\Software\Microsoft\Office\9.0\Outlook\Security
```

By assigning *0* to *DontTrustInstalledFiles*, you are specifying that Outlook trust all installed add-ins. A value of *1* specifies to not trust all add-ins.

Debugging Your COM Add-In

Debugging your add-in using Visual Basic 6.0 is easy. All you need to do is write your add-in, register it, set some breakpoints on the code statements you are interested in, and then run the add-in in the Visual Basic 6.0 environment. In the Project Properties dialog box, shown in Figure 8-5, you can set some debugging options. You can specify whether you want to wait for the component to be created by the host application or you want Visual Basic to start an instance of the host application for you. Most times, I specify to wait for the components to be created by the host application. After Outlook starts and creates the COM add-in, the code in the add-in will execute and stop on encountered breakpoints. You can then step through your code in the Visual Basic Editor.

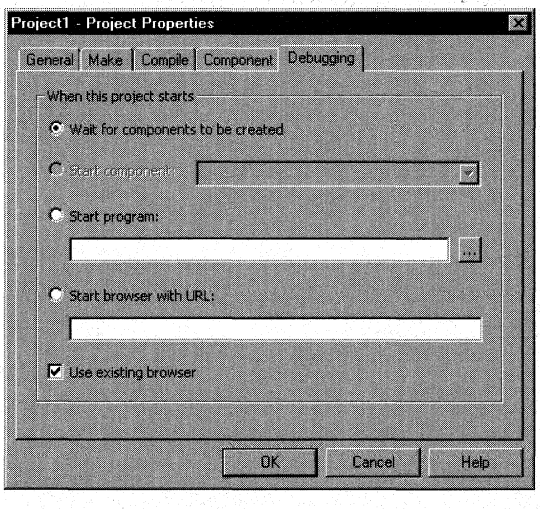


Figure 8-5. *The Debugging tab of the Project Properties dialog box in Visual Basic version 6.0. You can specify how you want Visual Basic to debug your ActiveX DLL.*

When debugging, be aware that message boxes in your add-in will appear in the Visual Basic development environment, not Outlook. If Outlook stops responding, you should switch to Visual Basic to see whether a message box is visible and waiting for you to respond.

WARNING One thing to watch out for in your COM add-ins is references to Inspector or Explorer objects in your code. If you do not properly destroy your variables, Outlook will exit but will stay in memory. Even if you set the variables holding references to these objects in your *OnBeginShutdown* procedure to Nothing, Outlook will still stay in memory. For this reason, both the Explorer and Inspector objects implement a Close event. You should add code to this event to destroy your references and check for any remaining Explorer or Inspector objects. If you find no Inspector objects and only one Explorer object, it's a sign that Outlook is properly shutting down.

Using COM Add-Ins from Custom Outlook Forms

There are many ways you can leverage COM add-ins from Outlook forms. One of the best ways is to use them to add functionality that might not be very easy to implement in Microsoft Visual Basic Scripting Edition (VBScript) or that might be very expensive to create. For example, you could create a CDO session in a COM add-in and then share that CDO session across multiple Outlook applications so that each application does not have to create and destroy a CDO session. By using VBScript, you can access the collections of COM add-ins available on your local machine and call public methods or set public properties on these add-ins. Since you can write add-ins in Visual Basic or Visual C++, you can implement advanced functionality that would be difficult to implement with VBScript. Furthermore, a COM add-in can provide a library of functions that you can reuse in all your custom Outlook forms.

Figure 8-6 shows a sample Outlook form that uses a COM add-in to launch other executable programs. Since VBScript doesn't support the *Shell* function, you can leverage the *Shell* function in your COM add-in.

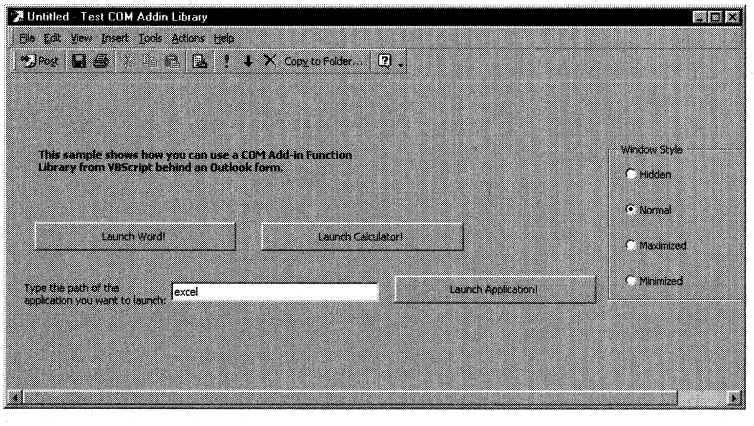


Figure 8-6. *A simple Outlook form that uses VBScript to leverage a COM add-in. From this form, you can launch any executable you need.*

Take a look at the following code from the Outlook form in Figure 8-6. Notice how you can access the COMAddins collection to retrieve a list of the COM add-ins

on your machine. You can then check whether the COM add-in you're interested in is loaded and connected in Outlook. To retrieve a COM add-in, use the *Item* method on the collection and either pass in the index of your add-in in the collection or pass a string that specifies the ProgID of the add-in. Notice how I use the *GetObject* method with the ProgID of the COM add-in. You would think that I could simply use the *Object* property of the COM add-in object that corresponds to my add-in. However, this technique didn't work on my test machines. If it works for you, by all means use this method. Otherwise, you should use the workaround in the code to make the COM add-in library work.

```
Dim oCOMAddinObj
Dim oCOMAddin

Sub cmdLaunchWord_Click
   Launch "winword"
End Sub

Sub cmdLaunchCalc_Click
   launch "calc"
end sub

Sub cmdLaunchApp_Click
   Launch item.userproperties.find("strAppPath").value
end sub

Function Item_Open()
   'On error resume next
   err.clear
   'Try to get a reference to the COM add-in
   set oCOMAddin = Application.COMAddIns.Item("OutlookHelper.Library")
   if err.number <> 0 then
      Msgbox "There was an error retrieving a reference to the COM " _
         & "Add-in Helper Library!  Closing form!"
      Item_Open = False
      exit function
   end if
   'Check to see whether the COM add-in is connected
   if oCOMAddin.Connect = False then
      msgbox "You must connect the COM Add-in before using this app!"
      Item_Open = False
      exit function
   end if
   'Get the real COM add-in object
   'This doesn't work in Outlook!
   set oCOMAddinObj = _
      Application.COMAddins.Item("OutlookHelper.Library").object
   'Workaround: use GetObject
   set oCOMAddinObj = GetObject("","OutlookHelper.Library")
End Function
```

(continued)

```
Sub Launch(strAppPath)
   'Get the Windows style
   iStyle = item.userproperties.find("strWindowsStyle").value
   iError = oCOMAddinObj.CustomShell(strAppPath, iStyle)
   if iError = 0 then
      msgbox "Error launching application!"
   end if
end Sub
```

In the next example, the add-in doesn't do much besides add a single public function named *CustomShell* that the user can call. This function leverages the *Shell* function in Visual Basic and allows you to shell out to another program. The function also provides a bit of error checking just in case some bogus values get past the Outlook test form. If the add-in successfully shelled out to executable, it returns the ID of the executable. If not, it returns zero.

```
Implements IDTExtensibility2
Dim oApp As Outlook.Application      'Global Outlook Application Object

Private Sub IDTExtensibility2_OnAddInsUpdate(custom() As Variant)

End Sub

Private Sub IDTExtensibility2_OnBeginShutdown(custom() As Variant)
   Set oApp = Nothing
End Sub

Private Sub IDTExtensibility2_OnConnection( _
   ByVal Application As Object, _
   ByVal ConnectMode As AddInDesignerObjects.ext_ConnectMode, _
   ByVal AddInInst As Object, custom() As Variant)
      Set oApp = Application
End Sub

Private Sub IDTExtensibility2_OnDisconnection( _
   ByVal RemoveMode As AddInDesignerObjects.ext_DisconnectMode, _
   custom() As Variant)

End Sub

Private Sub IDTExtensibility2_OnStartupComplete(custom() As Variant)

End Sub

Public Function CustomShell(strAppPath, iWindowStyle)
   If strAppPath = "" Then
      'Return back an error
      Err.Raise vbObjectError + 513, "Shell Function", "A blank " _
         & "pathname was passed!"
      Exit Function
   Else
      'Check iWindowStyle
```

```
      If CInt(iWindowStyle) < 0 Or CInt(iWindowStyle) > 6 Then
         'Make it normal with focus
         iWindowStlye = vbNormalFocus
      End If
      'Try to execute the command and return the value
      iReturn = Shell(strAppPath, CInt(iWindowStyle))
      CustomShell = iReturn
   End If
End Function
```

OUTLOOK 2000 OBJECT MODEL

To help you develop COM add-ins as well as other applications, the object model has been updated in Outlook 2000 with over 100 new methods and properties and a bunch of new events that your applications can hook into. Figure 8-7 shows the hierarchy for the Outlook 2000 object model. In the rest of this chapter, we'll look at some of the objects, methods, properties, and events, and I'll give you hints for using them in your own applications. For more information on the Outlook 2000 object model, consult the help file named vbaoutl9.chm on the companion CD.

NOTE The events discussed in this section are not available from VBScript behind your Outlook forms. You must either use VBA, Visual Basic, or Visual C++ to receive these events.

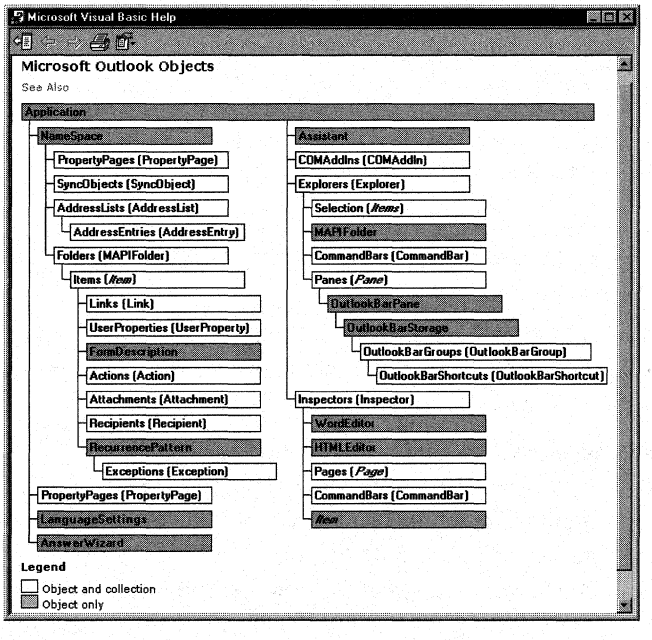


Figure 8-7. *The Outlook 2000 object model hierarchy.*

Objects and Collections

Outlook 2000 contains some collections and objects that consist of item types you can create, such as distribution lists, as well as user interfaces, such as custom property pages.

DistListItem Object

The DistListItem object represents a distribution list in a Contacts folder, which allows your users to do away with personal address books. The DistListItem object can hold multiple recipients both from the address list in Exchange Server as well as from one-off addresses.

You can use the *CreateItem* method on the Application object to create a new DistListItem object, or you can add the DistListItem object to a Contacts folder by using the *Add* method of the Items collection for the folder. The following code shows you how to use both methods to create a DistListItem object:

```
Dim oApp As Outlook.Application
Dim oNS As Outlook.NameSpace
Dim oExplorer As Outlook.Explorer
Dim oContact As Outlook.MAPIFolder
Dim oItems As Outlook.Items
Dim oDistList As Outlook.DistListItem
Dim oDistList2 As Outlook.DistListItem

Set oApp = CreateObject("Outlook.Application")
Set oNS = oApp.GetNamespace("MAPI")
Set oExplorer = oApp.ActiveExplorer
Set oContact = oNS.GetDefaultFolder(olFolderContacts)
Set oItems = oContact.Items
Set oDistList = oItems.Add(olDistributionListItem)
oDistList.DLName = "My new distribution list"
oDistList.Save
Set oDistList2 = oApp.CreateItem(olDistributionListItem)
oDistList2.DLName = "My other new distribution list"
oDistList2.Save
```

The DistListItem object inherits many of the methods and properties that other Outlook items also inherit, but it also has some unique methods and properties, which are described in the following sections.

Adding new members to the distribution list

To add new members to your DistListItem object, you use the *AddMembers* method. Before you call this method, however, you must create a new Recipients collection to hold the names you want to add to the distribution list. The easiest way to create the new Recipients collection is to create a new mail item and use the Recipients collection available on the mail item. Then you can populate the collection and create the new DistListItem object as shown here:

```
Dim oDistList As Outlook.DistListItem
Dim oTempMail As Outlook.MailItem
Dim oTempRecips As Outlook.Recipients

Set oDistList = oItems("My new distribution list")
Set oTempMail = oApp.CreateItem(olMailItem)
Set oTempRecips = oTempMail.Recipients
oTempRecips.Add "Thomas Rizzo"
oTempRecips.Add "Aaron Con"
oDistList.AddMembers oTempRecips
oDistList.Display
```

Removing members from a distribution list

To remove members from your DistListItem object, you use the *RemoveMembers* method. This method is similar to the *AddMembers* method in that you need to pass it to a valid Recipients object that contains the members you want to remove. The following code shows you how to use this method:

```
Dim oDistList As Outlook.DistListItem
Dim oTempMail As Outlook.MailItem
Dim oTempRecips As Outlook.Recipients

Set oDistList = oItems("My new distribution list")
Set oTempMail = oApp.CreateItem(olMailItem)
Set oTempRecips = oTempMail.Recipients
oTempRecips.Add "Thomas Rizzo"
oTempRecips.Add "Aaron Con"
oDistList.RemoveMembers oTempRecips
oDistList.Display
```

Retrieving the name of the distribution list

The DistListItem object contains a property named *DLName*. This property can be used to set or return the name of the distribution list. The following code finds all the distribution lists in your Contacts folder and returns their names in a message box:

```
Dim oItem As Outlook.DistListItem

RestrictString = "[Message Class] = 'IPM.DistList'"
Set oRestrictedItems = oItems.Restrict(RestrictString)
For Each oItem In oRestrictedItems
   strDLs = strDLs & vbLf & oItem.DLName
Next
MsgBox "You have " & oRestrictedItems.Count & " DL(s) in your " _
   & "contact folder.  Names: " & strDLs
```

Counting the number of users in a distribution list

Sometimes you'll want to know how many users are on a distribution list before you mail items to it. To retrieve the count for the number of users contained in a distribution list object, you must use the *MemberCount* property. Note that this count does

not include the member count for nested distribution lists in your original list. For example, if you have a distribution list with 20 members, and one of those members is a distribution list with 50 members, the *MemberCount* property will return 20, not 70. The following code finds all the distribution lists in your Contacts folder and returns a sum of all the *MemberCount* properties:

```
RestrictString = "[Message Class] = 'IPM.DistList'"
Set oRestrictedItems = oItems.Restrict(RestrictString)
For Each oItem In oRestrictedItems
   intCount = intCount + oItem.MemberCount
Next
MsgBox "Member count for all DLs is: " & intCount
```

SyncObject Object and SyncObjects Collection

Outlook 2000 allows users to set up quick synchronization folder groups. These synchronization groups, or profiles, allow users to configure different synchronization scenarios, such as which folders get synchronized offline and which filters apply to those folders. Users can then select the proper profile for their connection speed or synchronization preferences.

The Outlook SyncObjects collection contains all the synchronization profiles set up for the current user. Your program can start or stop any of these synchronization profiles using the methods of the SyncObject object. You can also monitor the progress of the synchronization by hooking into the events provided by the SyncObject object, named SyncStart, Progress, OnError, and SyncEnd. Let's look at how to use the SyncObjects collection and the SyncObject object.

Finding a SyncObject in the SyncObjects collection

The SyncObjects collection contains one property, named the *Count* property, and one method, named the *Item* method, that you can use to find out more information about the SyncObject objects contained in the collection. The *Item* method allows you to identify an object in the collection by specifying a numeric or named index. By using the *Item* method, you can quickly retrieve a SyncObject object. The *Count* property returns the number of SyncObject objects contained in the collection.

The next code example shows you how to use both the *Item* method and the *Count* property. First, the code finds a SyncObject object named *slow modem*, and then it displays the number of synchronization profiles the user currently has set up. Note that you cannot create or delete SyncObject objects programmatically. Only the user can do this through the Outlook user interface.

```
Dim oSyncObjects As Outlook.SyncObjects
Dim oSyncObject As Outlook.SyncObject

Set oSyncObjects = oNS.SyncObjects
Set oSyncObject = oSyncObjects("slow modem")
```

```
MsgBox "You have " & oSyncObjects.Count & " SyncObjects!"
strNames = vbLf
For Each oSyncObject In oSyncObjects
   strNames = strNames & vbLf & oSyncObject.Name
Next
MsgBox "Names: " & strNames
```

Starting and stopping synchronization

Once you find a SyncObject object, you might want to start or stop the synchronization process. You can do this by using the *Start* and *Stop* methods of the SyncObject object, as shown here:

```
Set oSyncObjects = oNS.SyncObjects
Set oSyncObject = oSyncObjects("slow modem")
oSyncObject.Start
```

Monitoring the progress of synchronization

The SyncObject object provides four events that your application can hook into to track the progress of a synchronization process: SyncStart, SyncEnd, Progress, and OnError. To implement these events, you must first declare a variable using the Dim statement in Visual Basic that defines a SyncObject object and uses the keyword WithEvents. The SyncStart event is fired when Outlook starts synchronizing using a particular SyncObject. The SyncEnd event is fired immediately after Outlook finishes synchronizing. The OnError event is fired when Outlook encounters an error when synchronizing. The *OnError* event procedure is passed both the error code and the description of the error as a string.

The final event, Progress, is fired periodically by Outlook to report on the progress of synchronization. Four parameters are passed to the *Progress* event procedure: *State*, *Description*, *Value*, and *Max*:

- *State* is a Long data type that indicates the status of the synchronization. State can have the value *olSyncStopped (0)* or *olSyncStarted (1)*.
- *Description* is a string that describes the current state of synchronization.
- *Value* is a variable of Long data type that indicates the current value in the synchronization process. *Value* can be the number of items synchronized in the folder.
- *Max* is a Long parameter that indicates the maximum value for the third parameter. The ratio of the third parameter (the value) to the fourth parameter (the maximum) represents the percent of the synchronization process that is complete.

The following code shows you how to use these events in your application. These events are not available from VBScript behind an Outlook form.

```
Private Sub oSyncObject_Progress( _
   ByVal State As Outlook.OlSyncState, _
   ByVal Description As String, ByVal Value As Long, _
   ByVal Max As Long)
      strPercent = Description & Str(State / Max * 100) & "% "
      MsgBox strPercent & vbLf & "State: " & State & vbLf & _
         "Max: " & Max
End Sub
```

Outlook Bar Object Model

One of the most significant enhancements to the Outlook 2000 object model is the Outlook Bar objects, which allow you to manipulate the Outlook Bar shortcuts as well as the user interface. With Outlook 2000, Outlook Bar shortcuts hold not only file and folder shortcuts but also URL shortcuts to Web pages, and you can customize Outlook to meet the needs of your applications. Let's take a look at the objects and collections for the Outlook Bar object model. Figure 8-8 shows how the objects in the Outlook Bar object model work together.

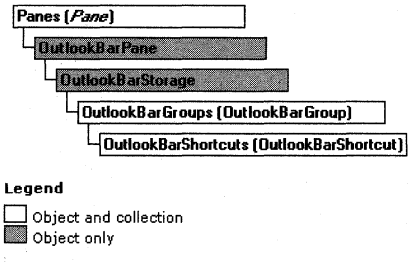


Figure 8-8. *The relationship between the objects and collections in the Outlook Bar object model.*

Panes Collection

The Panes collection enables developers to access the available Outlook application window panes. Although Outlook supports three panes—the OutlookBar, the FolderList, and the Preview panes—only the OutlookBar pane is accessible as an object in the Panes collection. If you try to access either of the other two panes, you will receive an error.

The Panes collection is retrieved from an Explorer object by using the new *Pane* property on that object. Once you retrieve the Panes collection, you can use the *Item* method of the Pane object and pass in either the numeric index or the name of the desired Pane object. To retrieve the OutlookBarPane object, you should pass in the text *OutlookBar* to the *Item* method.

The Panes collection also supports the *Count* property. Use this property to retrieve the number of Pane objects in the collection.

OutlookBarPane Object

After passing the text *OutlookBar* to the *Item* method of the Panes collection, Outlook returns an OutlookBarPane object. The OutlookBarPane object contains events and properties that let you control and monitor the Outlook Bar. These are the four properties you will use on the OutlookBarPane object:

- *Contents.* This read-only property returns the OutlookBarStorage object for the current OutlookBarPane object. From the returned object, you can retrieve the shortcuts and groups for the Outlook Bar.
- *CurrentGroup.* This property returns or sets the current group displayed in the Outlook Bar. You must pass a valid OutlookBarGroup object as the value for this property.
- *Name.* This read-only property returns a string that indicates the name of the current OutlookBarPane object.
- *Visible.* This property returns or sets the visibility of the OutlookBarPane object. *Visible* takes a Boolean value that specifies whether you want to show the Outlook Bar in the user interface.

The following code shows you how to use the OutlookBarPane object in your applications:

```
Dim oPanes As Outlook.Panes
Dim oOutlookBarPane As Outlook.OutlookBarPane

Set oPanes = oExplorer.Panes
Set oOutlookBarPane = oPanes("OutlookBar")
'Flip whether the pane is visible
oOutlookBarPane.Visible = Not (oOutlookBarPane.Visible)
```

The OutlookBarPane object also provides two events that you can capture when users work with the Outlook Bar: BeforeGroupSwitch and BeforeNavigate. The BeforeGroupSwitch event is fired whenever the user or object model attempts to switch to a new visible group. The *BeforeGroupSwitch* event procedure takes two parameters, *Group* and *Cancel*. If you set the *Cancel* parameter to True, the switch is canceled. The *Group* parameter is an OutlookBarGroup object containing the Outlook group that the user is trying to navigate to. The code on the following page shows you how to use BeforeGroupSwitch and cancel it when a user tries to navigate to a specific Outlook group.

```
Dim WithEvents oOutlookBarPane As Outlook.OutlookBarPane
Private Sub oOutlookBarPane_BeforeGroupSwitch( _
ByVal ToGroup As Outlook.OutlookBarGroup, Cancel As Boolean)
   If ToGroup.Name = "My Shortcuts" Then
      MsgBox "You cannot switch to the My Shortcuts group!"
      Cancel = True
   Else
      MsgBox "Now switching to the " & ToGroup.Name & " group."
   End If
End Sub
```

The BeforeNavigate event fires when the user attempts to click on an Outlook Bar shortcut. The *BeforeNavigate* event procedure takes two parameters, *Shortcut* and *Cancel*. *Shortcut* is an OutlookBarShortcut object, the Outlook Bar shortcut the user is trying to navigate to, and *Cancel* is a Boolean, which you can set to True to cancel the navigation. The following code example shows you how to use BeforeNavigate:

```
Dim WithEvents oOutlookBarPane As Outlook.OutlookBarPane
Private Sub oOutlookBarPane_BeforeNavigate( _
   ByVal Shortcut As Outlook.OutlookBarShortcut, Cancel As Boolean)
On Error Resume Next
'Need to watch out for file shortcuts!
Err.Clear
Set oTempFolder = Shortcut.Target
strName = oTempFolder.Name
If Err.Number = 0 Then
   If strName = "Inbox" Then
      MsgBox "Sorry, you can't switch to your Inbox."
      Cancel = True
   Else
      MsgBox "Now switching to the " & Shortcut.Name & " shortcut."
   End If
End If
End Sub
```

OutlookBarStorage Object

The OutlookBarStorage object is a container for the objects in an OutlookBarPane object. This object contains only one property—the *Groups* property—which you will use in your applications. The *Groups* property returns an OutlookBarGroups collection, which enables you to scroll through the groups on the Outlook Bar. The following code shows you how to use the *Groups* property to retrieve the OutlookBarGroups collection and then scroll through each group in the collection:

```
Dim oOutlookBarStorage As Outlook.OutlookBarStorage
Dim oOutlookBarGroups As Outlook.OutlookBarGroups
Dim oOutlookBarGroup As Outlook.OutlookBarGroup
```

```
Set oPanes = oExplorer.Panes
Set oOutlookBarPane = oPanes("OutlookBar")
Set oOutlookBarStorage = oOutlookBarPane.Contents
Set oOutlookBarGroups = oOutlookBarStorage.Groups
strGroups = vbLf
For Each oOutlookBarGroup In oOutlookBarGroups
   strGroups = strGroups & vbLf & oOutlookBarGroup.Name
Next
MsgBox "The names of the groups on your Outlook Bar: " _
   & strGroups
```

OutlookBarGroups Collection

The OutlookBarGroups collection contains OutlookBarGroup objects that represent all the Outlook groups on your Outlook Bar. Use this collection to count and add new groups to the Outlook Bar. This collection supports one property, *Count*, which you can use to retrieve the number of groups in the collection, as shown in the following code:

```
Set oPanes = oExplorer.Panes
Set oOutlookBarPane = oPanes("OutlookBar")
Set oOutlookBarStorage = oOutlookBarPane.Contents
Set oOutlookBarGroups = oOutlookBarStorage.Groups
MsgBox "The number of Outlook groups on your Outlook Bar is: " _
   & oOutlookBarGroups.Count
```

This collection also supports three methods—*Add*, *Item*, and *Remove*. The *Add* method adds a new, empty OutlookBarGroup object to the collection and returns a reference to this new OutlookBarGroup object. The *Add* method takes two parameters: one is a string that specifies the name of the group to add; the other is optional and specifies a number indicating the insertion position for the new Outlook group. The top of the bar is at position 1.

The *Item* method allows you to retrieve an OutlookBarGroup object by name or by index. The *Remove* method allows you to delete an OutlookBarGroup object by specifying the index of the object you want to remove.

The following example uses all three of these methods together. It creates a new OutlookBarGroup object, finds the object by using the *Item* method, and deletes the object by using the *Remove* method.

```
'Create the new group at the top of the bar
Set oNewOLBarGroup = oOutlookBarGroups.Add("My New Group", 1)
MsgBox "Added Group"
Set oTempOLBarGroup = oOutlookBarGroups("My New Group")
MsgBox "Got Group Named: " & oTempOLBarGroup.Name
'Since you have to remove a group by numeric index, we can loop
```

(continued)

```
'through the collection, find the OutlookBarGroup by name, and
'get the corresponding index
intCounter = 0
boolFound = 0
For Each oOutlookBarGroup In oOutlookBarGroups
   intCounter = intCounter + 1
   If oOutlookBarGroup.Name = "My New Group" Then
      boolFound = intCounter
   End If
Next
If boolFound <> 0 Then
   oOutlookBarGroups.Remove boolFound
   MsgBox "Deleted Group"
End If
```

The OutlookBarGroups collection is interesting because it supports three events that you can hook into: BeforeGroupAdd, BeforeGroupRemove, and GroupAdd. These three events enable you to trace when users try to add or remove certain Outlook groups and, if desired, cancel the user's action of removing or adding an Outlook group.

The BeforeGroupAdd event is fired before a new group is added to the Outlook Bar through either the user interface or code. The *BeforeGroupAdd* event procedure is passed a Boolean parameter named *Cancel* which, if set to True, will not be added to the new group by Outlook. The next code snippet shows you how to use the BeforeGroupAdd event to cancel a user's attempt to add an Outlook group. Note that because the group hasn't been created yet, a reference to the new group is not passed the *BeforeGroupAdd* event procedure, so you have no way of knowing which group the user is trying to add. However, since the GroupAdd event passes you the group the user added, you can write code in that event procedure to remove the group if the user is not allowed to add it.

```
Dim WithEvents oOutlookBarGroups As Outlook.OutlookBarGroups

Private Sub oOutlookBarGroups_BeforeGroupAdd(Cancel As Boolean)
   MsgBox "You are not allowed to add groups to your Outlook Bar!"
   Cancel = True
End Sub
```

The BeforeGroupRemove event is fired before a group is removed from an Outlook Bar. You can hook into this event with your custom applications to prevent users from deleting Outlook groups you created programmatically. The *BeforeGroupRemove* event procedure is passed two parameters. The first is an OutlookBarGroup object, which corresponds to the Outlook group that a program or user is trying to remove. The second is the *Cancel* Boolean parameter, which you can set to True to cancel the removal of the Outlook group. The following code checks

to see whether the user is trying to remove her Outlook Shortcuts group, and when it finds out that she is, the code cancels the action:

```
Private Sub oOutlookBarGroups_BeforeGroupRemove( _
ByVal Group As Outlook.OutlookBarGroup, Cancel As Boolean)
   If Group.Name = "Outlook Shortcuts" Then
      MsgBox "You cannot remove this group!"
      Cancel = True
   End If
End Sub
```

The GroupAdd event fires when a new group has been added successfully to the Outlook Bar. The *GroupAdd* event procedure is passed an OutlookBarGroup object, so you know which group has been added. If the user adds the new group using the Outlook user interface, the group will be named *New Group* because this is the default name for newly created Outlook groups. The following code displays a message box that shows the name of the Outlook group you added:

```
Private Sub oOutlookBarGroups_GroupAdd( _
ByVal NewGroup As Outlook.OutlookBarGroup)
   MsgBox "You added the " & NewGroup.Name & " group!"
End Sub
```

OutlookBarGroup Object

The OutlookBarGroup object represents an Outlook group on your Outlook Bar. The OutlookBarGroup object supports three properties and no methods. Use these three properties to access information about the Outlook group as well as shortcuts inside the Outlook group:

- *Name.* This property returns or sets the name of the OutlookBarGroup using a string.
- *Shortcuts.* This property returns the set of Outlook shortcuts contained in the group as an OutlookBarShortcuts collection.
- *ViewType.* This property returns or sets the way icons are displayed in the Outlook Bar. This property can be two values, either *olLargeIcon (1)* or *olSmallIcon (2).*

The following example shows you how to use all these properties. It loops through the OutlookBarGroups collection, and then retrieves each OutlookBarGroup object and displays information about it.

```
For Each oOutlookBarGroup In oOutlookBarGroups
   strName = oOutlookBarGroup.Name
   Set oOutlookBarShortcuts = oOutlookBarGroup.Shortcuts
```

(continued)

```
    intShortcutCount = oOutlookBarShortcuts.Count
    strNames = vbLf
    For Each oOutlookBarShortcut In oOutlookBarShortcuts
       strNames = strNames & vbLf & oOutlookBarShortcut.Name
    Next
    Select Case oOutlookBarGroup.ViewType
       Case olLargeIcon:
          strViewType = "Large Icons"
       Case olSmallIcon:
          strViewType = "Small Icons"
    End Select
    MsgBox "The following information is for the " & strName _
       & " group." & vbLf & "The ViewType is: " & strViewType _
       & vbLf & "The number of shortcuts in the group" _
       & " is: " & intShortcutCount & vbLf & _
       & "The shortcuts are named:" & strNames & vbLf
Next
```

OutlookBarShortcuts Collection

The OutlookBarShortcuts collection contains a set of OutlookBarShortcut objects and represents the shortcuts in an OutlookBarGroup object. This collection supports the *Count* property, which returns the number of OutlookBarShortcut objects in the collection.

This collection also supports three methods so that you can manipulate it: *Add*, *Item*, and *Remove*. The first method, *Add*, allows you to create a new shortcut in your Outlook group. The return value for *Add* is the new OutlookShortcut object you created. This method takes three parameters. The first parameter is a Variant that is the target for the shortcut. The target can be either a MAPIFolder object or a string that specifies a URL. Outlook 2000 supports placing URL shortcuts on your Outlook Bar. The second parameter is a string that specifies the name of the shortcut you are creating. The final parameter is an optional parameter that specifies the insertion position of the new shortcut. A value of *0* specifies to insert the shortcut at the top of the Outlook group. The following code example adds both a folder and a URL shortcut to a newly created Outlook group using the *Add* method:

```
'Create the new group at the top of the bar
Set oNewOLBarGroup = oOutlookBarGroups.Add("My New Group", 1)
'Get the shortcuts in the new group
Set oOutlookBarShortcuts = oNewOLBarGroup.Shortcuts
'Now add a shortcut that points to a folder
Set oFolder = oNS.GetDefaultFolder(olFolderInbox)
'Optionally, we can set a variable to retrieve the
'new shortcut. 0 at the end means add it to the
'top of the group.
Set oNewShortcut = oOutlookBarShortcuts.Add(oFolder, "My Inbox", 0)
```

```
'Now let's create a new shortcut to a Web page
strEXHTTP = "http://www.microsoft.com/exchange"
oOutlookBarShortcuts.Add strEXHTTP, "Exchange Web site"
```

The second method, *Item*, allows you to specify the index number or the name of the shortcut you want to retrieve from the collection. The third method, *Remove*, takes the index of the OutlookBarShortcut object you want to remove from the collection. The following code shows you how to find and remove all shortcuts that point to the Inbox:

```
On Error Resume Next
For Each oOutlookBarGroup In oOutlookBarGroups
   Set oOutlookShortcuts = oOutlookBarGroup.Shortcuts
   intCounter = 1
   For Each oOutlookShortcut In oOutlookShortcuts
      'Watch out for File System shortcuts
      Err.Clear
      If oOutlookShortcut.Target.Name = "Inbox" Then
         If Err.Number = 0 Then
            oOutlookShortcuts.Remove intCounter
         End If
      End If
      intCounter = intCounter + 1
   Next
Next
```

The OutlookBarShortcuts collection supports these events: BeforeShortcutAdd, BeforeShorcutRemove, and ShortcutAdd. The BeforeShortcutAdd event fires before a new shortcut is added to the Outlook Bar. The *BeforeShortcutAdd* event procedure is passed a parameter named *Cancel*, which you can set to *True* to cancel the attempted addition of the shortcut. The following code shows you how to hook into this event and cancel the action of a user trying to add a new shortcut to his Outlook Shortcuts group. Note that you are not passed the new Outlook shortcut object because the event is fired before the new shortcut is created.

```
Dim WithEvents oOutlookShortcuts As OutlookBarShortcuts

'The oOutlookShortcuts variable must be set before the event
'will fire
Set oOutlookBarGroup = oOutlookBarGroups("Outlook Shortcuts")
Set oOutlookShortcuts = oOutlookBarGroup.Shortcuts

Private Sub oOutlookShortcuts_BeforeShortcutAdd(Cancel As Boolean)
   On Error Resume Next
   MsgBox "You are not allowed to add shortcuts!"
   Cancel = True
End Sub
```

The second event supported by the OutlookBarShortcuts collection is the BeforeShortcutRemove event, which fires before a shortcut is removed from a group in the Outlook Bar. The *BeforeShortcutRemove* event procedure is passed two parameters: an OutlookBarShortcut object that corresponds to the shortcut the user or program is trying to remove; and *Cancel*, which is a Boolean parameter that you can set to True to cancel the removal. The following code shows you how to use this event to prevent a user from removing his Calendar shortcut from the Outlook Shortcuts group:

```
Dim WithEvents oOutlookShortcuts As OutlookBarShortcuts

'The oOutlookShortcuts variable must be set before the event
'will fire
Set oOutlookBarGroup = oOutlookBarGroups("Outlook Shortcuts")
Set oOutlookShortcuts = oOutlookBarGroup.Shortcuts

Private Sub oOutlookShortcuts_BeforeShortcutRemove( _
ByVal Shortcut As Outlook.OutlookBarShortcut, Cancel As Boolean)
   On Error Resume Next
   If Shortcut.Target.Name = "Calendar" Then
      MsgBox "You can't remove the shortcut to your calendar!"
      Cancel = True
   End If
End Sub
```

The third event supported by the OutlookBarShortcuts collection is ShortcutAdd. This event fires after a new Outlook shortcut has been added to the Outlook Bar. This event passes to you, as an OutlookBarShortcut object, the newly added shortcut. The following example shows you how to hook into this event:

```
Dim WithEvents oOutlookShortcuts As OutlookBarShortcuts
'The oOutlookShortcuts variable must be set before the event
'will fire
Set oOutlookBarGroup = oOutlookBarGroups("Outlook Shortcuts")
Set oOutlookShortcuts = oOutlookBarGroup.Shortcuts

Private Sub oOutlookShortcuts_ShortcutAdd( _
ByVal NewShortcut As Outlook.OutlookBarShortcut)
   MsgBox "You added the " & NewShortcut.Name & " shortcut!"
End Sub
```

OutlookBarShortcut Object

The OutlookBarShortcut object represents an Outlook shortcut on your Outlook Bar. Use this object to inquire about the target a shortcut points to. You make the inquiry by using the *Target* property. The *Target* property returns a Variant object; the data type of this Variant is determined by the target for the shortcut. If the shortcut points to a folder, the data type is MAPIFolder. If the shortcut points to a file system folder, the data type is Object. If the shortcut points to a URL or a file system path, the data

type is String. You can see how to use this object in the Account Tracking application we examine in Chapter 10.

Selection Collection Object

To provide your applications with the ability to identify what the user has selected in an Explorer window in Outlook, the Outlook object model has added a Selection collection object. The Selection object contains the set of items that a user has selected in the user interface. For example, you could use this collection to validate that the user has selected the proper item to enable your application to continue. You could also use it to dynamically add menu options and toolbar buttons when a user selects a certain item.

You must use the *Item* method with the object's index as a parameter to retrieve a specific object in the collection. The object is returned to you as an Object variable, so if you want to call a specific method or property on the object, you should coerce the object to a specific data type first.

This collection supports the *Count* property, which returns the number of items selected in a collection. You could use the *Count* property to determine the number of items the user has currently selected.

> **NOTE** To see the Selection collection in action, refer to the enhanced Account Tracking application in Chapter 10.

Explorers Collection

An Explorer object represents the window in which the contents of a folder are displayed. To make it easier for you to access these Explorer windows in your application, Outlook provides an Explorers collection below the Application object. The Explorers collection contains all the Explorer objects in your application, even those that are not visible.

The Explorers collection also contains the *Count* property, which you can use to find out how many Explorer objects are in the collection. The *Count* property could be used to identify any open Explorers that need to be closed before exiting your application. This is important because Outlook cannot terminate properly when Explorers are running.

The Explorers collection contains two methods, *Add* and *Item*. By using the *Add* method, you can create a new Explorer object and specify the folder to display in that Explorer's window. The *Add* method takes two parameters. The first parameter is either a MAPIFolder object or a string containing a path to the folder. The second parameter is an optional Long data type that specifies the display mode for the folder. This value can be *olFolderDisplayNormal (0)*, *olFolderDisplayFolderOnly (1)*, or *olFolderDisplayNoNavigation (2)*. The *Add* method returns the newly created Explorer object. This new Explorer object is initially hidden, and you must call the *Display* method to reveal it. The code on the following page shows you how to use the *Add* method.

```
Dim oExplorers As Outlook.Explorers
Dim oExplorer As Outlook.Explorer

Set oFolder = oNS.GetDefaultFolder(olFolderContacts)
Set oExplorers = oApp.Explorers
Set oExplorer = oExplorers.Add(oFolder, olDisplayNormal)
oExplorer.Display
```

The *Item* method of the Explorers collection is used to access an individual Explorer object in the collection by passing the object's index. The following example shows you how to do this:

```
Set oFolder = oNS.GetDefaultFolder(olFolderContacts)
Set oExplorers = oApp.Explorers
Set oExplorer = oExplorers.Item(1)
MsgBox oExplorer.Caption
```

The Explorers collection includes a single event, NewExplorer, which you use for tracking a newly created Explorer object that has not been made visible yet. This event passes you an Explorer object that is being opened or created. The following code shows you how to use the NewExplorer event:

```
Dim WithEvents oExplorers As Outlook.Explorers

Set oExplorers = oApp.Explorers
Private Sub oExplorers_NewExplorer( _
ByVal Explorer As Outlook.Explorer)
   MsgBox "You opened or added a new Explorer with the caption: " _
      & Explorer.Caption
End Sub
```

Inspectors Collection

An Inspector object represents the window in which an Outlook item is displayed. To make it easier for you to find out which Inspector objects are available in your application, the Outlook object model added an Inspectors collection. This collection can contain Inspector objects that are not currently visible to the user as well as Inspector objects that are. The Inspectors collection is accessed from the Application object in Outlook.

The Inspectors collection contains one property, the *Count* property. This property returns to you the number of Inspector objects in the collection.

The Inspectors collection also contains two methods, *Add* and *Item*. The *Add* method takes one parameter, which is a valid Outlook Item object, to display in the new Inspector. This method returns an Inspector object. You must call the *Display* method on the returned Inspector object to display the item. The following code example shows you how to use this method:

```
Set oFolder = oNS.GetDefaultFolder(olFolderTasks)
Set oItem = oFolder.Items.GetFirst
Set oInspectors = oApp.Inspectors
Set oInspector = oInspectors.Add(oItem)
oInspector.Display
```

The *Item* method allows you to access an Inspector object in the collection. You must pass the numeric index of the Inspector object you want to retrieve.

Links Collection and Link Object

Because Outlook 98 journals are not supported in Exchange Public Folders, users are not allowed to share their journal information with others. To make group tracking activities possible, Outlook 2000 supports a feature named Activity Tracking. Activity Tracking is the ability to associate, with a contact, items and documents so that Outlook can search folders that you specify for any linked items. To enable Activity Tracking, open a contact and select the Activities tab. A sample Activities contact is shown in Figure 8-9.

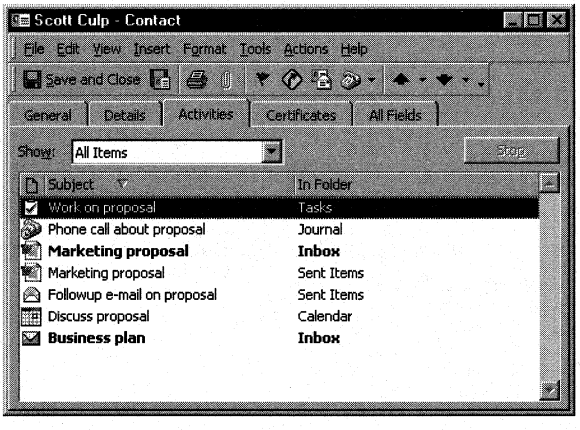


Figure 8-9. *The Activities tab for a specific contact. Outlook will find all linked items for a specified contact.*

Outlook can search both private and public folders. To specify which folders Outlook should search, right-click on the Contacts folder and select Properties. In the Contact Properties dialog box, click on the Activities tab. On this tab, you'll see a list of searchable folders you specified. You can also add new folders to the search criteria as shown in Figure 8-10.

Your ability to use Activity Tracking would be limited if the Outlook object model didn't support working with these links programmatically. For this reason, a Links collection and a Link object have been added to the Outlook object model.

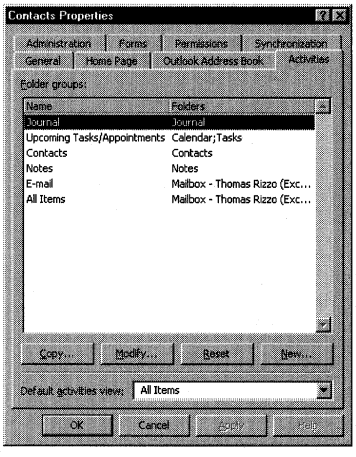


Figure 8-10. *The Activities tab for a folder. From here, you can add new folders for Outlook to search for linked items to the contacts in the folder.*

The Links collection contains a set of Link objects that comprise other items linked to a particular Outlook item. You can use the methods and properties of this collection to add, delete, or count the number of links to the particular item.

The Links collection contains three methods: *Add*, *Item*, and *Remove*. The first method, *Add*, creates a new Link object in the collection. You must pass to this method the object that you want to link to, and currently that object must be an Outlook Contact object. You would use the Links collection and the *Add* method, then, on the mail items, task items, and other types of Outlook items. The following example shows you how to use the *Add* method:

```
Dim oLinks As Outlook.Links
Dim oLink As Outlook.Link
Dim oContact As Outlook.ContactItem

Set oFolder = oNS.GetDefaultFolder(olFolderInbox)
Set oMailItem = oFolder.Items.Find("[Message Class] = 'IPM.Note'")
Set oLinks = oMailItem.Links
Set oContactFldr = oNS.GetDefaultFolder(olFolderContacts)
Set oItems = oContactFldr.Items
Set oContact = oItems.GetFirst

oLinks.Add oContact
MsgBox "Added a link to the " & oContact.FullName & " contact on " _
   & "the item " & oMailItem.Subject
```

The second method of the Links collection, the *Item* method, allows you to quickly retrieve an item in the collection by using either its index or its name. The following code shows you how to retrieve a Link object in the collection by using the name of the contact that the link refers to:

```
Set oFolder = oNS.GetDefaultFolder(olFolderInbox)
Set oMailItem = oFolder.Items.Find("[Message Class] = 'IPM.Note'")
Set oLinks = oMailItem.Links
Set oLink = oLinks.Item("Don Hall")
MsgBox "The link refers to the " & oLink.Name & " contact on " _
   & "the item " & oMailItem.Subject
```

The third method supported in the Links collection is *Remove*. This method allows you to remove a link from the collection by specifying the index of the Link object. The next bit of code shows you how to find and remove a specific Link object in the collection. Note that a user or application can associate an Outlook item with multiple contact items as links. This means that a single task could be linked to more than one contact.

```
Set oFolder = oNS.GetDefaultFolder(olFolderInbox)
Set oMailItem = oFolder.Items.Find("[Message Class] = 'IPM.Note'")
Set oLinks = oMailItem.Links
Counter = 1
For Each oLink In oLinks
   If oLink.Name = "Don Hall" Then
      oLinks.Remove Counter
      oMailItem.Save
   End If
   Counter = Counter + 1
Next

MsgBox "Removed the Don Hall link object."
```

The Link object contains properties but no methods. Of all the properties, you will probably use only three in your applications: *Item*, *Name*, and *Type*. The *Item* property returns the Outlook item that is represented by the Link object. For Outlook 2000, this property always returns an Outlook Contact item, which you can then manipulate in your code. The following example shows you how to use the *Item* property:

```
Set oFolder = oNS.GetDefaultFolder(olFolderInbox)
Set oMailItem = oFolder.Items.Find("[Message Class] = 'IPM.Note'")
Set oLinks = oMailItem.Links
Set oLink = oLinks.Item(1)
Set oContact = oLink.Item
MsgBox "The contact name is " & oContact.FullName
```

The *Name* property returns the name of the contact that the Link object is representing. This name is the display name for the contact. The *Type* property returns the Outlook item type represented by the Link object. As of publication, the only valid type is olContact.

PropertyPages Collection, PropertyPage Object, and PropertyPageSite Object

For information on the PropertyPages collection, the PropertyPage object, and the PropertyPageSite object, refer to the Account Tracking application enhancements in Chapter 10.

Methods, Properties, and Events for Existing Objects

Outlook 2000 adds more capabilities to existing objects in the Outlook object model. These enhancements include new methods and properties for the objects as well as a host of new events that your applications can use to receive notifications from Outlook.

Application Object

Recall that the Application object is the topmost object in the Outlook object model, so you must create an Application object before you create any other objects. Let's take a look at some of the methods, properties, and events of the Application object.

ActiveWindow method

The *ActiveWindow* method returns the object that represents the topmost Outlook window on the desktop. The return type for this object can be either an Explorer object or an Inspector object. If there is no currently open Explorer or Inspector object, this method returns nothing. Use this method to determine which object the current user is viewing and, if necessary, change the state of that object. The following code shows you how to use the *ActiveWindow* method:

```
Set oWindow = oApp.ActiveWindow
If Not (oWindow Is Nothing) Then
   If oWindow.Class = olExplorer Then
      strTop = "Explorer"
   ElseIf oWindow.Class = olInspector Then
      strTop = "Inspector"
   End If
   MsgBox "The topmost object is a(n) " & strTop & " object."
End If
```

AnswerWizard property

The *AnswerWizard* property returns an AnswerWizard object for the application. If you want more information on the AnswerWizard object model, see the Office 2000 documentation.

COMAddIns property

The *COMAddIns* property returns a COMAddIns collection that represents all COM add-ins currently loaded and connected in Outlook. You can use this collection to quickly access COM add-ins and their exposed objects. The following example shows you how to use the *COMAddIns* property:

```
Dim oCOMAddins As Office.COMAddIns
Dim oCOMAddin As Office.COMAddIn

Set oCOMAddins = oApp.COMAddIns
strName = vbLf
For Each oCOMAddin In oCOMAddins
   strName = strName & vbLf & oCOMAddin.ProgId
Next
MsgBox "The COM Add-Ins ProgIDs in Outlook are: " _
   & strNameExplorers and Inspectors properties
```

Explorers and *Inspectors* properties

The *Explorers* property returns the Explorers collection. The *Inspectors* property returns the Inspectors collection. For more information on these collections, see the "Objects and Collections" section earlier in this chapter.

LanguageSettings property

The *LanguageSettings* property returns a LanguageSettings object that you can use to retrieve language-related information about Outlook. For example, you can retrieve the install language, the user interface language, and the help language for Outlook. For COM add-ins, you can use this information to load the proper resource string for a user interface according to the language of the user.

ProductCode property

The *ProductCode* property returns a string that is the globally unique identifier (GUID) for the Outlook product. If you need to identify Outlook in your COM add-ins or applications, this GUID can be used to identify it.

ItemSend event

The ItemSend event fires whenever an attempt is made to send an item by using Outlook. This event returns an object, which is the item the user or application is trying to send, and a Boolean named *Cancel*. If you set *Cancel* to True, Outlook will stop the send action and leave the Inspector open for the user. If you do cancel the send, you should display an explanation in a message box so that users know what they need to add or delete to successfully send the item. The following code checks to see whether a user added a subject and a category to his message before he is able to send the item. Note that the ItemSend event does not fire when a user posts an item to a folder. In this case, you should monitor the folder for the ItemAdd event.

```
Dim WithEvents oApp As Outlook.Application

Private Sub oApp_ItemSend(ByVal Item As Object, Cancel As Boolean)
   If Item.Subject = "" Then
      MsgBox "You must add a subject!"
      Cancel = True
   ElseIf Item.Categories = "" Then
```

(continued)

```
      MsgBox "You must have a category!"
      Cancel = True
   End If
End Sub
```

NewMail event

The NewMail event fires when a new item is received in the Inbox of the current user. This event does not pass any parameters. Note that there is no one-to-one correspondence between the number of arriving messages and the number of times this event fires; the Inbox could receive many new messages but Outlook might fire this event only once. The following code shows you how to use the NewMail event:

```
Private Sub oApp_NewMail()
   MsgBox "You have received new mail!"
End Sub
```

OptionsPagesAdd event

For information on the OptionsPagesAdd event, see the Account Tracking application in Chapter 10.

Quit event

The Quit event is fired when Outlook begins to close. Using this event, you can persist any settings or other information as well as destroy any objects that are left open by your application.

Reminder event

The Reminder event is fired immediately before a reminder is displayed. This event passes one parameter, which is an object that corresponds to the item firing the reminder. You cannot cancel this event, so you are notified only that a reminder is going to appear. The following example shows you how to use the Reminder event:

```
Private Sub oApp_Reminder(ByVal Item As Object)
   MsgBox "The following item " & Item.Subject & " has " _
      & "fired a reminder."
End Sub
```

Startup event

The Startup event is fired after Outlook and any of its COM add-ins have been loaded. You can use this event to initialize VBA programs you created with Outlook.

NameSpace Object

The NameSpace object has been enhanced incrementally in Outlook 2000. The main enhancements you are mostly likely to use in your applications are the ability to dynamically add a .pst file to the NameSpace object, and the ability to create custom property pages for folders. Let's examine more closely the additions to the NameSpace object.

SyncObjects property

The *SyncObjects* property returns the SyncObjects collection for the NameSpace object. For more information on the SyncObjects collection, see the "Object and Collections" section earlier in the chapter.

AddStore method

The *AddStore* method allows you to dynamically connect an existing .pst file to Outlook and to create a new .pst file. This method takes one parameter, which is a path to the .pst file you want to access or create. If you pass in a path for the .pst file and the .pst file doesn't exist, Outlook will create the file. You can then retrieve the information in the .pst file by using the Outlook object model. The following example shows you how to use the *AddStore* method to access an existing .pst file on the hard drive:

```
oNS.AddStore "c:\my new store.pst"

'Retrieve a folder from the newly connected store
Set oFolder = oNS.Folders("Personal Folders").Folders("My Folder")

'Display the folder
oFolder.Display
```

OptionsPagesAdd event

For more information on this event, please see the enhanced Account Tracking application in Chapter 10.

Explorer Object

One of the most frequent requests by Outlook developers is to have more granular control over the way Explorers and Inspectors are graphically displayed on the screen. With Outlook 2000, you can not only control the location of your Explorer and Inspector windows but also receive events from these objects that describe what the user is doing with the user interface. Let's take a look at the additions to the Explorer object in Outlook 2000.

Caption property

The *Caption* property returns the string for the Explorer window text. This property is read-only.

CurrentView property

The *CurrentView* property returns or sets the view for the Explorer. When you set this property, you will cause the BeforeViewSwitch and ViewSwitch events to fire on the Explorer object. Since Outlook supports only a single initial view of a folder, you can use this property to customize per-user settings for the initial view. To do this, use the FolderSwitch event for the Explorer object. When this event fires, check the current folder and current user. Based on the current folder and current user, set the

CurrentView property appropriately. The following code snippet shows an example of this functionality:

```
Dim WithEvents oExplorer As Outlook.Explorer
Private Sub oExplorer_FolderSwitch()
On Error Resume Next
   If oExplorer.CurrentFolder.Class = olFolder Then
      If oExplorer.CurrentFolder.Name = "Contacts" Then
         If oNS.CurrentUser.Name = "Thomas Rizzo" Then
            oExplorer.CurrentView = "By Category"
         End If
      End If
   End If
End Sub
```

Height property

The *Height* property returns or sets the height of the Explorer window in pixels. You can use this property to dynamically change the height of your Explorer window.

Left property

The *Left* property returns or sets the distance from the left edge of the screen to the left edge of the Explorer window in pixels.

Panes property

The *Panes* property returns the Panes collection for the Explorer object. For more information on the Panes collection, see the "Objects and Collections" section earlier in the chapter.

Selection property

The *Selection* property returns a Selection collection, enabling you to access the items currently selected by the user. For more information on the Selection collection, see the "Objects and Collections" section earlier in the chapter.

Top property

The *Top* property returns or sets the distance, in pixels, from the top edge of the screen to the top edge of the Explorer window.

Width property

The *Width* property returns or sets the width of the Explorer window in pixels. The next code sample uses the *Top*, *Width*, *Left*, and *Height* properties to move an Explorer window around the screen. Notice how the code first sets the *WindowState* property to *olNormalWindow*. Outlook will return an error if the window is already maximized or minimized when you try to set these properties.

```
oExplorer.WindowState = olNormalWindow
oExplorer.Top = 100
oExplorer.Width = 200
oExplorer.Left = 300
oExplorer.Height = 100
```

***WindowState* property**

The *WindowState* property returns or sets the window state. The possible values for this property include *olMaximized (1)*, *olMinimized (2)*, and *olNormalWindow (3)*.

***Activate* method**

The *Activate* method activates an Explorer object by bringing it to the foreground and giving it keyboard focus. You can use this method to highlight a specific Explorer or Inspector window for your application.

***IsPaneVisible* method**

The *IsPaneVisible* method returns a Boolean that specifies whether a particular pane is visible in the Explorer window. You pass in the desired pane as a parameter to this method. The possible values you can pass are *olOutlookBar (1)*, *olFolderList (2)*, and *olPreview (3)*. Use the *IsPaneVisible* method in conjunction with the *ShowPane* method, which is described next.

***ShowPane* method**

The *ShowPane* method either hides or displays a specific pane in your Explorer window. You must pass to this method the pane you are interested in as well as a Boolean parameter that is either set to True to display the pane or set to False to hide the pane. The following code example shows you how to use the *IsPaneVisible* method with the *ShowPane* method to hide and display panes in an Explorer window:

```
'Flip the settings that the user already has
boolFolderList = oExplorer.IsPaneVisible(olFolderList)
boolOutlookBar = oExplorer.IsPaneVisible(olOutlookBar)
boolPreviewPane = oExplorer.IsPaneVisible(olPreview)

oExplorer.ShowPane olFolderList, Not (boolFolderList)
oExplorer.ShowPane olOutlookBar, Not (boolOutlookBar)
oExplorer.ShowPane olPreview, Not (boolPreviewPane)
```

Activate event

The Activate event is fired when an Explorer or an Inspector window becomes the active window. You can use this event to see whether a specific Explorer or Inspector window is made the active window and then customize the toolbar for that window. The following code shows you how to use this event:

```
Dim WithEvents oExplorer as Outlook.Explorer

Private Sub oExplorer_Activate()
   MsgBox "This Explorer window has become active!"
End Sub
```

BeforeFolderSwitch event

The BeforeFolderSwitch event occurs before the Explorer navigates to the new folder. This event passes to you, as an object, the folder the user is trying to navigate to as

well as a Boolean parameter named *Cancel*. To keep the user on the current folder and prevent the user from navigating to the new folder, set *Cancel* to True. If the user navigates to a folder in the file system, the BeforeFolderSwitch event will not pass an object for that folder. For an example of this event in action, look at the Account Tracking application in Chapter 10.

BeforeViewSwitch event

The BeforeViewSwitch event fires when a user tries to switch views. This event passes the name of the view the user is trying to change to as well as the Boolean variable *Cancel*. To cancel the view change and maintain the user's current view, set the *Cancel* variable to True. The following code sample shows you how to use the BeforeViewSwitch event:

```
Private Sub oExplorer_BeforeViewSwitch( _
ByVal NewView As Variant, Cancel As Boolean)
   If NewView = "By Category" Then
      Cancel = True
   End If
End Sub
```

Deactivate event

This event is fired when the Explorer or Inspector window is no longer the active window. This event does not pass any parameters.

FolderSwitch event

This event fires after a user successfully switches folders. This event does not pass any parameters.

SelectionChange event

This event is fired after the user selects a different item in the current view. This event does not pass any parameters. The next code sample shows you how to use this event with the Selection collection:

```
Private Sub oExplorer_SelectionChange()
   On Error Resume Next
   Set oSelection = oExplorer.Selection
   strName = vbLf
   For Each oItem In oSelection
      strName = strName & vbLf & oItem.Subject
   Next
   MsgBox "New Selection: " & strName
End Sub
```

ViewSwitch event

This event fires when the user successfully changes the view in the Explorer window. This event does not pass any parameters.

Close event

This event fires when the Explorer window is being closed. Most likely, you will listen for this event only when developing COM add-ins that need to correctly destroy Explorer objects or the variables that reference them.

Inspector Object

Since the properties, methods, and events for the Inspector object we have seen are the same as their Explorer object counterparts, I'm not going to dive into the details of these. I have listed them, however, below. For more information about them, reference the descriptions of the additions to the Explorer object.

- *Caption* property
- *Height* property
- *Top* property
- *Width* property
- *WindowState* property
- *Activate* method
- Activate event
- Deactivate event
- Close event

Folders Collection

The Folders collection contains three new events that you can use in your applications: FolderAdd, FolderChange, and FolderRemove.

FolderAdd event

The FolderAdd event fires when a folder is added to the Folders collection. This event passes the added folder as a MAPIFolder object. You cannot cancel this event. You might want to hook into this event to prompt the user to add the folder to a specific group on the Outlook Bar. The following code uses the FolderAdd event:

```
Dim WithEvents oFolders As Outlook.Folders

Set oFolders = oNS.Folders("Mailbox - Thomas Rizzo").Folders

Private Sub oFolders_FolderAdd( _
ByVal Folder As Outlook.MAPIFolder)
   MsgBox "You have added the " & Folder.Name & " folder!"
End Sub
```

FolderChange event

The FolderChange event fires when something is changed in the specified Folders collection, such as deleting all the items in the folder. This event passes the changed folder as a MAPIFolder object, but it does not pass the actual folder property that was changed. You have to figure out which property was changed programmatically. You can't cancel this event. The code on the following page shows you how to use the FolderChange event.

```
Private Sub oFolders_FolderChange( _
ByVal Folder As Outlook.MAPIFolder)
   MsgBox "You changed the " & Folder.Name & " folder!"
End Sub
```

FolderRemove event

The FolderRemove event fires when a folder is removed from the collection. It does not pass any parameters, so if you need to know which folder was removed, your code has to figure this out. Outlook will only notify you that some folder was removed. You can't cancel the FolderRemove event.

MAPIFolder Object

The MAPIFolder object provides three interesting properties: *WebViewAllowNavigation*, *WebViewOn*, and *WebViewURL*. All three are described in Chapter 10 in the context of the Account Tracking application enhancements.

Items Collection

Recall that the Outlook Items collection is a collection of objects in a particular folder. The type of object the Items collection contains depends on the items in the folder. For example, in your Calendar folder, the Items collection will most likely contain AppointmentItem objects. The following sections highlight the enhancements to the Items.

ItemAdd event

The ItemAdd event fires when a new item is added to the folder. This event returns, as an object, the item added to the collection for the folder. Remember that before you attempt to call methods or properties on a returned object, you should check which type of object was returned. The type of object returned might not be the type you expected and could cause unwanted behavior in your application. For an example of how to use this event, see the Account Tracking application in Chapter 10.

ItemChange event

The ItemChange event is fired when an item in the collection is changed in any way. The event passes to you, as an object, the item that was changed; it does not pass you the changed property. This means that you use your code to determine what was changed on the item. For an example of using this event, see the Account Tracking application later in Chapter 10.

ItemRemove event

The ItemRemove event is fired when an item is removed or deleted from the collection. This event does not pass any parameters. This means that your code must figure out which items were deleted from the collection.

Characteristics of Item Types

The following section describes some characteristics of all the item types in Outlook, such as the PostItem, the MailItem, and the AppointmentItem objects. The events we'll discuss can be used with Visual Basic/VBA and from VBScript, so the examples are written using VBScript.

Links Property

The *Links* property returns the Links collection for the object. For more information on the Links collection, refer to the "Methods, Properties, and Events for Existing Objects" section earlier in this chapter.

AttachmentAdd Event

The AttachmentAdd event fires after an attachment has been added to an Outlook item. This event passes the attachment as an Attachment object. Once the object is passed, you can perform tasks, such as checking the attachment for viruses, before the user sends the item. You cannot, however, cancel or stop the user from adding an attachment to the item. The following VBScript example shows you how to use this event:

```
Sub Item_AttachmentAdd(ByVal NewAttachment)
   if NewAttachment.Type = 1 then
   item.save
   if Item.Size > 10000 then
      msgbox "Sending a message with an attachment this large" _
         & " may take a long time."
   end if
   end if
End Sub
```

AttachmentRead Event

The AttachmentRead event fires after an attachment has been opened for reading and passes the attachment as an Attachment object. You can use this event to perform certain actions when the user opens the attachment, such as making a backup copy of the attachment or marking this attachment as checked out for a document management solution. The following example shows you how to prompt the user to save changes when the user opens an attachment to read it:

```
Sub Item_AttachmentRead(ByVal ReadAttachment)
   if ReadAttachment.Type = 1 then
      msgbox "Make sure to save any changes that " _
         & "you make to the attachment."
   end if
End Sub
```

BeforeAttachmentSave Event

The BeforeAttachmentSave event fires before an attachment is saved with the item. Since this event is supported in both VBScript and Visual Basic/VBA, you must follow the appropriate syntax. When you use this event from VBScript behind your Outlook form, the BeforeAttachmentSave event passes the attachment that is trying to be saved as an Attachment object. If you're using Visual Basic or VBA, this event passes both the attachment as well as a Boolean parameter named *Cancel*. To abort the save, set *Cancel* to True. The following two code examples show you a VBScript version and a Visual Basic/VBA version of this event, respectively. Notice how the VBScript version is a function and the Visual Basic/VBA version is a subroutine.

```
Function Item_BeforeAttachmentSave(ByVal SaveAttachment)
   If SaveAttachment.Type = 1 then
      If SaveAttachment.FileName = "sales.mdb" Then
         MsgBox "You cannot save this file!"
         Item_BeforeAttachmentSave = False
      End If
   End if
End Function

Private Sub oMailItem_BeforeAttachmentSave( _
ByVal Attachment As Outlook.Attachment, Cancel As Boolean)
   If Attachment.Type = 1 then
      If Attachment.FileName = "sales.mdb" Then
         MsgBox "You cannot save this file!"
         Cancel = True
      End If
   End if
End Sub
```

BeforeCheckNames Event

The BeforeCheckNames event fires before Outlook starts resolving names for the recipients of an item. You can use this event to check the names of the recipients from another data source such as a database. This event, like the BeforeAttachmentSave event, has two different syntaxes depending on whether you are calling the event from VBScript or Visual Basic/VBA. In VBScript, this event is implemented as a function that you can cancel by setting the name of the function to False. In Visual Basic/VBA, you are passed a Boolean parameter named *Cancel*. To cancel the name resolution, you set *Cancel* to True. The two following examples show you the VBScript version and the Visual Basic/VBA version:

```
Function Item_BeforeCheckNames()
   'You can use this event to cancel Outlook's resolution
   'And put your own resolution in
```

```
   Item_BeforeCheckNames = False
End Function

Private Sub oMailItem_BeforeCheckNames(Cancel As Boolean)
   'You can use this event to cancel Outlook's resolution
   'And put your own resolution in
   Cancel = True
End Sub
```

VBA SUPPORT IN OUTLOOK 2000

Outlook 2000 supports VBA. Now some of you reading this book might be thinking that your Outlook forms already support VBA, but this is not the case. In fact, you still need to write VBScript behind your Outlook forms. The VBA support in Outlook provides a way to customize the Outlook environment using the Outlook object model and all the events just discussed without using a separate development tool such as Visual Basic.

VBA Architecture

When writing your VBA applications in Outlook, you must contain your code in a VBA project. Each project is associated with a particular user. This means that different users on the same machine can customize Outlook differently using VBA. These projects can contain code modules or user forms. (User forms are different from Outlook forms.) To share information among these VBA projects, you must export your file and have the receiving user import your file into her VBA project.

Creating a VBA Application

The first step in creating your VBA application is to launch the Visual Basic Editor in Outlook. You can find the Visual Basic Editor on the Tools menu, via the Macro option. The Visual Basic Editor is shown in Figure 8-11. Once you are in the editor, you can add class modules, code modules, and user forms, depending on the needs of your application. You can even write code that responds to Outlook events by declaring your variables using the WithEvents keyword.

The Outlook object model is automatically available to your VBA application. After you finish writing your macro in the editor, you can explicitly run it or create a button on your Outlook toolbar that runs the macro when clicked. Figure 8-12 shows a sample application that converts incoming mail to a specific message class using a VBA macro and the Outlook object model.

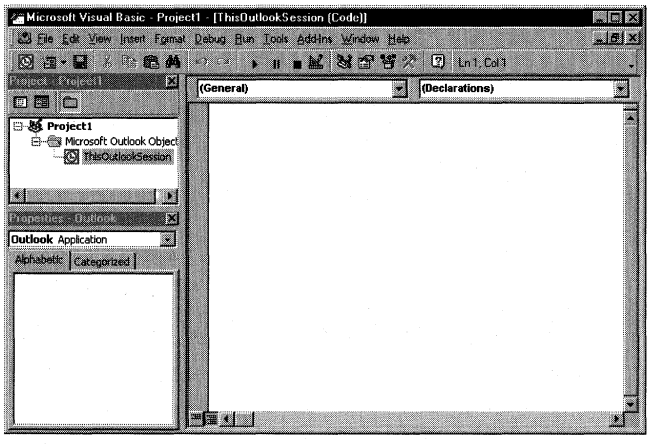


Figure 8-11. *The Visual Basic Editor in Outlook 2000. From here, you have the full power of VBA for your application.*

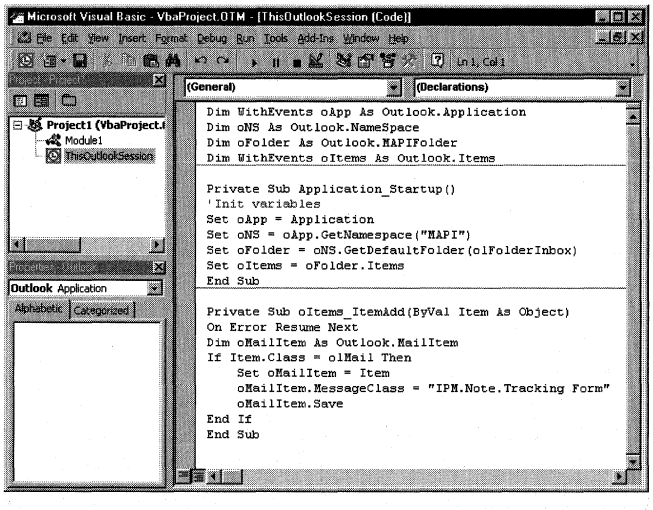


Figure 8-12. *The sample mail conversion code in the Visual Basic Editor.*

Choosing What to Write: COM Add-In or VBA Program?

By now you must be wondering whether you should write VBA programs or COM add-ins to customize your Outlook environment. While both technologies have their merits, I believe that if more than one user is going to run your program in an Outlook client, you should write a COM add-in. COM add-ins are easily distributed, and you can control a user's ability to run them.

If you want to customize only the Outlook client, writing a quick VBA program is easier than writing a full-blown COM add-in. To deploy your application in VBA, however, users must import the VBA file into their Outlook client, which is not the best deployment method. I predict that if you use COM add-ins many users will be installing your application.

Chapter 9

Outlook Team Folders Wizard

The Microsoft Outlook Team Folders Wizard, an Outlook 2000 add-in, shipped shortly after the release of Outlook 2000. The Outlook Team Folders Wizard provides a set of turnkey collaborative applications that take advantage of Outlook 2000 features and Microsoft Exchange Server public folders. Later in this chapter, we'll take a look at the Outlook 2000 development features that the Team Folders Wizard takes advantage of. First, we'll look at the features that the Team Folders Wizard provides for users who create collaborative applications.

FEATURES OF THE TEAM FOLDERS WIZARD

The Outlook Team Folders Wizard provides the following six applications, called folder templates, that users can quickly employ as a basis for their solutions. Users can use these templates directly from the Team Folders Wizard.

- Discussion Forum
- Document Library
- Team Calendar
- Team Contacts
- Team Project
- Team Tasks

The Team Project application consists of the other five templates. If you want to quickly learn what these five templates can do, install Team Folders, run Wizard setup, create a Team Project application, and scroll through its features.

Figure 9-1 shows how to select a template from the Team Folders Wizard. Once the user selects the template for her application, she can type the title of her application into the space provided. Now the interesting features of the Team Folders Wizard kick in. Namely, the user can select the Exchange Server location for her Team Folders application. The user will probably select a public folder for this location because she wants to share the application with others.

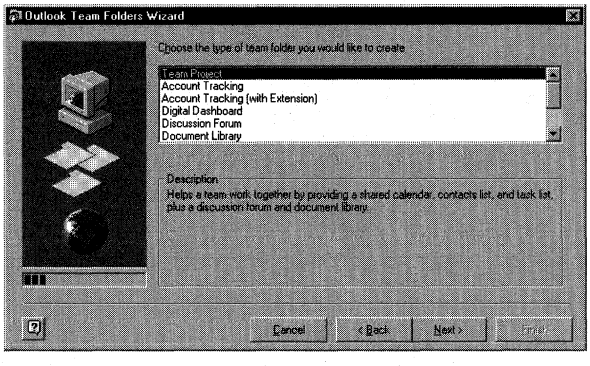


Figure 9-1. *Selecting a template from the Team Folders Wizard. You can extend the built-in templates or add new ones, as you'll see later in the chapter.*

After selecting the location, the user needs to select a Web location on which the HTML files for the Team Folders application will reside. Yes, you read that sentence correctly. The Team Folders Wizard requires you to publish some files to a Web server. If you're only testing your custom application or evaluating the Team Folders Wizard, you can publish your files to your local machine instead of to a Web server by typing something like *c:/temp/* in the Web Page Destination box. Remember that if you publish the files to your local drive, it's going to be hard for anyone else to use the application, unless of course you want to share out your hard drive to everyone else. Figure 9-2 illustrates the step of selecting where to publish your files.

The final step of the wizard allows the user to select the team members for the application using the Global Address List (GAL) in Exchange Server. This step, shown in Figure 9-3, also allows a user to set the permissions for team members and the default permissions for non–team members.

Once you've completed this step, the Team Folders Wizard will create the folder, copy the appropriate files to both the folder and Web server, and allow the user creating the folder to start administering his application. As an option, you can have the Team Folders Wizard invite the team members to join the newly created Team Folders application. When a user accepts this invitation, Outlook adds to that user's Outlook bar a shortcut to the Team Folders application.

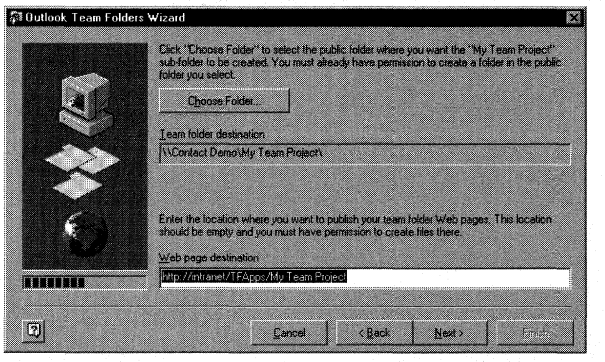


Figure 9-2. *Selecting the Exchange Server location for the Team Folders application and the Web server that will publish the HTML files.*

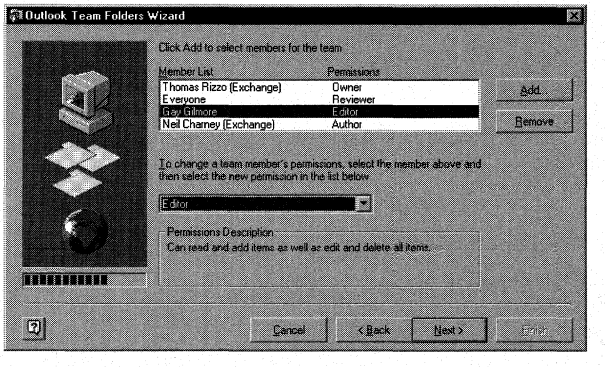


Figure 9-3. *The Team Folders Wizard allows users to select team members in their application and set the permissions those team members will have.*

ARCHITECTURE OF THE TEAM FOLDERS WIZARD

Now that we've seen how the Team Folders Wizard works, let's take a look at the architecture that comprises both the wizard and the applications it deploys. Understanding this architecture will help you build new templates as well as extensions to the wizard.

As stated earlier, the Team Folders Wizard is an Outlook 2000 COM add-in. Therefore, when you click on the File menu and select New Team Folders, you're actually instantiating a COM add-in. One neat feature of this COM add-in is that it loads on demand. When Outlook starts, you don't have to wait for the COM add-in to load; the add-in loads only when you select the New Team Folders menu item. Figure 9-4 depicts the registry entries for the Team Folders COM add-in.

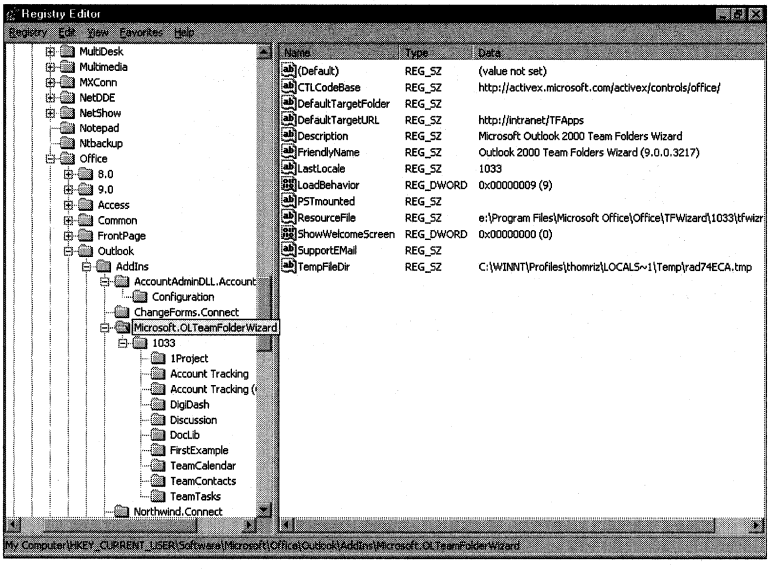


Figure 9-4. *The Team Folders Wizard is implemented as a COM add-in in Outlook 2000. The required registry entries for all COM add-ins are part of the Team Folders Wizard registry entries.*

The Team Folders applications, which are deployed by the wizard, are not COM add-ins. Since we haven't seen a deployed Team Folders application yet, let's look at a completed Team Project application. Figure 9-5 shows the home page of the Team Project application, and Figure 9-6 shows the Team Calendar page in the Team Project application.

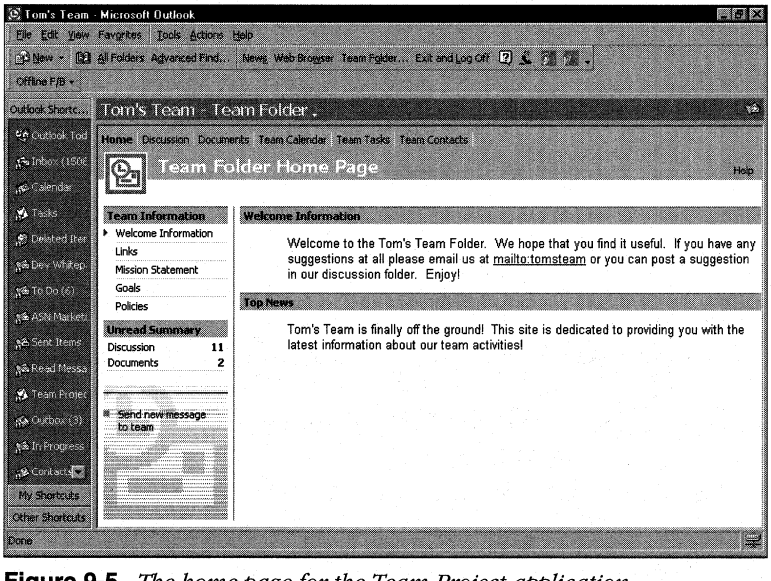


Figure 9-5. *The home page for the Team Project application.*

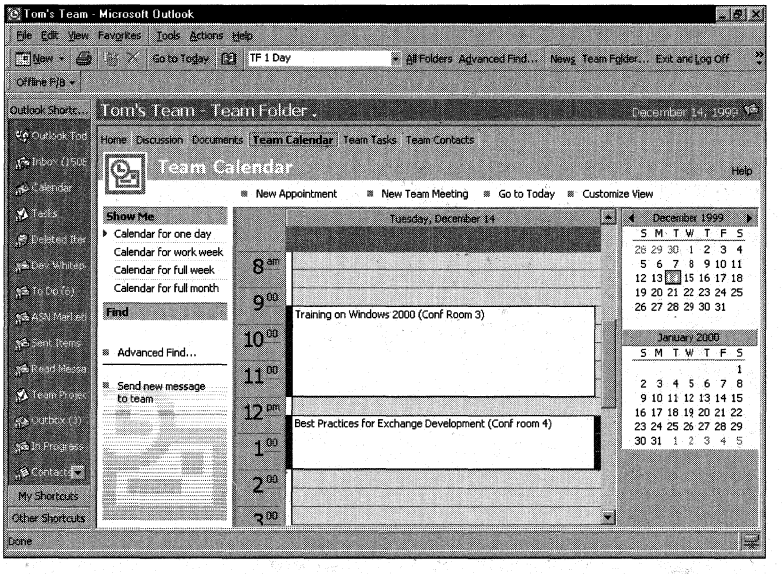


Figure 9-6. *The Team Calendar page for the Team Project application.*

NOTE The Team Project application is hosted in the right-hand pane of Outlook 2000. Your deployed Team Folders applications are actually folder home pages in Outlook 2000. This is why you need a Web server to place the files for the Team Folders application.

You might be wondering whether you can simply host the deployed Team Folders applications directly inside Microsoft Internet Explorer, since they're HTML-based. The answer is "no"—hosting the Web pages inside Outlook provides you with many key benefits, such as full access to the Outlook object model and View control. You'll see how you can directly access Outlook information inside Internet Explorer later in this chapter.

In addition to the COM add-in and the folder home pages, we need to discuss the rest of the architecture for the Team Folders applications. The logic for the Team Folders applications is programmed inside the folder home pages using Microsoft Visual Basic Scripting Edition (VBScript). Furthermore, the Team Folders applications leverage two ActiveX controls that ship with the Team Folders Wizard: the Outlook View control and the Outlook Permissions control.

The Outlook View control, which is similar to the Chart and Spreadsheet Office Web Components, packages Outlook functionality into a reusable component. Therefore, you can place the Outlook View control inside a Web page, Outlook form, or even a Microsoft Visual Basic application. You'll learn more about the Outlook View control a little later in the chapter. The Outlook Permissions control, shown in Figure 9-7, provides a user interface that makes it easy for users to set permissions on a folder.

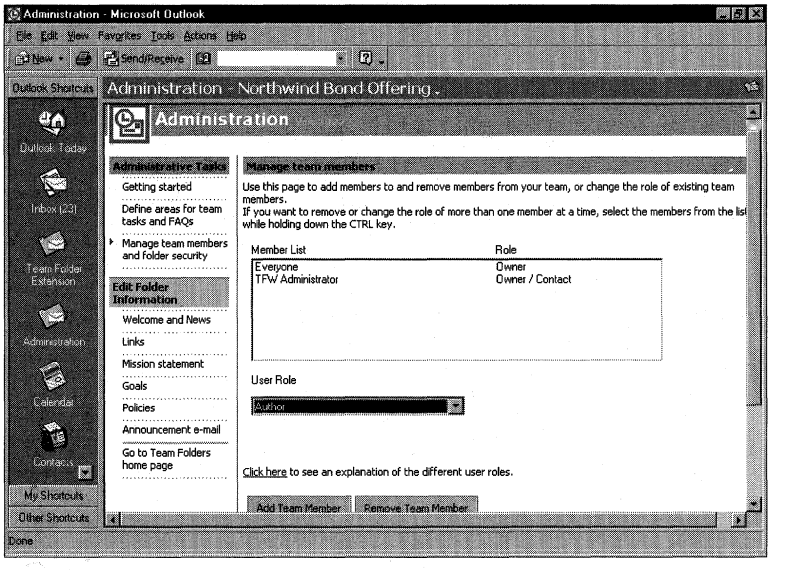


Figure 9-7. *The Outlook Permissions control hosted inside the Administration page for the Team Project application.*

The final part of the Team Folders Wizard architecture is the .pst files that contain the application folders. When you select a template in the wizard, you're actually telling the wizard which .pst file you want it to copy to the folder location selected by the user. We'll see how .pst files work with the Team Folders Wizard when we create our own custom template later in the chapter.

EXTENDING THE TEAM FOLDERS WIZARD

A key requirement of tools like the Team Folders Wizard is to allow developers to extend them in order to customize or replace features of the application. The Team Folders Wizard provides a number of extensibility options for developers who want to customize both the applications that the wizard deploys and the wizard itself.

Modifying the Provided HTML Pages

The first way you can change the applications in the Team Folders Wizard is to modify the HTML pages that comprise the Outlook folder home pages. These HTML pages are stored locally on the machine of the user who is creating Team Folders applications. Therefore, you have to modify these HTML pages either before deploying the Team Folders Wizard or on the desktop of every developer creating Team Folders applications.

The HTML pages that ship with the Team Folders Wizard contain HTML, Dynamic HTML (DHTML), and VBScript code to perform the wizard's functions. Before you begin modifying the HTML pages, be aware that applications such as Team Project have six files you must modify in order to make the changes effective.

Why do you need to make the same changes in these six files? When you deploy a Team Project application, you'll notice that it creates a number of subfolders named Calendar, Contacts, and so on. Figure 9-8 shows this hierarchy of folders.

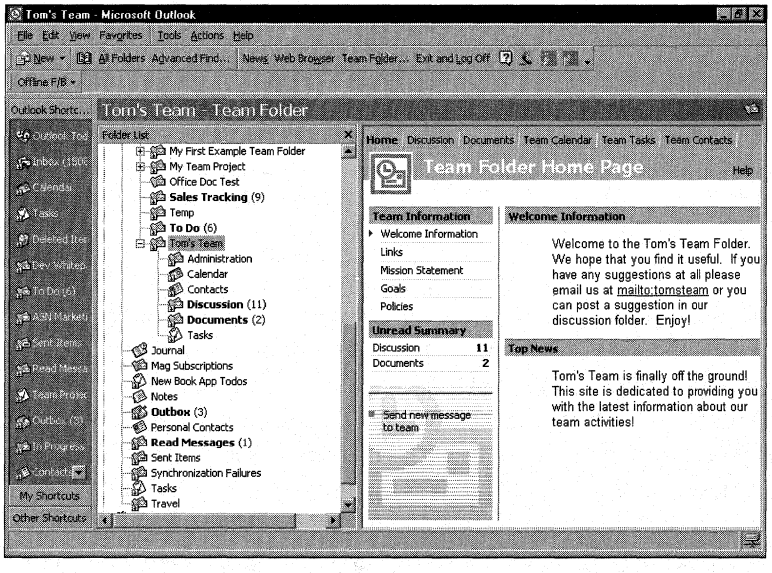


Figure 9-8. *The hierarchy of folders created by the Team Project application.*

The folder home page HTML file associated with the topmost folder is default.htm. However, if you click on the Calendar folder, the folder home page HTML file associated with it will be cal.htm. You might be wondering why the Calendar folder doesn't just use the default.htm file to display its information. There is a good reason for this. Imagine that instead of navigating to the Calendar folder by using the default links at the top of the HTML page, a user opens the Outlook folder list and selects the Calendar folder. The desired default behavior is to have the Team Project application open but default to the Team Calendar section of the project. If default.htm was associated with the Calendar folder, the Welcome page would be displayed.

The Team Project application is the only Team Folders application for which you need to modify multiple HTML pages. The other applications contain only one or two files that you need to modify, such as default.htm and admin.htm. Be aware that you also might see hcal.htm or hcon.htm in the same folder, the help files for the Team Folders applications. When a user clicks the Help button in the folder home

page, these help files will display help information in a Help window. You also might want to modify these files in order to add help information to your custom functionality.

> **WARNING** The HTML pages in the Team Folders applications might operate differently than the HTML pages you're used to. For example, when you click on the hyperlinks at the top of the default page for the Team Project application, you might expect a separate HTML page to load, showing the contents of a folder associated with the hyperlink you clicked. Instead, the HTML page changes dynamically. The Outlook View control is also dynamically updated to point to the newly selected Outlook folder.
>
> Therefore, if you select Team Calendar in the default.htm file at the root of the Team Project application, you won't navigate to the Team Calendar folder and cal.htm. Instead, the buttons and user interface change dynamically in default.htm, and the Outlook View control embedded in the page points at the Team Calendar folder. Watch out for this when you modify the Team Project application.

Now that you've seen the HTML included with the Team Folders applications, let's take a look at adding your own custom functionality to your applications. Since these files are simply HTML files, you can add HTML buttons, DHTML, ActiveX controls, and VBScript functions to them. The Outlook team designed the Team Folders Wizard applications with developer customization in mind.

You will find hints about how HTML pages are constructed in the HTML files themselves. For example, the default.htm file in the Team Folders Wizard applications includes an index for the HTML page that guides you to the portion of the code you must modify in order to meet your needs, as shown in the next bit of code. Suppose you bring up default.htm and search on Folder Name Properties (FNP). You can then jump to the section that allows you to specify folder name properties. This helps you both modify the HTML functionality as well as localize the content of your Team Folders application to another language.

```
'##############################
'Index for this page
'If you would like to make modifications to any part of this page, this
'index will help you locate the correct place in the code to make your
'changes
'LOC --> Localizable Areas
'FNP --> Folder Name Properties
'PCV --> Page Characteristic Variables
'NBP --> Navigation Bar Properties
'VTP --> View Tab Properties
'DVT --> Default View Tab
'TBB --> Navigation Buttons
'VCT --> View Control Tabs
'##############################
```

THE OUTLOOK VIEW CONTROL

Before we see an example of how to modify the Team Folders HTML pages, let's take a closer look at the Outlook View control. This control ships as part of the Team Folders Wizard, plus you can download it from Microsoft's Web site. The Team Folders Wizard provides a reference to the Internet location of the Outlook View control as part of the control's *CodeBase* property in the Team Folders HTML pages. For those of you who don't want to look at the Team Folders HTML pages, the location is *http://activex.microsoft.com/activex/controls/office/outlctlx.cab.*

You must install Outlook on your local machine before you can take advantage of the Outlook View control. This differs from the requirements for the Office Web Components, which only mandate that you have a license for Office 2000 on the local machine, not the actual Office application.

Programming the Outlook View Control

To take full advantage of the Outlook View control, you have to program it to meet your needs. This section highlights the most common methods and properties you can use with the View control. (For the full documentation for the Outlook View control, see the companion CD.) This section will also show you how to program the View control within HTML pages. You can also use and program the control in other ActiveX containers, such as Visual Basic or Outlook forms. The techniques for programming the View control in these other environments will be the same, except you won't need to instantiate the control using an <OBJECT> tag.

Instantiating the Outlook View Control

Before you can program the Outlook View control, you need to instantiate it. You can do this in one of two ways. You can use a tool such as Microsoft Visual InterDev to insert the control, which will provide the <OBJECT> tag needed to create the View control in your HTML application. Your other choice is to create the <OBJECT> tag for the Outlook View control yourself. The following <OBJECT> tag shows your Outlook calendar:

```
<OBJECT classid=CLSID:0006F063-0000-0000-C000-000000000046
codebase="http://activex.microsoft.com/activex/controls/office/
outlctlx.CAB#ver=9,0,0,3203" height="84%"
id=oViewControl style="BORDER-BOTTOM: silver 1px solid" width="100%">
<PARAM NAME="View" VALUE="">
<PARAM NAME="Folder" VALUE="Calendar">
<PARAM NAME="Namespace" VALUE="MAPI">
<PARAM NAME="Restriction" VALUE="">
<PARAM NAME="DeferUpdate" VALUE="0"></OBJECT>
```

No matter which technique you use to create the Outlook View control, you need to understand the parameters that are passed to the control. Notice the ID for the control in this code. The text you specify for the ID (in this case, *oViewControl*) will dictate how you refer to the control in your code. The code also contains a number of parameters: *View*, *Folder*, *Namespace*, *Restriction*, and *DeferUpdate*. Each parameter corresponds to a property of the control that you can set either in the <OBJECT> tag or programmatically. These properties are important for applications that use the Outlook View control. (We're not going to discuss the *Namespace* property below since—let's face it—the only namespace that the View control supports is MAPI.) Let's take a look at the other properties now.

View Property

The *View* property specifies the Outlook view you want to use in the View control—for example, Messages, Messages with AutoPreview, and Day/Week/Month. How do you obtain the list of current views to display in the control? Unfortunately, the Outlook object model doesn't have a Views collection that you can retrieve from. Instead, there are two ways you can programmatically determine which views are in the folder to which the View control is pointing.

The first way, depicted in Figure 9-9, is to grab the list of views from the Current View control on the Outlook Advanced toolbar. To grab the list, you must automate Outlook using the Outlook object model in your application. Get a reference to the Outlook Application object, which you can easily do by using the *OutlookApplication* property of the View control.

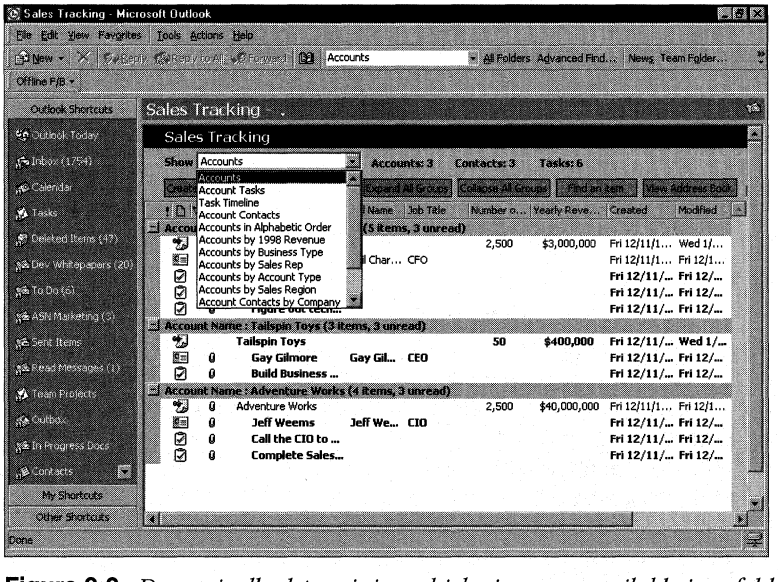


Figure 9-9. *Dynamically determining which views are available in a folder.*

You can then use the *ActiveExplorer* property to retrieve the active Explorer object in Outlook. Next, you can use the Explorer object's *CommandBars* property to retrieve the Office CommandBars collection. From that collection, you can retrieve the Advanced CommandBar object. From that object in turn you can retrieve the current View control. You can now retrieve all the views from that control. In the following code sample, the views are added to a drop-down list in an HTML page. This allows you to force the Outlook View control on the Web page to change views depending on which view you select from the list by using the *View* property on the View control.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML><HEAD>
<META content="text/html; charset=unicode" http-equiv=Content-Type>
<META content="MSHTML 5.00.2919.3800" name=GENERATOR></HEAD>
<BODY>
<P>This sample gets the custom views in the folder.</p>
<P><SELECT id=select1 name=select1() onChange="ChangeView()"
style="HEIGHT: 22px; WIDTH: 333px"></SELECT></P>
<P>
<OBJECT classid=clsid:0006F063-0000-0000-C000-000000000046 height="100%"
id=OVCtl1 width="100%" VIEWASTEXT></OBJECT>
<!-- By passing no parameters to the View control, you can
make it display the default folder it's in -->

<SCRIPT language="VBScript">

Set oApplication = OVCtl1.OutlookApplication
Set oCurrentFolder = oApplication.activeExplorer.currentfolder
set oFolder = oCurrentFolder
strPath = strFullPath
MsgBox "The Current Folder Path is: " & strPath
OVCtl1.Folder = strPath
Set oExplorer = oApplication.Explorers.Add(oFolder,0)
Dim oAdvancedCB
For each oCB in oExplorer.CommandBars
   if oCB.Name = "Advanced" then
      set oAdvancedCB = oCB
      exit for
   End If
next
set oCBCombo = oAdvancedCB.Controls("Current View")
for x= 1 to oCBCombo.ListCount
   AddToSelect(oCBCombo.List(x))
next
'Set the default view for the control to the
'currently selected item in the drop-down list
ChangeView()
```

(continued)

```
Sub AddtoSelect(strViewName)
   set oOption = document.createElement("OPTION")
   oOption.text=strViewName
   oOption.value=strViewName
   document.all.Select1.add(oOption)
End Sub

'*****************************************************
'Function StrFullPath()
'
'This function creates and returns the full path to the
'folder
'*****************************************************
Function StrFullPath()
   strFolderName = ""
   Set olRoot = oCurrentFolder
   While (olRoot <> "MAPI")
      strFolderName = oCurrentFolder.Name & "\" & strFolderName
      Set olRoot = oCurrentFolder.Parent
      If olRoot <> "MAPI" Then
         Set oCurrentFolder = oCurrentFolder.Parent
      End If
   Wend
   strFullPath = "\\" & strFolderName
end Function

Sub ChangeView()
   if select1.value <> "" then
      'Change the View control view
      OVCtl1.View = select1.value
   End If
End Sub

'Make sure to kill the Explorer or else
'Outlook will remain in memory!
set oExplorer = Nothing
</SCRIPT>
</BODY></HTML>
```

The second way you can determine which views are contained in the folder is to use Collaboration Data Objects (CDO). One drawback of this technique is that although CDO can recognize the custom views in the folder, it cannot detect standard Outlook views such as the Messages view or the Messages with AutoPreview view. This is because Outlook stores custom views as hidden messages inside the folder while storing standard Outlook views in a central location. Therefore, CDO doesn't offer you a good way to determine the default view names unless you hard-code them into your application. As you'll see, you can determine whether the default views should be used in the folder.

The advantage of using CDO to detect views is that you can use this technique from Active Server Pages (ASP) or other non-Outlook applications and can detect what the default view is for the folder. Outlook provides no easy way to detect what the folder owner has set as the folder's default view. The only caveat for detecting the folder's default view is that this technique will work only in an online mode. If users are working with an offline version of your application, you won't be able to detect the default view for the folder.

The following example of an HTML page containing some CDO code shows the views contained in the folder:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML><HEAD>
<META content="text/html; charset=unicode" http-equiv=Content-Type>
<META content="MSHTML 5.00.2919.3800" name=GENERATOR></HEAD>
<BODY>
<P>This sample gets the custom views in the folder. It first detects
the type of items in the folder and then displays some default views
for the type of folder as well as the custom views:</P>
<P><SELECT idselect=1 name=select1 onChange="ChangeView()"
style="HEIGHT: 22px; WIDTH: 333px"></SELECT></P>
<P>The default view is currently: </P>
<span id="defaultview"> </span>
<P>
<OBJECT classid=clsid:0006F063-0000-0000-C000-000000000046 height="100%"
id=OVCtl1 width="100%" VIEWASTEXT><PARAM NAME="View" VALUE="">
<PARAM
NAME="Folder" VALUE="inbox">
<PARAM
NAME="Namespace" VALUE="MAPI"><PARAM NAME="Restriction"
VALUE=""><PARAM NAME="DeferUpdate" VALUE="0"></OBJECT>
<SCRIPT ID=clientEventHandlersVBS LANGUAGE=vbscript>

'*****************************************************
'Inline code
'
'These lines of code are run when the browser reaches
'them when parsing the document. They set up the global
'variables that are needed throughout the application.
'*****************************************************
const ActMsgPR_IPM_PUBLIC_FOLDERS_ENTRYID      = &H66310102
const ActMsgPR_STORE_ENTRYID                   = &H0FFB0102
const ActMsgPR_STORE_SUPPORT_MASK              = &H340D0003
const ActMsgSTORE_PUBLIC_FOLDERS               = &H00004000
const ActMsgPR_DEFAULT_ENTRYID                 = &H36160102

Set oApplication = OVCtl1.OutlookApplication
```

(continued)

```
Set oNS = oApplication.GetNamespace("MAPI")
Set oCurrentFolder = oApplication.activeExplorer.currentfolder
strPath = strFullPath
OVCtl1.Folder = strPath
'Log on to CDO
set oSession = oApplication.CreateObject("MAPI.Session")
oSession.Logon "","",False,False
'Have Outlook get the EntryID for the folder
set oCurrentFolder = OVCtl1.ActiveFolder
strEntryID = oCurrentFolder.EntryID
strStoreID = oCurrentFolder.StoreID
'Have CDO get the item
'You can try this code if EntryID, StoreID aren't working
'Set objStores = oSession.InfoStores
'For i = 1 to objStores.Count
'    Set objStore = objStores.Item(i)
'    szID = objStore.Fields.Item(ActMsgPR_IPM_PUBLIC_FOLDERS_ENTRYID)
'    lMask = objStore.Fields.Item(ActMsgPR_STORE_SUPPORT_MASK)
'    Err.Clear
'    If lMask And ActMsgSTORE_PUBLIC_FOLDERS Then
' STORE_PUBLIC_FOLDERS
'         strStoreID = objStore.ID
'      End if
'Exit for
'Next
set oFolder = oSession.GetFolder(strEntryID,strStoreID)
On Error Resume Next
'Figure out the folder type from PR_CONTAINER_CLASS
'This application understands only mail folders
strContainerClass = oFolder.Fields(&H3613001E).value
'Load up the default views for this content type
If strContainerClass = "IPF.Note" Then
   AddtoSelect("Messages")
   AddtoSelect("Messages with AutoPreview")
End If
'Load up the custom views...stored as hidden
'messages with MsgClass
'IPM.Microsoft.FolderDesign.NamedView
'Get the HiddenMessages collection
Set oHiddenMsgs = oFolder.HiddenMessages
'Create a filter, but first clear it
oHiddenMsgs.Filter = Nothing
Set oFilter = oHiddenMsgs.Filter
oFilter.Type = "IPM.Microsoft.FolderDesign.NamedView"
'Scroll through and add hidden messages
For Each oHidden In oHiddenMsgs
   AddtoSelect(oHidden.Subject)
Next
'Figure out the current default view for the folder
```

```
'This may fail if it's an Outlook normal view
'If a custom view, say what it is
err.clear
On Error Resume Next
strViewEntryID = ""
strViewEntryID = oFolder.Fields(ActMsgPR_DEFAULT_ENTRYID).Value
If strViewEntryID = "" Then
   'It's a standard Outlook view
   defaultview.innerHTML = "Built-in Outlook View"
Else
   set oDefaultView = oSession.GetMessage(strViewEntryID,null)
   defaultview.innerhtml = oDefaultView.Subject
End If

Sub AddtoSelect(strViewName)
   set oOption = document.createElement("OPTION")
   oOption.text=strViewName
   oOption.value=strViewName
   document.all.Select1.add(oOption)
End Sub

'*************************************************
'Function StrFullPath()
'
'This function creates and returns the full path to the
'folder
'*************************************************
Function StrFullPath()
   strFolderName = ""
   Set olRoot = oCurrentFolder
   While (olRoot <> "Mapi")
      strFolderName = oCurrentFolder.Name & "\" & strFolderName
      Set olRoot = oCurrentFolder.Parent
      If olRoot <> "MAPI" Then
         Set oCurrentFolder = oCurrentFolder.Parent
      End If
   Wend
   strFullPath = "\\" & strFolderName
End Function

Sub ChangeView()
   If select1.value <> "" then
     'Change the view control view
     OVCtl1.View = select1.value
   End If
End Sub
</SCRIPT>
</BODY></HTML>
```

This code contains some important techniques. Contrary to what you might expect when you use CDO in a Web application, you won't receive an E_ACCESSDENIED error because CDO isn't allowed to be created by HTML client-side code. Why? Imagine you visit an Internet site and a malicious site administrator includes the call CreateObject("MAPI.Session"). He scrolls through your Outlook Inbox looking for passwords, credit card numbers, and so on. Then suppose that, posing as you, he sends an e-mail message containing a virus attachment to the addresses on your contacts list. This potential for security problems is why CDO is disabled in client-side HTML script.

Usually when developers need to use CDO, they wrap their CDO code in an ActiveX control that they sign and distribute. This works well for Internet applications; however, on intranet applications—specifically folder home pages in Outlook 2000—you don't want to create an ActiveX control just so that you can use CDO. Instead, you can use the *CreateObject* method of the Outlook Application object in the Outlook object model, which allows you to create the CDO object in your folder home page. Now before you start screaming security violation, note that this technique will not work if you run the HTML page outside Outlook, either on an intranet or the Internet. Since you're probably linking to trusted content in your folder home pages, you have nothing to worry about. Later in this chapter, you'll see some other ways Outlook locks down its object model when you run Outlook controls in other environments.

The code uses the MAPI property *PR_CONTAINER_CLASS* to determine the default item type in the folder. The code then uses the CDO MessageFilter object to search for all hidden messages in the folder that are of the message class *IPM.Microsoft.FolderDesign.NamedView*. Next, the code scrolls through each of these messages (each of which corresponds to a view) and adds the subject of the message (the name of the view) to the HTML page. Finally, the code uses the *PR_DEFAULT_VIEW_ENTRYID* property on the folder. This property contains the EntryID of the folder's default view. If the default view is an Outlook built-in view, this property will not contain any value.

Folder Property

The *Folder* property specifies which folder you want the View control to display in its interface. When you set this property, you need to pass in the path to the folder that you want to display. You might be wondering how you find the path to a folder. Unfortunately, Outlook doesn't provide a *FolderPath* property in its object model, meaning you must programmatically figure out the path for the folder at which you

want to point the View control. You can do this easily by using the *StrFullPath* function listed in the previous code sample. This function generates the folder path by simply walking the folder hierarchy until it reaches the topmost Outlook folder.

If you leave blank the *Folder* property in an HTML application's <OBJECT> tag, the View control defaults to your Inbox. If you prefer to have the control default to the current folder that the user is looking at in Outlook, do not pass any parameters to the View control inside your folder home page. Be aware that this is a departure from what the View control documentation says to do.

Restriction Property

The *Restriction* property specifies the restriction on the items contained in the folder. This property takes the same format as the *Restrict* method in the Outlook object model. Be aware that when you use the equal sign in this property, Outlook performs a search for strings that contain the criteria you specified. If you want to obtain an exact match for a string, you need to use the "greater than, less than" format. The following example shows the difference between these two formats:

```
'Does a contains
oVCtl1.Restriction = "[Subject] = 'Exchange'"
 'Does an exact match
oVCtl1.Restriction = "[Subject] <= 'Exchange' AND [Subject] >= 'Exchange'"
```

DeferUpdate Property

When set to True, the *DeferUpdate* property prevents the View control from displaying its contents until you reset the property to False. *DeferUpdate* is useful if you're dynamically modifying a folder's restriction or view; it prevents the user from seeing one set of items appear in the control and then suddenly disappear.

OpenSharedDefaultFolder Method

The *OpenSharedDefaultFolder* method allows you to open other user's folders, if you have permissions for those folders, inside the View control. Using this method along with multiple View controls on a single page, you could open up multiple calendars, task folders, or any other folders in your mailbox and other users' mailboxes for which you have permissions. *OpenSharedDefaultFolder* takes the name of the recipient's mailbox that you want to open and the folder type. Figure 9-10 shows how you can use the CDO address book with this method.

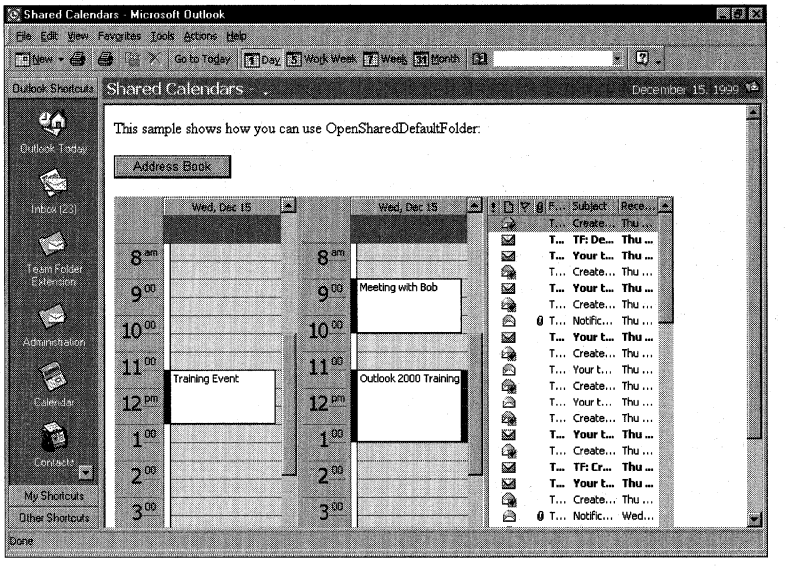


Figure 9-10. *A sample application that shows how to use the CDO address book with the* OpenSharedDefaultFolder *method.*

The following code allows you to select a user from the address book, and then adds a new View control to display that user's Calendar folder. This sample is limited to only three View controls, but you can add more in your code.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML><HEAD>
<META content="text/html; charset=unicode" http-equiv=Content-Type>
<META content="MSHTML 5.00.2919.3800" name=GENERATOR></HEAD>
<BODY>
<P>
This sample shows how you can use OpenSharedDefaultFolder:
<BR><BR>
<INPUT id=AddressBook name=AddBook type=button value="Address Book"
nclick="ShowAddBook()">
</P>
<SPAN ID=View1>
<OBJECT classid=clsid:0006F063-0000-0000-C000-000000000046 height="100%"
id=OVCtl1 width="30%" VIEWASTEXT><PARAM NAME="View" VALUE=""><
PARAM NAME="Folder" VALUE=""><PARAM NAME="Namespace" VALUE="MAPI"><
PARAM NAME="Restriction" VALUE=""><PARAM NAME="DeferUpdate" VALUE="0"></
OBJECT>
</SPAN>
```

```
<SPAN ID=View2>
<OBJECT classid=clsid:0006F063-0000-0000-C000-000000000046 height="100%"
id=OVCtl2 width="30%" VIEWASTEXT><PARAM NAME="View" VALUE=""><
PARAM NAME="Folder" VALUE=""><PARAM NAME="Namespace" VALUE="MAPI"><
PARAM NAME="Restriction" VALUE=""><PARAM NAME="DeferUpdate" VALUE="0"></
OBJECT>
</SPAN>
<SPAN ID=View3>
<OBJECT classid=clsid:0006F063-0000-0000-C000-000000000046 height="100%"
id=OVCtl3 width=30% VIEWASTEXT><PARAM NAME="View" VALUE=""><
PARAM NAME="Folder" VALUE=""><PARAM NAME="Namespace" VALUE="MAPI">
<PARAM NAME="Restriction" VALUE=""><PARAM NAME="DeferUpdate" VALUE="0"></
OBJECT>
</SPAN>

<SCRIPT ID=clientEventHandlersVBS LANGUAGE=vbscript>

'*****************************************************
'Inline code
'
'These lines of code are run when the browser reaches
'them when parsing the document. They set up the global
'variables that are needed throughout the application.
'*****************************************************
Set oApplication = window.external.OutlookApplication
'Log on to CDO
'Use CDO to pop up an Address book so that the person can select
'the user they want to open the calendar for
Set oSession = oApplication.CreateObject("MAPI.Session")
oSession.Logon "", "", False, False
iNumFolders = 0

Sub ShowAddBook()
On Error Resume Next 'to catch the cancel
   Set oRecips = oSession.AddressBook(, "Select a User", True, _
      True, 1, "User")
   If oRecips.Count <> 0 Then
      On Error GoTo 0
      strRecipName = oRecips.Item(1).Name
         If iNumFolders = 0 Then
            On Error Resume Next
            Err.Clear
            OVCtl1.OpenSharedDefaultFolder strRecipName, 9
            If Err.Number = 0 Then
```

(continued)

```
            'Bump up the folder count
            iNumFolders = iNumFolders + 1
         Else
            MsgBox "Error: " & Err.Number & " " & Err.Description
         End If
      ElseIf iNumFolders = 1 Then
         On Error Resume Next
         Err.Clear
         OVCtl2.OpenSharedDefaultFolder strRecipName, 9
         If Err.Number = 0 Then
            'Bump up the folder count
            iNumFolders = iNumFolders + 1
         Else
            MsgBox "Error: " & Err.Number & " " & Err.Description
         End If
      ElseIf iNumFolders = 2 Then
         On Error Resume Next
         Err.Clear
         OVCtl3.OpenSharedDefaultFolder strRecipName, 9
         If Err.Number = 0 Then
            'Bump up the folder count
            iNumFolders = iNumFolders + 1
         Else
            MsgBox "Error: " & Err.Number & " " & Err.Description
         End If
      Else
         MsgBox "No more view controls left!"
   End If
 End If
End Sub
</SCRIPT></P>
</BODY></HTML>
```

Hosting the Outlook View Control in Internet Explorer

When you host the Outlook View control directly inside Internet Explorer, the functionality you can perform in the control's object model is locked down. For example, you can't access the Namespace object from the View control, and you can't use the *CreateObject* method of the Application object. You can, however, create items, delete items, or change views in the folder via the user interface of the View control. Remember, when you host the View control in Internet Explorer, only the object model is locked down. Figure 9-11 and the following code show what happens when you attempt to perform restricted methods while hosting the View control in Internet Explorer.

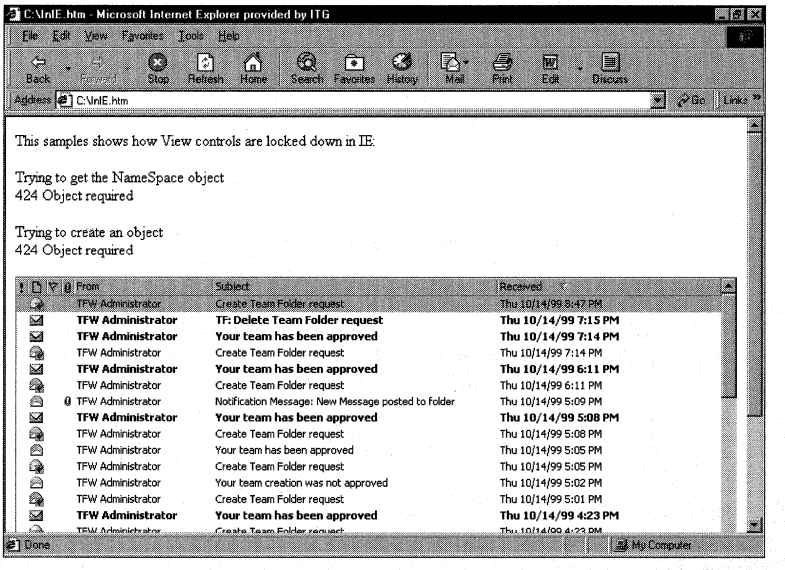


Figure 9-11. *Hosting the View control in Internet Explorer restricts the methods you can use with the control. Notice the errors you receive when attempting certain actions.*

```
<p>This sample shows how View controls are locked down
   in Internet Explorer:
<BR><BR>
Trying to get the Namespace object
<LABEL ID=NameErr></LABEL>
<BR><BR>
Trying to create an object
<LABEL ID=ObjErr></LABEL>

</P>
<OBJECT CLASSID="clsid:0006F063-0000-0000-C000-
000000000046" id=OVCtl1 VIEWASTEXT width=100% height=100%>
  <param name="View" value>
  <param name="Folder" value="inbox">
  <param name="Namespace" value="MAPI">
  <param name="Restriction" value>
  <param name="DeferUpdate" value="0">

</OBJECT>

<SCRIPT defer for=window event=onload ID=clientEventHandlersVBS
   LANGUAGE=vbscript>
```

(continued)

```
'*************************************************
'Inline code
'
'These lines of code are run when the browser reaches
'them when parsing the document. They set up the global
'variables that are needed throughout the application.
'*************************************************
On Error Resume Next
Set oApplication = OVCtl1.OutlookApplication

err.clear
'Try to get Namespace
Set oNS = oApplication.GetNamespace("MAPI")
document.all.NameErr.innerHTML = "<BR>" & err.number & " " & _
   err.description

err.clear
'Try creating an object
Set oSession = oApplication.CreateObject("CDO.Session")
document.all.ObjErr.innerHTML = "<BR>" & err.number & " " & err.description
</SCRIPT>
```

THE OUTLOOK PERMISSIONS CONTROL

The Outlook Team Folders Wizard ships with another control, the Outlook Permissions control, which you can either use as part of the wizard itself or use in the Outlook applications you create. The Team Folders Wizard uses this control in its template administration pages to allow application owners to modify which users have permissions for the Team Folders application.

Since the Outlook Permissions control has an object model, you can reuse the control in your own applications. The Permissions control is best used in Public Folder applications when you're providing a folder home page to a public folder and want users to have the ability to easily change permissions on the application.

The Permissions control requires a user interface. Therefore, you should not use the Permissions control in scenarios that do not require a user interface, such as working with server-side Active Server Pages.

Programming the Outlook Permissions Control

Programming the Outlook Permissions control is quite straightforward. The control properties are usually set at instantiation time in the folder home page in which you're using the control. Since the Outlook team created this control to be localized in multiple languages, a number of the properties allow you to set the labels for the control's user interfaces. Besides these localizable properties, the only properties that we'll look at are *InitSucceeded*, *TargetAdminFolder*, and *TargetFolder*.

NOTE A key requirement of the Permissions control is that CDO must be installed on the local machine.

Permissions Control Properties

The *InitSucceeded* property returns a Boolean that specifies whether the control successfully initialized on the Web page. If the user is working offline or doesn't have the correct permissions on the folder, the control will return False for this property. You should check this property in your Web applications that use the Permissions control.

The *TargetAdminFolder* property specifies the fully qualified path to the administration folder. If you don't set this property, the control will default to the *TargetFolder* property value with \administration appended.

The *TargetFolder* property specifies the fully qualified path to the root folder of your application. When you use the Permissions control's *Update* method, which we'll discuss momentarily, the control starts at the folder you specify in the *TargetFolder* property and sets permissions for all subfolders under that folder.

Permissions Control Methods

There are four Permissions control methods that you'll want to use in your applications: *Add*, *List*, *Remove*, and *Update*. When combined, these four methods provide the functionality needed to set, retrieve, and display permissions for your folder.

The *Add* method invokes an Outlook address book so that the user can select the person for which he wants to add permissions. Once a user is selected, he receives the default permissions selected in the control.

The *List* method returns the list of users who have permissions on the folder specified in the *TargetFolder* property. You can specify one of two list types you want returned to you. The first type is a list of the users' display names, separated by semicolons. To receive a list of this type, specify *Users* or a constant of 1. The second type is a list of the e-mail aliases for the users who have permissions on the folder. To receive a list of this type, specify *UserEmail* or a constant of 2.

The *Remove* method will remove the user who is currently selected in the control. After calling this method, you must call the *Update* method to make your changes permanent.

The *Update* method, to which you always must pass a constant of 1, will make permanent your changes in the control to the folder contained in the *TargetFolder* property and all that folder's subfolders. The only exception is the Administration folder, for which all users (except those with Owner or OwnerContact rights) receive Reviewer permissions so that they can read (but not modify) messages in the folder. This is because the Administration folder contains welcome messages, links, and other information in the Welcome page of a Team Folders application.

Example: Permissions Control Web Application

The following Web application shows how you can take advantage of the Outlook Permissions control in your own applications. Note that if you leave the *TargetAdminFolder* property blank, the Permissions control applies your permissions to all the subfolders of the folder specified in the *TargetFolder* property. However, if you set the *TargetAdminFolder* property to some path, the Permissions control will give all the users in the control Reviewer permissions for the administration folder you specify, even if they already have different permissions for that folder. Figure 9-12 shows how you can use this control to update permissions in Public Folders.

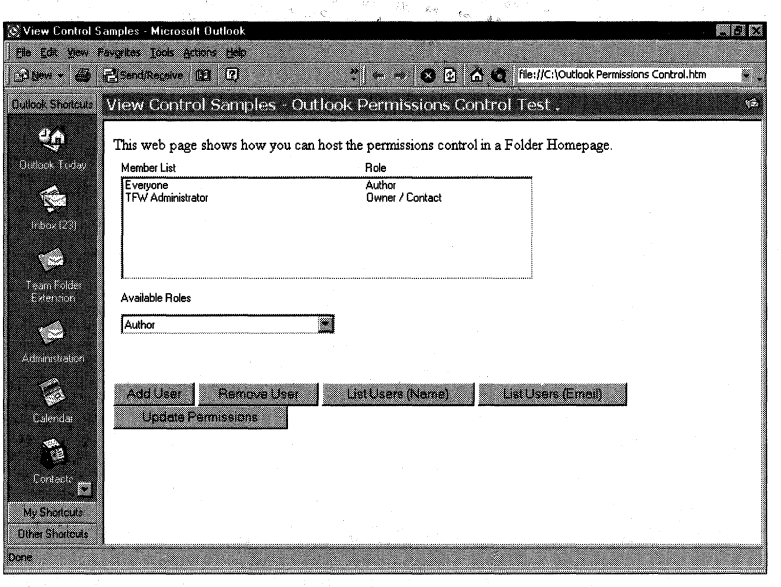


Figure 9-12. *A folder home page hosting the Outlook Permissions control. Using this control, you can easily update permissions in public folders.*

```
<HTML>
<HEAD>
<META NAME="GENERATOR" Content="Microsoft Visual Studio 6.0">
<TITLE>Outlook Permissions Control Test</TITLE>
<Script Language=VBScript>
Function CreateControl()
   dim L_SelectTeam_Text, L_AddMemberLabel_text, L_RoleEveryone_Text, _
      L_RoleNoAccess_Text, L_RoleEditor_Text, L_RoleAuthor_Text
   dim L_RoleReviewer_Text, L_RoleCustom_Text, L_RoleNonEditAuthor_Text, _
      L_RoleOwner_Text
   dim L_RoleOwnerContact_Text, L_LabelUserName_Text, _
      L_LabelUserRole_Text,L_LabelAvailableRoles_Text
   dim L_ProgFormCaption_Text

   L_SelectTeam_Text = "Select Team Members"
```

```
L_AddMemberLabel_text = "Add"
L_RoleEveryone_Text = "Everyone"
L_RoleNoAccess_Text = "No Access"
L_RoleEditor_Text = "Editor"
L_RoleAuthor_Text = "Author"
L_RoleNonEditAuthor_Text = "Nonediting Author"
L_RoleReviewer_Text = "Reviewer"
L_RoleOwner_Text = "Owner"
L_RoleCustom_Text = "Custom"
L_RoleOwnerContact_Text = "Owner / Contact"
L_LabelUserName_Text = "Member List"
L_LabelUserRole_Text = "Role"
L_LabelAvailableRoles_Text = "Available Roles"
L_ProgFormCaption_Text = "Updating Permissions"

Dim MyCont

MyCont = "<OBJECT classid=clsid:1786454A-B4A0-11D2-97C7-"&_
000000000000  codebase=" & Chr(34) & "_
"http://activex.microsoft.com/   activex/controls/office/
olTFACL.cab#version=1,3210,0,1" & Chr(34) & " id=OLTFCTRL1 style=" & _
Chr(34) & "HEIGHT:240px" & Chr(34) & ">"
MyCont = MyCont & "<param name=" & Chr(34) & "_ExtentX" & Chr(34) & _
   " value=" & Chr(34) & "11827" & Chr(34) & ">"
MyCont = MyCont & "<param name=" & Chr(34) & "_ExtentY" & Chr(34) & _
   " value=" & Chr(34) & "9948" & Chr(34) & ">"
MyCont = MyCont & "<param name=" & Chr(34) & "TargetFolder" & _
   Chr(34) & " value=" & Chr(34) & _
   "\\Public Folders\All Public Folders\View Control Samples" & _
   Chr(34) & ">"
MyCont = MyCont & "<param name=" & Chr(34) & "TargetAdminFolder" & _
   Chr(34) & " value=" & Chr(34) & _
   "\\Public Folders\All Public Folders\View Control Samples\ & _
   "Outlook Form"  & Chr(34) & ">"
MyCont = MyCont & "<param name=" & Chr(34) & "EveryoneLabel" & _
   Chr(34) & " value=" & Chr(34) & L_RoleEveryone_Text & Chr(34) & ">"
MyCont = MyCont & "<param name=" & Chr(34) & "AddressBookTitle" & _
   Chr(34) & " value=" & Chr(34) & L_SelectTeam_Text & Chr(34) & ">"
MyCont = MyCont & "<param name=" & Chr(34) & "AddressBookToLabel" & _
   Chr(34) & " value=" & Chr(34) & L_AddMemberLabel_text & Chr(34) &
   ">"
MyCont = MyCont & "<param name=" & Chr(34) & "RoleNoAccess" & _
   Chr(34) & " value=" & Chr(34) & L_RoleNoAccess_Text & Chr(34) & ">"
MyCont = MyCont & "<param name=" & Chr(34) & "RoleEditor" & _
   Chr(34) & " value=" & Chr(34) & L_RoleEditor_Text & Chr(34) & ">"
MyCont = MyCont & "<param name=" & Chr(34) & "RoleAuthor" & Chr(34) & _
   " value=" & Chr(34) & L_RoleAuthor_Text & Chr(34) & ">"
MyCont = MyCont & "<param name=" & Chr(34) & "RoleNoneditingAuthor" & _
```

(continued)

```
      Chr(34) & " value=" & Chr(34) & L_RoleNonEditAuthor_Text & _
      Chr(34) & ">"
   MyCont = MyCont & "<param name=" & Chr(34) & "RoleReviewer" & _
      Chr(34) & " value=" & Chr(34) & L_RoleReviewer_Text & Chr(34) & ">"
   MyCont = MyCont & "<param name=" & Chr(34) & "RoleOwner" & Chr(34) & _
      " value=" & Chr(34) & L_RoleOwner_Text & Chr(34) & ">"
   MyCont = MyCont & "<param name=" & Chr(34) & "RoleCustom" & Chr(34) & _
      " value=" & Chr(34) & L_RoleCustom_Text & Chr(34) & ">"
   MyCont = MyCont & "<param name=" & Chr(34) & "RoleOwnerContact" & _
      Chr(34) & " value=" & Chr(34) & L_RoleOwnerContact_Text & _
      Chr(34) & ">"
   MyCont = MyCont & "<param name=" & Chr(34) & "LabelUserName" & _
      Chr(34) & " value=" & Chr(34) & L_LabelUserName_Text & Chr(34) & ">"
   MyCont = MyCont & "<param name=" & Chr(34) & "LabelUserRole" & _
      Chr(34) & " value=" & Chr(34) & L_LabelUserRole_Text & Chr(34) & ">"
   MyCont = MyCont & "<param name=" & Chr(34) & "LabelAvailableRoles" & _
      Chr(34) & " value=" & Chr(34) & L_LabelAvailableRoles_Text & _
      Chr(34) & ">"
   MyCont = MyCont & "<param name=" & Chr(34) & "ProgFormCaption" & _
      Chr(34) & " value=" & Chr(34) & L_ProgFormCaption_Text & _
      Chr(34) & "></OBJECT>"
   CreateControl = MyCont
End function

Sub AddUser()
   OLTFCTRL1.Add
End Sub

Sub RemoveUser()
   OLTFCTRL1.Remove
End Sub

Sub UpdateUsers
   Dim UList
   On Error Resume Next
   UList = OLTFCTRL1.Update(1)
   If err.number <> 0 then
      MsgBox "Error: " & err.number & " " & err.description
   End If
End Sub

Sub ListUser(LType)
   Dim UList
   On Error Resume Next
   UList = OLTFCTRL1.List(LType)
   If err.number <> 0 Then
      MsgBox "Error: " & err.number & " " & err.description
   End If
   MsgBox UList
End Sub
```

```
</SCRIPT>
</HEAD>
<BODY>
This Web page shows how you can host the Permissions
control in a folder home page.
<Script Language="VBScript">
document.writeln(CreateControl())
</SCRIPT>
<DIV>
<INPUT type="button" value="Add User" onclick="AddUser"
id=button1 name=button1>
<INPUT type="button" value="Remove User" onclick="RemoveUser"
id=button2 name=button2>
<INPUT type="button" value="List Users (Name)" onclick="ListUser(1)"
id=button3 name=button3>
<INPUT type="button" value="List Users (Email)" onclick="ListUser(2)"
id=button4 name=button4>
<INPUT type="button" value="Update Permissions" onclick="UpdateUsers"
id=button5 name=button5>
</DIV>
</BODY>
</HTML>
```

THE TEAM FOLDERS WIZARD ADMINISTRATION EXTENSION

To enhance the capabilities of the Team Folders Wizard, you can use the Team Folders Wizard Administration Extension provided on this book's companion CD. This extension provides a number of features that the Team Folders Wizard lacks. These features make it easier for administrators to control the creation of Team Folders applications and are listed here:

- Approval and rejection of Team Folders applications
- Creation of distribution lists (DLs) for Team Folders applications
- Deletion of Team Folders applications, including DLs and folders associated with the application

The Administration Extension also incorporates some Exchange Server features that make the Team Project application easier to use. The extension provides a useful subscription/notification feature for any public folder application. The user can subscribe to a public folder, and an agent will notify her when a new item has been created there. In addition, the agent will send a link to the new item created in the folder.

Architecture of the Team Folders Wizard Administration Extension

Before diving into the code behind this extension, let's look at its architecture. The extension is comprised of a mailbox for the Event Scripting Agent. This agent processes Team Folders application requests, processes the approval/rejection of those requests, and dynamically creates the subscription/notification agents in the team folders themselves. The agent uses a distribution list, known as the Administrative Agent Distribution List (AADL), which names the administrators who can approve or reject the creation of Team Folders applications.

In addition to this agent, a client-side ActiveX control runs as part of the extension. This control checks to see whether the Team Folders application has been approved; if it hasn't, the control doesn't allow the Team Folders application to display any windows. To host this control, we modify the Team Folders HTML files to load the ActiveX control.

There are two aspects of the extension you should be aware of. First, I modified only the Team Project template to support the extension. After looking through the changes in the Team Project template, you should be able to easily modify the other templates of the Team Folders Wizard to support the extension. Second, I didn't sign the ActiveX control, and I have the CodeBase pointing to the standard Team Folders CodeBase location I specified when I set up the Team Folders Wizard. I didn't want to sign the control because you might want to give it your own signature, depending on how you use it. If you use the control as is from the companion CD, Outlook will ask whether you want to run an ActiveX control on a Web page.

Installing the Team Folders Wizard Administration Extension

Rather than detail the steps for installing this extension from the companion CD, I'll highlight the tools included with it that make installing the extension much easier. I'll also show you how to use a number of Exchange Server programming components, such as Active Directory Services Interface (ADSI) and the Event Scripting Configuration Library.

The companion CD contains three installation programs for the Team Folders Wizard Administration Extension. The first one is the agent installation program, which you must run on your Exchange server. The agent installation program creates the mailbox for the Team Folders Wizard Administration Extension agent and allows you to select membership for the AADL. This installation program also creates the agent and dynamically populates its script. Figure 9-13 depicts the user interface for the agent installation program. You can see the code for this installation program on the companion CD.

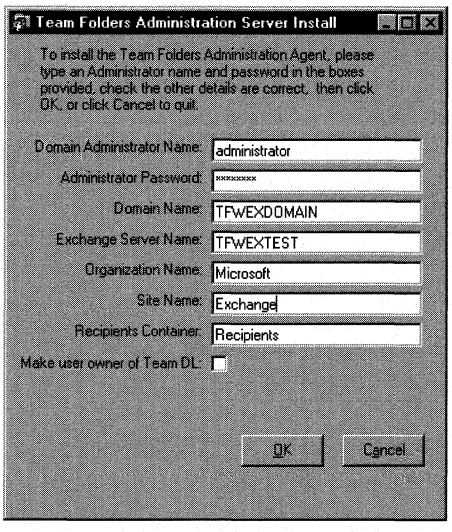


Figure 9-13. *The agent installation program for the Team Folders Wizard Administration Extension.*

The second installation program on the companion CD is the template installation program. You need to run this program on all client machines on which you want to use the Team Folders template. This program makes a backup of the original Team Folders templates and replaces the project template with the modified template. We'll look at the code for this later in the chapter.

The final program installs the ActiveX control. You can either modify the *CodeBase* property on the HTML pages to point at the .cab file included for the ActiveX control, or you can manually register the control from the .cab file.

Modifying the HTML for a Team Folders Wizard Template

Since the Team Folders templates are HTML files, you can easily modify them to fit your needs, including adding custom controls, HTML, or even script to the page. By looking at the modifications the extension makes to the HTML pages, you can determine what types of modifications you can make to your own applications.

The key modifications the extension makes to the Administration page of the Team Project template are the addition of a menu item, a Delete This Team button, and the ActiveX control. Since modifying the HTML for the Team Folders templates is pretty straightforward, I won't spend much time discussing how to modify the intrinsic HTML buttons and other elements of the user interface. However, the ActiveX control and some of the script in the HTML page contain the interesting code we'll examine next. Figure 9-14 shows the user interface of the modified Administration page.

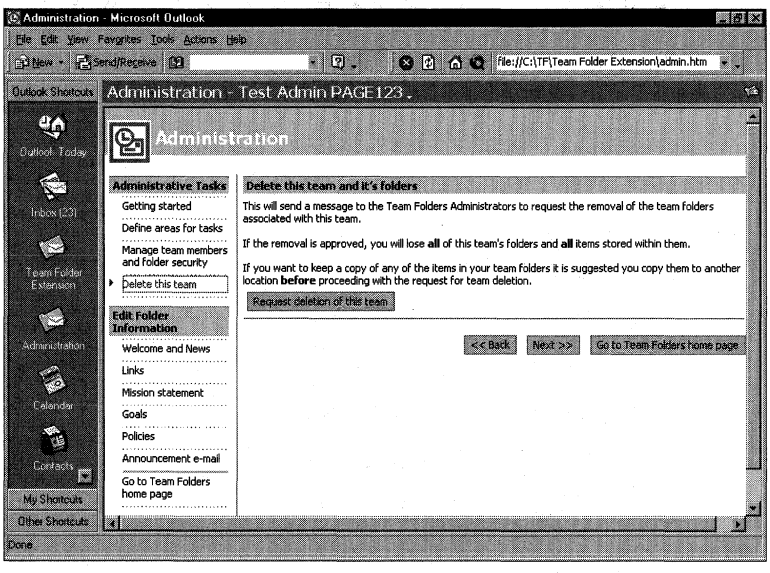


Figure 9-14. *The modified Administration page. Notice the new team deletion capabilities.*

The code in this section shows some of the modifications the extension makes to the Administration page. The first code snippet uses the ActiveX control called Session, which checks to see whether the Team Folders application has been approved. If it hasn't, the code displays the nice message shown in Figure 9-15.

```
<SCRIPT LANGUAGE="vbscript">
<!--
Function IsApproved()
   Dim CurrentState
   CurrentState = Session.TeamApproved(g_Fullpath, False,True)
   Select Case lcase(CurrentState)
   Case "requested"
      IsApproved = False
   Case "notadministrator"
      IsApproved = False
   Case "approved"
      IsApproved = True
   Case Else
      IsApproved = False
   End Select
End Function

   If (not IsIEOK) or (Not IsOLOK) or  (cInt(lErrCode) <> 0) Then
   'there were no initialization errors
      ErrorBox.style.display = "inline"
   Else
```

```
      'Added by the Team Folders Wizard Extension.
      'Check to see whether this team has been approved.
      If IsApproved = False Then
         IsAdmin = False
         TheBodyUnapproved.style.display = "inline"
      Else    'Added
         TheBody.style.display = "inline"
      End If    'Added
   End If
-->
</SCRIPT>
```

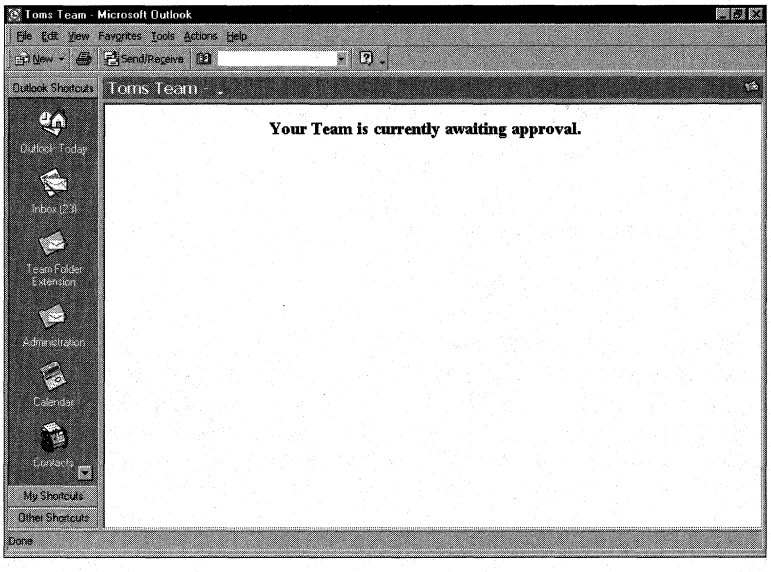


Figure 9-15. *This page is shown if the application is waiting for administrator approval.*

Here is the code for the *TeamApproved* function in the ActiveX control:

```
Public Function TeamApproved(strFolderPath, _
   TopTeamFolder As Boolean, AdminFolder As Boolean)
   On Error GoTo errHandler
   Dim Pos
   Dim strBody

   DoMAPILogon
   'Get the Administration folder and InfoStore

   If GetFolderAndInfoStore(CStr(strFolderPath), oTeamFolder, _
      oInfoStore) = False Then Exit Function
```

(continued)

```
If InDebug Then MsgBox "Got Folder & InfoStore for strFolderPath: " _
   & strFolderPath
Dim strAddressID

strAddressID = GetAddressIDFromAlias(strTFWAAlias)

If strAddressID = 0 Then
   MsgBox "Error getting address of the TFWA alias in TeamApproved."
End If
Set TFWAAddress = oSession.GetAddressEntry(strAddressID)

If InDebug Then
   If Not (TFWAAddress Is Nothing) Then
      MsgBox "Got TFWAAddress"
   Else
      MsgBox "Didn't get TFWAAddress"
   End If
End If

EnsureTFWA_IsIn_ACL oTeamFolder

'Get the main team folder

If TopTeamFolder = False Then
   'We're currently looking at the Administration folder,
   'and need to move up one folder level to access the main team folder

   Set oTeamFolder = oSession.GetFolder(oTeamFolder.FolderID, _
      oInfoStore.ID)

   EnsureTFWA_IsIn_ACL oTeamFolder

   If InDebug Then MsgBox "Looking at : " & oTeamFolder.Name
End If

'We also need to make sure the Administration folder is a member of
'all the subfolders
'If AdminFolder = True Then

    EnsureTFWA_IsIn_ACL_AllSubFolders oTeamFolder

'End If
'Determine the actual team path, and store it for notifying the
'AADL
Pos = InStrRev(strFolderPath, oTeamFolder.Name)
strTeamPath = Left(strFolderPath, Pos + Len(oTeamFolder.Name))
If InDebug Then MsgBox "strTeamPath: " & strTeamPath
```

```
If TeamStatusMessageFound(oTeamFolder) = False Then
   If strErrNo = 0 Then
      If InDebug Then MsgBox "No Team Status Message found"
      'We have not done any of the administration required
      'to start the team approval process

      If AdminFolder = True Then
         If InDebug Then MsgBox "Admin Page"

         'Hide the team folder from the team members
         If InDebug Then
            MsgBox "Attempting to Hide folder from team members"
         End If

            If SetFolderACL(oTeamFolder) = False Then
               TeamApproved = "An Error occurred 1. Ensure The " & _
                  "TFW Administrator has owner rights to this folder."
               Exit Function
            End If
            If InDebug Then MsgBox "Hidden folder from team members"

               'Create the message body for the message
               strBody = CreateMsgBody("Requested")

            If CreateHiddenStatusMessage(oTeamFolder, strBody) = _
               False Then
               TeamApproved = "An Error occurred 2. Ensure The " & _
                  "TFW Administrator has owner rights to this folder."
               Exit Function
            End If

            SendAgentMsg strBody, "TF: Create Team Folder request"
         End If
         TeamApproved = "Requested"
      Else
         Select Case strErrNo
         Case -2147024891
            strErrDesc = "You do not have permission to read " & _
               "entries in this folder."
         Case Else
            'Do nothing
         End Select

         TeamApproved = "Error encountered: " & strErrDesc & _
            " " & strErrNo
         Err.Clear
      End If
```

(continued)

```
        Else
          TeamApproved = strCurrentTeamStatus
        End If
        Exit Function
    errHandler:
        MsgBox "Error in TeamApproved function.  Err #" & Err.Number _
           & " Description: " & Err.Description
End Function
```

The *TeamApproved* function makes sure that the mailbox object of the Team Folders Wizard Administration Extension has Owner rights on all the folders for the application. When performing subscription and notification, this mailbox must be able to dynamically create agents in the application folders—and to do so, the mailbox must have Owner rights.

Next, *TeamApproved* checks to see whether a hidden message exists in the application's root folder. If no hidden message exists, it means that the application is a new one. If that's the case, the control creates a new hidden message and sends an approval notice to the extension agent. The control then sets the application status to *Requested*.

If the control does find a hidden status message in the root folder, it checks the message to see whether the team has been approved, has been rejected, or is still waiting for approval. The control returns this status to the HTML page so that the page can act accordingly.

The next function in the Administration page, *RequestDeleteTeam*, deletes the team. Since the Team Folders applications do not provide a good way for users to delete their applications, the extension offers this capability. This function requests that the agent delete all the applications' team folders, and forces the Outlook client to move to a folder other than the applications' team folder. This prevents Outlook from sitting on the folder while the agent deletes it in the background.

The Event Scripting Agent for the Administration Extension

I'd like to point out three aspects of the Event Scripting Agent for the Team Folders Wizard Administration Extension. (Chapter 13 will cover the Event Scripting Agent in much more detail.) First, the extension's Event Scripting Agent performs extensive logging, as shown in Figure 9-16. This logging allows you to watch the agent actions.

Second, the Event Scripting Agent uses subfolders to log the messages it receives, as shown in Figure 9-17. You can use these subfolders to troubleshoot any problems such as malfunctions in the agent or the functions it performs.

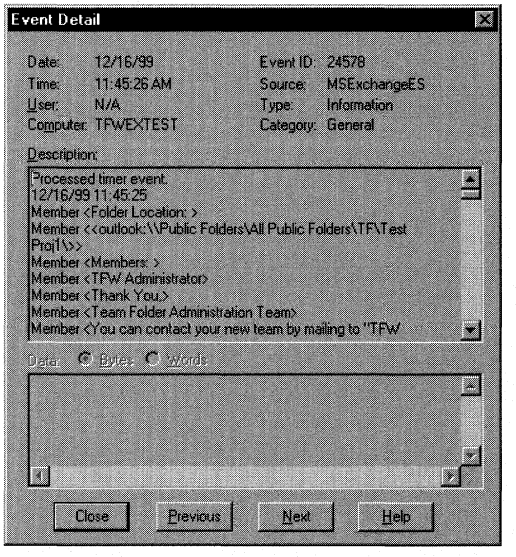


Figure 9-16. *The log from the extension's Event Scripting Agent.*

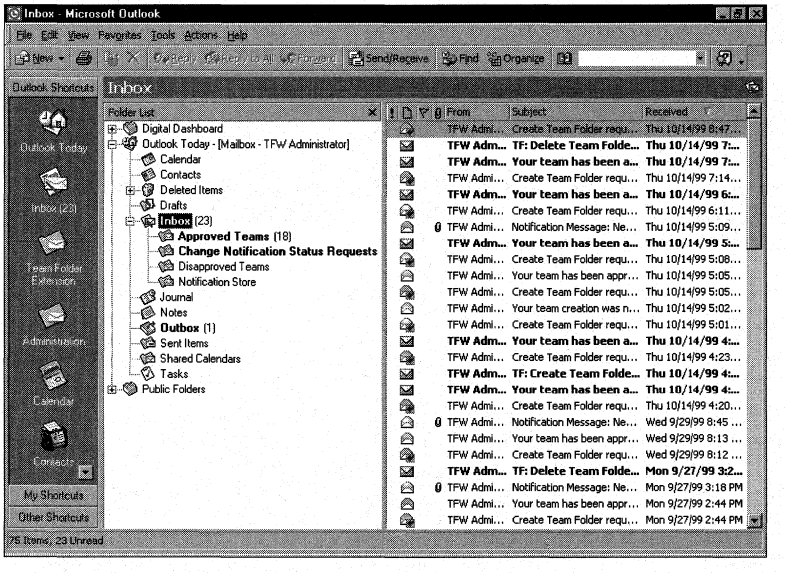


Figure 9-17. *The subfolder structure of the extension mailbox.*

Third, the Event Scripting Agent uses ADSI to dynamically create distribution lists and to add or remove members from these lists. You could take the code from this agent and build a generic agent that manages distribution lists. The code for managing the extension's DLs is shown on the following page.

```
'*************************************************************
'******* Customized Subroutines....                          *
'******* This subroutine creates a DL on the fly             *
'*************************************************************
Public Sub CreateNewDL(DLAlias)

On Error Resume Next

If UserIsOwnerofDL = True Then
   strOwner = objRequestor.Fields(&H3A00001E).Value
Else
   strOwner = "TFWA"
End If
LogIt "strOwner is " & strOwner

strLDAPPath = "LDAP://" & LDAPServer & "/" & LDAPRecipientContainer _
   & "," & LDAPSite & "," & LDAPOrg
strLDAPOwnerPath = "LDAP://" & LDAPServer & "/cn=" & strOwner & ", _
   " & LDAPRecipientContainer & "," & LDAPSite & "," & LDAPOrg

LogIt "LDAP Path: "  & strLDAPPath
LogIt "LDAP UserName: " & LDAPUserName
LogIt "LDAP Password: " & LDAPPassword
Logit "New Alias: " & DLAlias

Set objMyIADs = GetObject("LDAP:")
Set objADSI = objMyIADs.OpenDSObject(strLDAPPath, LDAPUserName, _
   LDAPPassword, 0)
Set objNewDL = objADSI.Create("groupOfNames", "cn=" & CStr(DLAlias))
set objOwner = objMyIADS.OpenDSObject(strLDAPOwnerPath, _
   LDAPUserName, LDAPPassword, 0)

objNewDL.Put "cn", CStr(RequestorsName & " " & TeamFolderName)
objNewDL.Put "uid", CStr(DLAlias)
objNewDL.Put "distinguishedName", CStr("cn=" & CStr(DLAlias) & _
   "," & LDAPRecipientContainer & "," & LDAPSite & "," & LDAPOrg)
objNewDL.Put "owner", objOwner.distinguishedName
objNewDL.Put "mail", CStr(DLAlias) & CStr(strSMTPAddr)
objNewDL.Put "Report-To-Originator", True
objNewDL.Put "Report-to-Owner", False
objNewDL.Put "Replication-Sensitivity", CInt(20)
objNewDL.Put "rfc822Mailbox", CStr(DLAlias) & CStr(strSMTPAddr)
objNewDL.Put "textEncodedORaddress", CStr(strx400Addr) & _
   CStr(DLAlias) & ";"
objNewDL.SetInfo
dim strPath
dim strUID
strUID = objNewDL.Get("uid")
strPath = strLDAPPath & vblf & strUID
LogIt "strPath: " & strPath
```

```
'Save DL alias to a hidden message in the team folder
StoreDLAliasName strPath

End Sub
'*************************************************************
'******* This subroutine adds a new member to a DL            *
'*************************************************************
Public Sub AddMemberToDL(NameToReplace, DLAlias)

On Error Resume Next
strLDAPPath = "LDAP://" & LDAPServer & "/" & NameToReplace & "," _
   & LDAPRecipientContainer & "," & LDAPSite & "," & LDAPOrg
strLDAPRecip = "LDAP://" & LDAPServer & "/cn=" & DLAlias & "," _
   & LDAPRecipientContainer & "," & LDAPSite & "," & LDAPOrg

LogIt "strLDAPPath: " & strLDAPPath
LogIt "strLDAPRecip: " & strLDAPRecip

Set objIADs = GetObject("LDAP:")
Set objLDAPRecip = objIADs.OpenDSObject(strLDAPRecip, _
   LDAPUserName, LDAPPassword, 0)

objLDAPRecip.Add (strLDAPPath)

If Err.Number = -2147016691 Then
'They're already in the DL
Else
   objLDAPRecip.SetInfo
End If
End Sub

'*************************************************************
'******* This subroutine removes a member from a DL           *
'*************************************************************
Public Sub RemoveMemberFromDL(NameToReplace, DLAlias)

On Error Resume Next

strLDAPPath = "LDAP://" & LDAPServer & "/" & NameToReplace & "," _
   & LDAPRecipientContainer & "," & LDAPSite & "," & LDAPOrg
strLDAPRecip = "LDAP://" & LDAPServer & "/cn=" & DLAlias & "," _
   & LDAPRecipientContainer & "," & LDAPSite & "," & LDAPOrg
Set objIADs = GetObject("LDAP:")
Set objLDAPRecip = objIADs.OpenDSObject(strLDAPRecip, _
   LDAPUserName, LDAPPassword, 0)
objLDAPRecip.Remove (strLDAPPath)
objLDAPRecip.SetInfo

End Sub
```

Subscription/Notification Functionality

The last functionality of the Administration Extension that we'll talk about is the subscription/notification functionality. For me, this is the most exciting part of the extension because it makes using any Public Folder application easier. The extension notifies users who subscribed to the Public Folder that a new item is available, meaning users don't have to check for new items in the Public Folder application. Users subscribe to the Public Folder by clicking a button in the home page of any folder in the Team Project application. The subscription/notification functionality is implemented on a folder-by-folder basis. Figure 9-18 shows the interface that allows the user to enable this functionality.

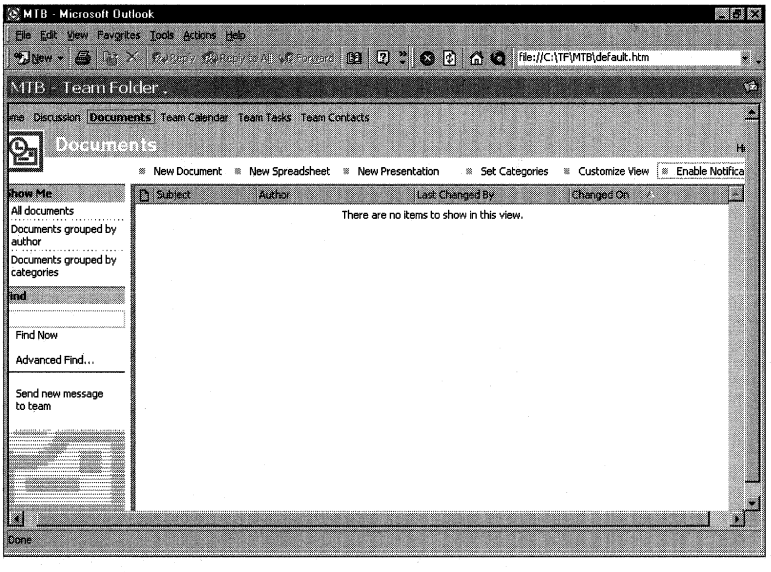


Figure 9-18. *The user interface for enabling subscription/notification.*

The extension tracks which users to notify about new items posted in the folder by storing in the folder a hidden message containing a list of users who've requested notification. The HTML pages in the extension use the *CurrentlyNotified* function from the ActiveX control to see whether the current user is on the folder's notification list. If the user is on the notification list, the text in the HTML offers the ability to unsubscribe to the notifications. Here is the code for the *CurrentlyNotified* function:

```
Public Function CurrentlyNotified(strFolderPath)
   If InDebug Then MsgBox "strFolderPath = " & strFolderPath
   If GetFolderAndInfoStore(CStr(strFolderPath), oTeamFolder, _
      oInfoStore) = False Then
      MsgBox "Could not open folder: " & strFolderPath
      Exit Function
   End If
```

```
    Dim oHiddenMessages, oFilter
    Dim oMessage
    Dim strTeamDLPath, objIADs, objLDAPRecip, arrTeamPath, _
       strDLAlias, strRecipContPath
    Dim arrMembers, strMember

    Set oHiddenMessages = oTeamFolder.HiddenMessages
    oHiddenMessages.Filter = Nothing
    Set oFilter = oHiddenMessages.Filter
    oFilter.Subject = "TFW Member Notification"
    If oHiddenMessages.Count >= 1 Then 'There should only ever be one.
       Set oMessage = oHiddenMessages.GetFirst
       arrMembers = Split(oMessage.Text, vbCrLf)

       CurrentlyNotified = False
       For Each strMember In arrMembers
         If strMember = oSession.CurrentUser.ID Then
           CurrentlyNotified = True
           Exit For
         End If
        Next
    Else
       'No notification message exists; therefore, no one is
       'currently notified
       CurrentlyNotified = False
    End If

End Function
```

If the user asks to be added to the notification list, the application sends a message to the extension's mailbox agent. The extension checks to see whether a notification agent already exists in the requested folder. If it does, the user is added to the hidden notification message in the folder. If the agent doesn't exist, the extension agent creates a new agent in the folder and dynamically binds a script to this newly created notification agent to implement the notification functionality. Figure 9-19 shows a folder containing the dynamically created agent.

The notification agent uses the Windows Scripting Host to create a shortcut to the newly added item and attaches the shortcut to the notification e-mail. The user can then double-click on the shortcut, causing Outlook to open the original item.

The code for the notification agent follows. Be sure to look at the section that uses the Windows Scripting Host, since you can probably reuse this functionality in a number of your Outlook applications.

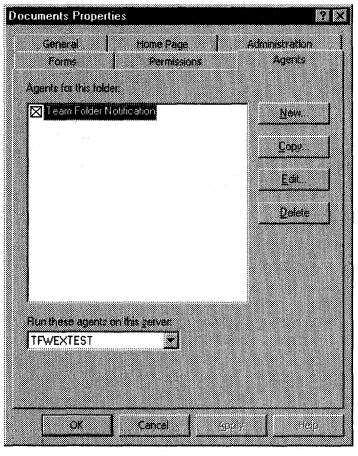


Figure 9-19. *A Team Folders with the notification agent installed in it.*

WARNING There are two caveats to using this solution. First, the solution will work only online because you're specifying a path to an item that's online. Second, if two items have the same subject in the folder, it's undetermined which one Outlook will open.

```
'DESCRIPTION: This event is fired when a new message is added
'to the folder
Public Sub Folder_OnMessageCreated()
   Const CdoFileData = 1

   Dim oNewMsg, oHidden, aryText, iCount, blnFoundMsg, objAddrEntry
   Dim objSession, objFolder, aryHiddenMsgText, strRecip
   Dim WSHShell, MyShortcut, strUserProfile
   Dim objInMsg, MyAttachment
   Dim strPath

   On Error Resume Next
   Set objSession = EventDetails.Session
   Set objFolder = objSession.GetFolder(EventDetails.FolderID, Null)
   LogIt "Running"
   LogIt "Folder: " & objFolder.Name

   blnFoundMsg = False

   For iCount = objFolder.HiddenMessages.Count To 1 Step -1

      Set oHidden = objFolder.HiddenMessages(iCount)
      If oHidden Is Nothing Then
         LogIt "oHidden is empty -"&_
```

```
      "there should be at least 1 member to notify "
      Exit For
   End If
   If oHidden.Subject = "TFW Member Notification" Then
      aryHiddenMsgText = Split(oHidden.Text, vbCrLf)
      blnFoundMsg = True
      Exit For
   End If
Next

Set oHidden = Nothing

If blnFoundMsg = False Then
   LogIt "Hidden is missing from folder " & objFolder.Name
   'Do nothing but exit
   Set objSession = Nothing
   Set objFolder = Nothing
   Exit Sub
End If

LogIt "Got Hidden "


Set oNewMsg = objSession.Outbox.Messages.Add
For Each strRecip In aryHiddenMsgText
   If strRecip <> "" Then
      Set objAddrEntry = objSession.GetAddressEntry(strRecip)
      oNewMsg.Recipients.Add , , , objAddrEntry.ID
      Script.Response = Script.Response & vbCrLf & strRecip
   End If
Next

oNewMsg.Recipients.Resolve
If oNewMsg.Recipients.Count = 0 Then
   LogIt "No Recipients to notify"
   Exit Sub
End If
Set objInMsg = objSession.GetMessage(EventDetails.MessageID, Null)

'This will create a shortcut and attach it to a message
Set WSHShell = CreateObject("WScript.Shell")

LogIt "Started"
LogIt "Subject: " & objInMsg.Subject

strUserProfile = WSHShell.ExpandEnvironmentStrings("%USERPROFILE%")
```

(continued)

```
    Set MyShortcut = WSHShell.CreateShortcut(strUserProfile & _
       "\msolink.url")
    LogIt strUserProfile

    Dim oParent
    strPath = "\" & objFolder.Name
    'Get the path to the folder; assume it's in the Public Folder tree
    set oParent = objSession.GetFolder(objFolder.FolderID,Null)
    Do While oParent.Name <> "All Public Folders"
       strPath = "\" & oParent.Name & strPath
       set oParent = objSession.GetFolder(oParent.FolderID,Null)
    Loop
    strPath = "\\Public Folders\All Public Folders" & strPath
    Dim strPathwithoutFileName
    strPathwithoutFileName = strPath
    strPath = strPath & "\~" & objInMsg.Subject
    LogIt "strPath: " & strPath

    'Set shortcut object properties and save them
    'You can also try the format outlook:EntryID
    'MyShortcut.TargetPath = "outlook:" & objInMsg.ID
    MyShortcut.TargetPath = "outlook:" & strPath
    LogIt "MyShortcut.TargetPath:  " & strPath
    MyShortcut.Save
    LogIt "Saved"

    Set MyAttachment = oNewMsg.Attachments.Add("Shortcut to " & _
       objInMsg.Subject)
    MyAttachment.Type = CdoFileData 'Attach the link, not a shortcut to it.
    MyAttachment.Source = strUserProfile & "\msolink.url"

    LogIt "New Message posted to folder "
    oNewMsg.Text = "A message has been posted to the folder <outlook:" _
    & strPathwithoutFileName & "> " & vblf & _
    "This message was posted by "  & objInMsg.Sender.Name & "." & _
    vblf & "The subject of the message is: " & objInMsg.Subject & _
    "." & vblf & "Open the attachment in this mail to read the new item."
    oNewMsg.Subject = "Notification Message: New Message posted to folder"
    oNewMsg.Update True, True
    oNewMsg.Send False, False

    Set MyShortcut = Nothing
    Set oNewMsg = Nothing
    Set WSHShell = Nothing
    Set objSession = Nothing
    Set objFolder = Nothing
    Set objInMsg = Nothing
End Sub
```

BUILDING A CUSTOM TEAM FOLDERS TEMPLATE

Although customizing the built-in templates can be interesting, you might want to create your own template and have the Team Folders Wizard deploy it. There are a number of reasons for doing this. First, the Team Folders Wizard provides an easy way for your users to create applications based on your template. Second, the Team Folders Wizard deploys your application to a folder. This includes copying forms, views, and folder home pages to the application folder. Instead of having to write a deployment and customization application yourself, you can simply leverage the Team Folders Wizard.

Creating a custom template is straightforward. Every custom template must consist of a personal folders file (.pst), a folder home page, and a Template initialization file (.ini). The .pst file contains the folders, items, views, and forms you want to deploy. The folder home page is the Web interface you want to display to the user. The .ini file contains settings that you specify for the wizard when it deploys your application.

Creating the .pst File

To create the .pst file, take your existing application and its folder and simply copy them to a personal folder. If you're creating a new application, you can just create a new .pst file and proceed. You'll need to create a Team Folders root folder to contain your application. The wizard will copy this root folder to the location specified by the user who is deploying your application using the Team Folders Wizard. The .pst file can also contain an Administration folder, which you need if you want your application to have an administration module.

Creating the Folder Home Page

There's no special tasks that you need to perform inside of the folder home page in order to allow the Team Folders Wizard to deploy it. However, you might want to create a separate folder home page for each folder in your application. This arrangement differs from that of the built-in Team Folders templates, which use one folder home page that dynamically changes the HTML when a user clicks on its links.

Creating the Template.ini File

The Template.ini file tells the Team Folders Wizard how to specifically deploy your application. In this file, you tell the wizard how to replace strings in your folder home pages, associate application folders with folder home page files, and localize your application if needed. A sample Template.ini file follows on the next page.

```
'*************************************************************
'Template.ini File                                            *
'*************************************************************
'
'The individual files are made up of a list of keys and replacement items.
'The keys are the values that should be searched for in the file.
'
'Notation is as follows:
'
'       My String Name=mystring <none - default string>
'
'The global section applies to all HTML files listed in the
'HTMLFiles section. This could provide a convenient mechanism
'for changing elements such as:
'
'Style sheet, corporate logo, and so on
'
'*************************************************************



'-------------------------------------------------------------------
'GLOBAL SECTION
'-------------------------------------------------------------------
'[Global] keys added by the search engine at run time:
'
'(1) Title
'
'Retrieved from user entry for project name in Team Folders Wizard.
'Replaces <OLTF@Title> tag in HTML files.
'
'(2) CodeBaseURL
'
'Specifies location for download of View and Permissions controls.
'It is CtlCodeBase value for the wizard add-in entry under
'HKEY_CURRENT_USER (HKCU), created during installation of Outlook 2000
'Team Folders. Replaces <OLTF@CodeBaseURL> tag in HTML files.
'
'Registry value for CtlCodeBase can be customized with Custom Installation
'Wizard (CIW) or overwritten by running Codebase.bat to install Outlook
'2000 Team Folders. Codebase.bat is included with the Team Folders
'Administration Kit.
'
'[Global] keys that MUST be defined by template designer:
'
'(1) AdminFolder
'
'Used for updating permissions for the project Administration folder
```

```
'*** Requires full path relative to root folder. ***
'
'(2) ShowNavBar
'
'Used to show (value of 1) or hide (value of 0) folder navigation
'buttons
'Replaces <OLTF@ShowNavBar> tag in HTML files
'
'-----------------------------------------------------------------

[Global]
'AdminFolder=Administration\
ShowNavBar=0

'-----------------------------------------------------------------
'INCLUSIVE HTML FILES LIST SECTION
'-----------------------------------------------------------------
'Following section lists the names of all the files that
'need to be edited
'
'-----------------------------------------------------------------

[HTMLFiles]
acctext.htm
'admin.htm

'-----------------------------------------------------------------
'INDIVIDUAL HTML FILES SECTION
'-----------------------------------------------------------------
'Following sections have search and replace actions
'specific to the file
'
'-----------------------------------------------------------------

[acctext.htm]
'value=string

[Admin.htm]
'value=string

'-----------------------------------------------------------------
'FOLDER URL MAPPING
'-----------------------------------------------------------------
'This section identifies which HTML file webview each
'project folder is to be associated with
'
'Assumptions:
'
```

(continued)

```
'* All target HTML files are in the \Webview directory.
'* There is no limit to nesting of the project folders.
'* There can be only one root per project.
'* The folder names cannot contain the equal sign since this is being
'  used as the item delimiter  (key=value).
'* Folder URL mapping occurs **after** folder name
'  localization.
'
'[Folders] section pseudo key=value format: (table
'key=idnum,foldername,webview,fullpathrelativetoroot
'
'example: 10=Update Permissions,OLTFaclsTest.htm,\administration\
'
'path notation
'
'root folder:                                    "root"
'folders nested one level deep:                  "\"
'folders nested > 1 level must specify path:     "\folder1\folder2\"
'-----------------------------------------------------------------


[Folders]
'1=Account Tracking,acctext.htm,root
'2=Administration,admin.htm,\

'-----------------------------------------------------------------
'PUBLISH UNIQUE MESSAGE CLASS
'-----------------------------------------------------------------
'This section identifies which folders contain forms that need
'unique message class assignment
'
'*** Please Note: ***
'
'The ID assigned to each folder in the project from the [Folders]
'section above is also referenced for folder identity in this section
'
'(1) The first item in the comma-delimited value string represents the
'    ID of the folder containing the form.
'(2) The second item in the comma-delimited value string represents the
'    existing form MessageClass.
'(3) The last item in the comma-delimited value string represents the
'    concatenated IDs for folders with associated HTML files that need
'    to be updated with the new unique MessageClass string.
'(4) The folder IDs used in this section must match the assignments in the
'    [Folder] section.
'
'  [FormClassAssignment] section pseudo key=value format: (table)
'
```

```
'ex) idnum=locationfolderid,messageclass,HTMLFolderId1+HTMLFolderId2+...
'
'-----------------------------------------------------------------

[FormClassAssignment]
'1=1,IPM.Post.Account info,1
'2=1,IPM.Contact.Account contact,1

'-----------------------------------------------------------------
'LOCALIZABLE STRINGS
'-----------------------------------------------------------------
'This section identifies strings that can be localized.
'At the moment, this consists solely of the folder names.
'
'-----------------------------------------------------------------

[Strings]
```

Replacing HTML Strings in Template.ini

By having the Team Folders deploy your application, you can have the wizard replace HTML strings inside your folder home page files. One way to take advantage of this is to change the CodeBase location for ActiveX controls or change constants in your HTML code, depending on the application the user chose to deploy.

The strings in your HTML page must be marked with the format OLTF@searchstring. For example, to replace a string named "*username*" with the string *"Thomriz"*, you would place OLTF@username in your HTML file. Then, if the name of the HTML file in your Template.ini file was default.htm, you would place a line under the individual HTML files section, as shown in the following example:

```
[default.htm]
username=Thomriz
```

You can replace the strings either globally or at the individual file level. This example shows how to perform a replacement at an individual file level. The Team Folders Wizard performs a global replacement for names such as OLTF@CodeBaseURL, which specifies the location for the View control and Permissions control, and OLTF@Title, which contains the name of the application provided by the user.

Global Section in Template.ini

In addition to specifying the replacement strings in the global section of Template.ini, you need to specify the path to the Administration folder (if you have one) and whether to display the Internet Explorer navigation bars so that you can move backward and forward between Web pages in your folder home page. The navigation bar setting, ShowNavBar, is helpful only if you use the Team Folders built-in templates or if you implement the functionality that turns on and shuts off the navigation bar (the same functionality that the Team Folders templates implement in your application).

The AdminFolder key takes the full path, relative to your application's root folder, to the Administration folder in your .pst file. For example, if your folder is located directly under the root folder in the .pst file, the value for this key will be *"Administration\"*.

HTMLFiles Section in Template.ini

The HTMLFiles section in Template.ini specifies the HTML files that are included in your application. This section also specifies on which files to perform search and replace actions for your global replacement strings.

Folders Section in Template.ini

The Folders section contains the mappings between your Outlook folders and their corresponding folder home page HTML files. The format for each mapping is keynum=foldername; pagename; path where keynum is any unique integer; foldername, which is the folder name; pagename, which is the HTML file to associate with the folder; and path, which is the path, relative to the root application folder, to the folder. Note that if the folder is the root folder, you need to place *root* for the path. Also, you should not include the name of the folder when you specify the path. For example, if you wanted to associate the default.htm HTML file to a folder called My Application under your application's root folder, the Folders section of Template.ini would look like this:

```
[Folders]
1=My application,default.htm,\
```

FormClassAssignment Section in Template.ini

I recommend that you don't use the FormClassAssignment section of the Template.ini file. This section publishes a unique message class for your form when the form is deployed to a folder. This is important because the Outlook forms cache doesn't work correctly when multiple forms with the same name are published and used from different folders. Instead of using this section of the Template.ini file, give your forms unique names to prevent them from clashing with forms in other applications on your user's system.

Strings Section in Template.ini

The Strings section contains strings that you want to localize throughout the Template.ini file. For example, if you wanted to localize the word *Calendar* into multiple languages, you could place the following key under this section:

```
[Strings]
CalendarFolder = Calendar
```

Then, throughout Template.ini, you would use *%CalendarFolder%*. The Team Folders Wizard will replace all instances of *%CalendarFolder%* with the literal you specify in the Strings section before processing the Template.ini file.

File Folder Structure for Your Template

When creating the file folder for your template files, including the .pst file and Template.ini file, you need to lay out your folder correctly in order for the Team Folders Wizard to process it. You must place the Template.ini file and the .pst file at the root level of your folder, and you must place the HTML files for your folder home page in a folder named Web view.

Deploying Your Template

Once you've created your .pst file, folder home page files, and Template.ini file, you must deploy your template to the computers of the users who will create applications based on it. In addition to copying the actual files to the users' computers, you'll need to add certain settings to their registries so that the Team Folders Wizard can detect the new template.

To make the template available, you need to create a key under HKEY_CURRENT_USE\Software\Microsoft\Office\Outlook\Addins\Microsoft.OLTeamFolderWizard\1033. The number at the end of this path will vary depending on the language settings for the computer you're deploying to. In this case, *1033* indicates that the language is English. Under this key, you'll need to create a number of other keys:

- *AppPath*. This key specifies the path of the template in 8.3 format. If your template is located at C:\program files\microsoft office\template\myapp, you'll need to specify C:\progra~1\micros~1\templa~1\myapp.
- *Description*. This key contains a string value that describes your template to the users of the Team Folders Wizard. The wizard displays this string when a user clicks on the application in the wizard's interface.
- *FriendlyName*. This key contains a string value that the wizard displays in the list of applications available. The user can select this name in the list box of available templates.
- *PSTAppRoot*. This key contains a string value that specifies the path, relative to the root of your .pst file, to your template's root folder. If your template's root folder is the root of the .pst, this string is the name of the template's root folder. For example, if your template is named Accounts and is located beneath the root of your .pst, this value would be *Accounts*.
- *PSTName*. This key contains the name of the .pst file that contains your application. The Team Folders Wizard will dynamically try to mount this .pst file when users create your applications. This file must be in the folder specified by your AppPath key.

- *PSTTitle*. This key contains a string that specifies the name of the .pst file in the Outlook interface when the Team Folders Wizard mounts your .pst. Make sure that this name does not conflict with another folder root in your Outlook client.
- *UseFinishScreen (optional)*. This keyword contains a DWORD value that specifies whether to show the final screen of the wizard after it deploys your application. If you do not create an Administration page for your application, set this value to *0* so that the wizard does not display an error when it finishes deploying your application.

You can also customize other keys for the Team Folders Wizard that will affect your application. The main keys for the Team Folders Wizard are located at HKEY_CURRENT_USER\Software\Microsoft\Office\Outlook\AddIns\Microsoft.OLTeamFolderWizard. Figure 9-4 on page 276 shows these registry keys.

- *DefaultTargetFolder*. This key contains a string value that specifies the default Public Folder path to install your application. The user can accept this location in the wizard rather than having to browse for it. An example value for this key is \\Public Folders\All Public Folders\Applications\My Application.
- *DefaultTargetURL*. This key contains a string value that specifies the default address of a Web location for the folder home pages for your application. The wizard will display this address, and the user can accept it as the location from which to deploy the HTML pages.
- *UseWelcomeScreen*. This key contains a DWORD value that specifies whether to display the Welcome page to the Team Folders Wizard. If you specify a value of *1*, the Welcome page will display; specifying a value of *0* will not display the Welcome page.
- *CTLCodeBase*. This key contains a string value that indicates the CodeBase location for the ActiveX controls, specifically the View control and the Permissions control, in Team Folders applications.

Example: Account Tracking Template

In order to show you how to build a custom template, I've taken the Account Tracking application and turned it into a Team Folders template. This sample template makes it easier for multiple groups in a corporation to deploy the Account Tracking application with its forms, views, and starter items intact. Figure 9-20 shows how you can add the Account Tracking application to the list of templates in the Team Folders Wizard.

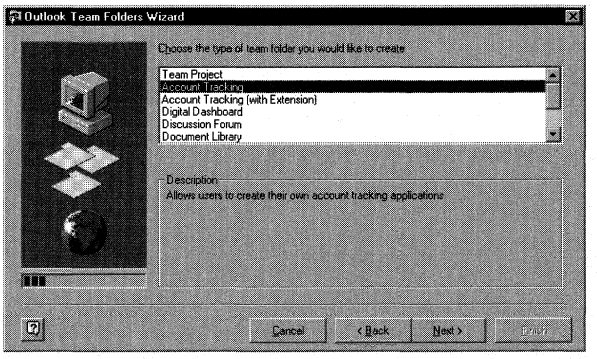


Figure 9-20. *The Account Tracking application as one of the Team Folders Wizard templates.*

While modifying the Account Tracking application to make it part of the Team Folders Wizard, I added a number of new features to the application's folder home page, shown in Figure 9-21. First, the application detects the custom views contained in the folder so that you don't have to hard-code them into the application. Second, the application shows you how to use the *Restriction* property of the View control.

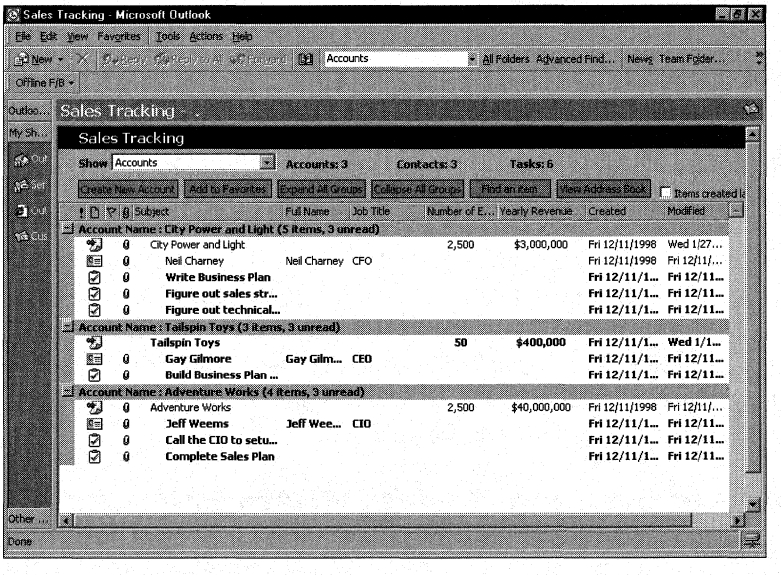


Figure 9-21. *The new folder home page for the Account Tracking application.*

The final and most interesting feature of the new folder home page is that it shows you how to detect whether the application has the default Outlook views enabled. A user or developer can specify that the application should not use the standard Outlook views. You need to detect whether the default Outlooks views are enabled in your applications so that you don't display them in your user interface.

The application detects the setting for the default Outlook views by checking a property on one of its folders. The hexadecimal value for this property is &H36E1003. This property will return a value of *1* if the default Outlook views should not be displayed for the folder; if they should be displayed, either the property will return *0* or the field won't exist. The following code checks this property in the folder home page:

```
'Load the default views for this content type
   'Make sure that the default views are enabled first,
   'by checking a property on the folder
   On Error Resume Next
   If oFolder.Fields(&H36E10003).Value = 1 Then
      'Default views should not be displayed!
      'Do not attempt to add them
   Else
      'Default views should be displayed
      'Note: This sample adds only two of the default
      'views for message folders
      'You can use the CommandBars method to get the default Outlook
      'views if you want
      If strContainerClass = "IPF.Note" Then
         AddtoSelect "Messages",oElement
         AddtoSelect "Messages with AutoPreview",oElement
      End If
   End If
```

CREATING A TEAM FOLDERS WIZARD EXTENSION

In addition to creating a template for the Team Folders Wizard, you can extend the wizard interface itself. Extending the wizard interface allows users to further customize your applications before the wizard deploys them. The Team Folders Wizard extensions are registered individually for each template. Therefore, to add an extension to all templates, you need to register the extension with each template. You'll see how to register an extension later in this chapter.

Interfaces of the Team Folders Wizard Extension

A Team Folders extension is an ActiveX DLL that implements specific interfaces required by the Team Folders Wizard. The wizard uses these interfaces to inform an extension of specific events, such as when the user enters the portion of the wizard that implements the extension. This occurs after the user has selected the permissions for the Team Folders application but before the wizard displays the last screen.

When creating your ActiveX DLL, you should implement a similar interface to that of the Team Folders Wizard. This means implementing Back, Next, and Cancel buttons directly on your form. Plus, you need to make sure that you code the extension

correctly so that if a user returns to it after clicking the Next button, the extension can correctly return to the appropriate screen. You don't want your extension to repeat any unnecessary steps.

Creating an extension is quite easy. This book's companion CD contains a sample extension for the Account Tracking template. This extension allows the user creating an application from the Account Tracking template to specify whether to automatically add the folder to her Favorites folder as well as to add a shortcut on the Outlook bar. Figure 9-22 shows the extension's user interface.

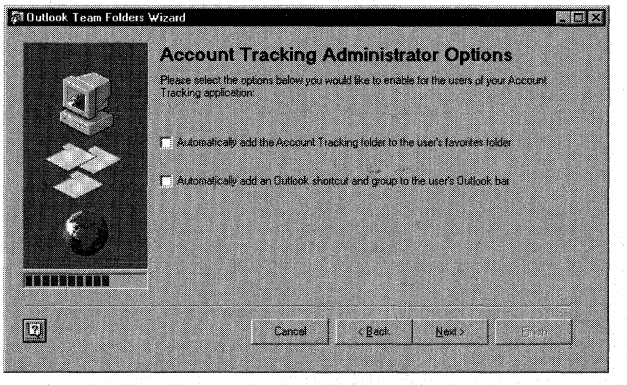


Figure 9-22. *The user interface of the Account Tracking extension.*

Now, you need to add some references to your Microsoft Visual Basic project in order to determine which extension interfaces to implement in your ActiveX DLL. First, you need to add a reference to the file tfexten.dll (ITeamFoldersExtension typelib). Next, add the Microsoft Scripting Runtime reference. Figure 9-23 shows how to add these references.

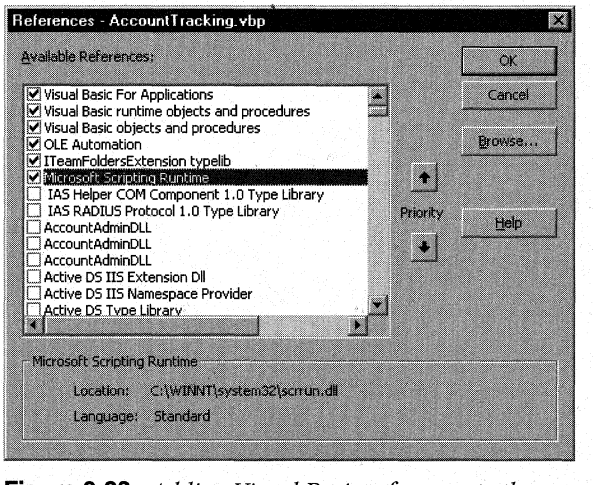


Figure 9-23. *Adding Visual Basic references to the necessary files to create a Team Folders extension.*

Once you've added these references, you need to type *Implements ITeamFolderExtension* in your Visual Basic code. In the drop-down lists, you'll see three interfaces that you need to implement: *ExtExec*, *ExtUndo*, and *ExtCancel*.

The wizard calls the *ExtExec* function when the user enters your extension by clicking on Next in the wizard. Here is a stub for this function:

```
Private Function ITeamFoldersExtension_ExtExec( _
   ByVal UserPerms As Scripting.IDictionary, _
   ByVal TempDirectory As String, ByVal CurrentStep As Integer, _
   ByVal TotalSteps As Integer) As Long
End Function
```

The Team Folders Wizard passes you a number of parameters for this interface. The first parameter is a Dictionary object called *UserPerms* that contains the permissions list for the application. The second parameter, *TempDirectory*, specifies the path to the temporary directory where the folder home page files for the application are located before being deployed. You have Write permissions to these files in the temporary directory. The Account Tracking extension uses this location to modify the files to specify the preferences of the user. The third parameter, *CurrentStep*, specifies the number of steps of the wizard that the user already has been through. When you combine *CurrentStep* with the next parameter, *TotalSteps*, you can display the progress bar showing the progress of your wizard's extension.

Your extension should return a value of Long data type, which tells the wizard whether the user clicked Next, Back, or Cancel, or if the extension didn't work. The specific literals for this return value appear in the following code for the Account Tracking extension's *ExtExec* function that shows the user interface:

```
Private Function ITeamFoldersExtension_ExtExec( _
   ByVal UserPerms As Scripting.IDictionary, _
   ByVal TempDirectory As String, ByVal CurrentStep As Integer, _
   ByVal TotalSteps As Integer) As Long
'Shows user interface
'Returns  Success = 0
'         Failure = -1
'         Back = 1
'         Cancel = 2
'This extension needs to know only where the working directory is so that
'it can modify the HTML file
    Debug.Print "in ITeamFolderExt"

    gWizardStep = CurrentStep + 1
    gWizardSteps = TotalSteps
    gTempDirectory = TempDirectory
    Debug.Print gTempDirectory & gWizardSteps
    frmAppOptions.Show vbModal    'as modal so that it doesn't
                                  'return too early
    ITeamFoldersExtension_ExtExec = gRetValue
End Function
```

As the code shows, the extension retrieves the values for the step counters, gets the temporary directory, and then displays its Visual Basic form as a modal form. Once the user leaves the form, the extension returns the correct value as determined by the user's actions in the form.

The wizard calls the *ExtUndo* function when the user reenters your extension after he has already left. This can happen if the user clicks the Back button on one of your subsequent extension pages or on the wizard's final page to return to a specific extension. This function receives the same parameters as the *ExtExec* function and returns a Long value that means the same as each value in the *ExtExec* function. For example, returning a value of *0* indicates success. The code for this function follows:

```
Private Function ITeamFoldersExtension_ExtUndo( _
   ByVal UserPerms As Scripting.IDictionary, _
   ByVal TempDirectory As String, ByVal CurrentStep As Integer, _
   ByVal TotalSteps As Integer) As Long
Should show user interface from last step in extension
'Returns   Success = 0
'          Failure = -1
'          Back = 1
'          Cancel = 2
    gWizardStep = CurrentStep - 1
    gWizardSteps = TotalSteps
    gTempDirectory = TempDirectory
    frmAppOptions.Show vbModal
    ITeamFoldersExtension_ExtUndo = gRetValue
End Function
```

As you can see, the *ExtExec* and *ExtUndo* functions are very similar. The function sets the progress bar for the extension to the correct value. It then displays the Visual Basic form as a modal form, and after the user is finished interacting with the form, the function returns the correct return value. Depending on what your extension does, you might need to implement more functionality.

The *ExtCancel* function is called when the user clicks the Cancel button elsewhere in the wizard. In this function, place any cleanup code that your extension requires. This function passes you the same parameters as the *ExtUndo* and *ExtExec* functions; however, the return value for *ExtCancel* is a bit different than that of the previous two functions. The *ExtCancel* function can return one of only two values rather than four. Since the user has clicked Cancel, she won't care about moving backward or forward in the wizard. Therefore, the return value needs to indicate only success or failure. Here is the code for this function:

```
Private Function ITeamFoldersExtension_ExtCancel( _
   ByVal UserPerms As Scripting.IDictionary, _
   ByVal TempDirectory As String, ByVal CurrentStep As Integer, _
```

(continued)

```
    ByVal TotalSteps As Integer) As Long
'Should show no user interface
'Returns   Success = 0
'          Failure = -1
    gTempDirectory = TempDirectory
    'MsgBox " Implement Cancel functionality for extension here"
    ITeamFoldersExtension_ExtCancel = eSuccess

End Function
```

Visual Basic Form for the Account Tracking Extension

Besides implementing the extension interfaces, the Visual Basic project for the Account Tracking extension includes the actual Visual Basic form shown to the user as part of the application. This form contains all the logic for the extension as well as the navigation for the user. The code for the form follows:

```
Dim strText
Dim oCreatedFile As Scripting.File
Dim fs As Scripting.FileSystemObject
Dim iFavorites
Dim iAddShortcut

Private Sub btnBack_Click()
    gRetValue = eStepBack    'Function returns 1
                             'indicating extension Undo
    frmAppOptions.Hide
End Sub

Private Sub btnHelp_Click()
    MsgBox "Insert Your Help Routines Here"
End Sub

Private Sub btnNext_Click()
    'Need to set return value for function and exit function
    'OLTFWizardExt returns 0 if successful
    Dim fsResult
    Dim oFile As Scripting.File
    Dim oBUFile As Scripting.File
    Dim oTextStream As Scripting.TextStream

    'Open the HTML file and set some global variables
    'storing the user's selections
    Set oFile = fs.GetFile(strBrowseAddress)
    If oFile Is Nothing Then
        MsgBox "Error opening acctext.htm"
    Else
```

```
        'Before writing to the file, make a clean backup
        'Make sure not to overwrite if already there!
        On Error Resume Next
        oFile.Copy gTempDirectory & "\webview\acctext2.htm", False
        On Error GoTo 0
        Set oBUFile = fs.GetFile(gTempDirectory & "\webview\acctext2.htm")
        'Overwrite the backup over the original
        oBUFile.Copy strBrowseAddress, True
        Debug.Print "made copy"
        'Reset oFile
        Set oFile = fs.GetFile(strBrowseAddress)
        Set oTextStream = oFile.OpenAsTextStream(ForAppending, _
           TristateUseDefault)
        With oTextStream
           .WriteLine "<Script Language = vbscript>"
           .WriteLine "boolAddtoFavorites = " & _
              checkAddtoFavorites.Value
           .WriteLine "boolAddShortcut = " & checkAddShortcut.Value
           .WriteLine "</Script>"
         End With
        oTextStream.Close
    End If


    Debug.Print "creating file"
    'Create a new file in the temporary address to mark down the values
    Set oCompleted = fs.CreateTextFile(gTempDirectory _
       & "\complete.txt", True)
    ' Remember user selections
    oCompleted.WriteLine checkAddtoFavorites.Value
    oCompleted.WriteLine checkAddShortcut.Value
    oCompleted.Close
    Set fs = Nothing

    gWizardStep = gWizardStep + 1
    gRetValue = eSuccess    ' Extension finished; return 0 success code
    Me.Hide
End Sub

Private Sub btnCancel_Click()
    'Insert your cancel functionality here and then set the
    'OLTFWizardExt return value to 2 (Cancel)
    gRetValue = eCancelWiz
    Me.Hide
End Sub

Private Sub Form_Activate()
    progBar.Max = gWizardSteps
    progBar.Value = gWizardStep
```

(continued)

```
    Set fs = New FileSystemObject
    Dim oTS As Scripting.TextStream

    Set oCreatedFile = Nothing
    Debug.Print "in Activate"
    'Check to see whether a file called complete.txt already exists
    On Error Resume Next
    Set oCreatedFile = fs.GetFile(gTempDirectory & "\complete.txt")
    On Error GoTo 0
    If Not (oCreatedFile Is Nothing) Then
       Debug.Print "File exists"
       'Pull the values from the file
       Set oTS = oCreatedFile.OpenAsTextStream(ForReading)
       iFavorites = oTS.ReadLine
       iAddShortcut = oTS.ReadLine
       If iFavorites = 1 Then
          checkAddtoFavorites.Value = vbChecked
       Else
          checkAddtoFavorites.Value = vbUnchecked
       End If
       If iAddShortcut = 1 Then
          checkAddShortcut.Value = vbChecked
       Else
          checkAddShortcut.Value = vbUnchecked
       End If
    End If
End Sub

Private Sub Form_Load()
    'Load form strings
    strText = ""
    'Need to load all strings here
    Debug.Print "Form Load: " & gTempDirectory
    If gTempDirectory <> "" Then
       strBrowseAddress = gTempDirectory & "\webview\acctext.htm"
    Else
       MsgBox "Error - no browsable address found"
    End If
End Sub
```

The interesting aspects of the code lie in the *btn_Next* and *Activate* subroutines. The *Activate* subroutine sets the progress bar for the form that shows the user the wizard's progress based on the number of steps already performed. This subroutine also checks to see whether a file named complete.txt exists in the temporary directory created by the Team Folders Wizard. This file stores the user's preferences, as you'll see in the *btn_Next* subroutine. I use a text file to be absolutely sure that the preferences are stored—just in case the user clicks Back or Next, possibly several times, to move between the extension and the built-in Team Folders Wizard

pages. If the text file doesn't exist, the application creates it. If it does exists, the application loads the values from the text file and makes them the default for the form controls.

The application calls the *btn_Next* subroutine when the user clicks the Next button on the extension's form. In this subroutine, the extension needs to set some variables so that the folder home page knows what the user selected from the extension. If no backup exists, the subroutine first makes a backup copy of the original folder home page file so that if the user unselects something in the user interface, the extension doesn't have to parse and delete text from the HTML file. Instead, the extension can overwrite the existing file with the clean backup and add the necessary information to it.

Once the backup is completed or verified, the subroutine writes the necessary values to the file. Then, the subroutine creates the complete.txt file using the FileSystemObject of the scripting runtime. If complete.txt already exists, the extension overwrites it.

Folder Home Page for the Account Tracking Extension

Now that we've seen what the extension can do, let's see how the folder home page implements the preferences set in the extension. The folder home page included with the extension template is a slightly modified version of the Account Tracking template you saw earlier. The key changes are the ability of the extension to check for preferences set by the user, to implement those preferences, and to make sure those preferences are implemented only once. The code for this functionality follows:

```
'Check to see whether there is a hidden message
'in the user's Inbox that corresponds to our application.
'If not, do the action and create the hidden message.
'If there is, skip this section.
set oInbox = oSession.Inbox
set oHidden = oInbox.HiddenMessages
'Clear out any filters
oHidden.Filter = Nothing
Set oFilter = oHidden.Filter
oFilter.Fields.Add "strEntryID", 8, strEntryID
'If count is 1, then we don't want to do anything
'since we performed this action for the user already
If oHidden.count > 1 then
   MsgBox "Error!  Too many items meet the criteria!"
ElseIf oHidden.count = 0 then
   'Check to see whether we need to add to Favorites
   If boolAddtoFavorites = 1 Then
   AddToFavorites
   End If
```

(continued)

```
    If boolAddShortcut = 1 Then
       AddOutlookShortcut
    End If
    'Create a hidden message stating that we already
    'did all the work for this user so that next time
    'they come in, we don't do it again!
    'Assume everything was done.
    CreateHiddenMessage
Else
    'MsgBox "Hidden message already exists"
End If
End Sub

Function CheckOfflineStatus()
    On Error Resume Next
    If oInfoStore.Fields(&H6632000B).Value = True Then
       CheckOfflineStatus = True
       'MsgBox "Offline"
    Else
       CheckOfflineStatus = False
       'MsgBox "online"
    End If
End Function

Sub AddToFavorites
    'Check to see whether we're in the private store or a .pst file.
    'Check to see whether we're working offline.
    'If we are, don't try to add to Favorites
    'since the folder should already be in the Favorites
    'folder.
    'Get the Favorites folder EntryID.
    'MsgBox "Adding to Favorites"
    Set oInfoStores = oSession.InfoStores
    For Each otmpInfoStore In oInfoStores
       Set oInfoStore = otmpInfoStore
       boolPFStore = CheckForPFStore
       If boolPFStore = True Then
          'We found the Public Folder infostore
          'MsgBox "Found PF"
          Exit For
       End If
    Next
    boolOffline = CheckOfflineStatus
    strFavEntryID = oInfoStore.Fields(ActMsgPR_IPM_FAVORITES_ENTRYID).Value
```

```
    Set oFavFolder = oSession.GetFolder(strFavEntryID, Null)
    'MsgBox oFavFolder.Name

    If (boolPFStore = True And boolOffline = False) Then
        'Add to Favorites
        On Error Resume Next
        oViewControl.AddToPFFavorites
    Else
        'MsgBox "not adding to favorites"
    End If
End Sub

Function CheckForPFStore()
    On Error Resume Next
    Err.Clear
    lMask = oInfoStore.Fields.Item(ActMsgPR_STORE_SUPPORT_MASK)
    If lMask And ActMsgSTORE_PUBLIC_FOLDERS Then
        'It's a Public Folder store
        'MsgBox "PF"
        CheckForPFStore = True
    Else
        'MsgBox "not a PF"
        CheckForPFStore = False
    End If
End Function

Sub AddOutlookShortcut()
    Set oPane = oExplorer.Panes("OutlookBar")
    Set oOLBarStorage = oPane.Contents
    Set oOLBarGroups = oOLBarStorage.Groups

    Set oNewOLGroup = oOLBarGroups.Add("Account Tracking (ext)", _
        oOLBarGroups.Count + 1)
    Set oNewOLShortcuts = oNewOLGroup.Shortcuts
    oNewOLShortcuts.Add oAccountFolder, "Account Tracking (ext)"
End Sub

Sub CreateHiddenMessage()
    On Error Resume Next
    'MsgBox "Creating Hidden Message"
    Set oInbox = oSession.Inbox
    Set oHidden = oInbox.HiddenMessages
    Set oNewMsg = oHidden.Add("Acct Ext: " & oFolder.Name, _
        "", "IPM.Post.AcctExt")
    oNewMsg.Fields.Add "strEntryID", 8, strEntryID
    oNewMsg.Update
End Sub
```

The code checks in the user's Inbox for the existence of a hidden message that matches the ID of the current folder. If the code finds that message, it doesn't attempt to add the folder to the user's Favorites folder or the Outlook bar. However, if the code doesn't find that message, it checks the values put into the global variables that the Team Folders extension added.

If the user is online and has indicated to the extension to add the folder to the user's Favorites folder, the code uses the *AddtoPFFavorites* method of the View control to do so. You check online and offline status in your code by using a property on the InfoStore object. Since the application must be in the Favorites folder to be accessible offline, the code does not attempt to add the folder to Favorites folder if the user is working offline.

The code then adds the folder to the Outlook shortcut bar by using the OutlookBarShortcuts collection of the Outlook object model. Next, the code uses the HiddenMessages collection of CDO in the user's Inbox to create a new item. This item will prevent future hits on the folder home page that would cause these actions to recur. This is a good way to store user customization or completion information because you know the Inbox will roam with the user no matter what computer he logs on from. As long as the user logs on to Exchange Server, you can access his Inbox and retrieve your hidden message.

REGISTERING YOUR EXTENSION

Now that you know how to create an extension, let's examine how to register one. You must register an extension for each template that will use it. Under the registry key for the template that will use the extension, you must provide two new values: *ExtensionClass* and *ExtensionSteps*. *ExtensionClass* is a string that specifies the ProgID of your extension DLL. *ExtensionSteps* is an integer that specifies the number of steps your extension will add to the wizard. You can have multiple steps in your extension, so this number is not limited to *1*. However, you will need to code the navigation for your extension in order to call your Visual Basic forms. For example, if your extension contains two Visual Basic forms, the first form must implement under its Next button the code to hide itself and to show the second Visual Basic form. Figure 9-24 shows the registration for the Account Tracking extension.

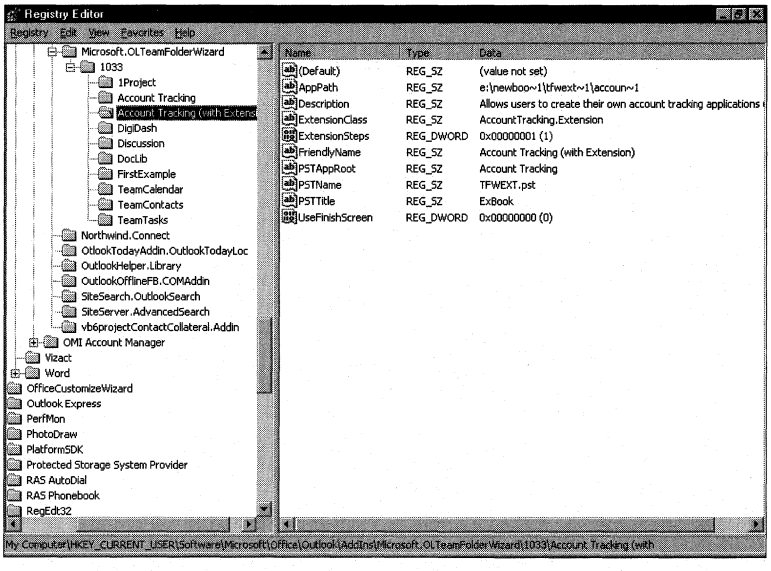


Figure 9-24. *The registration for the Account Tracking extension as part of a custom Team Folders template.*

DEPLOYING YOUR EXTENSION

Deploying your extension is no different than deploying a custom template. You need to register your ActiveX DLL on the user's computer and copy the DLL itself to the local hard disk. You also must add custom registry settings to the templates on the user's computer on which you want to use your extension. You have a number of choices for achieving these steps. You can distribute your extension to your users by employing Microsoft Systems Management Server, the Visual Basic Package and Deployment Wizard, or another tool. Systems Management Server provides the easiest and most powerful way to distribute your extension, but it requires that you have its infrastructure already established in your organization. If you used Visual Basic to create your extension, you'll probably use the Visual Basic Package and Deployment Wizard. This wizard is quite easy to use and can create customized setup programs for deploying your extension. For more information about this wizard, see the Visual Basic documentation.

Chapter 10

Outlook 2000 in Action: Enhancements to the Account Tracking Application

This chapter outlines Microsoft Outlook 2000 enhancements to the Account Tracking application. The main enhancement is the addition of a COM add-in that notifies users when new tasks or accounts are assigned to them. The implementation of this COM add-in will allow us to examine how to respond to events, how to add custom toolbar buttons and event handlers for those buttons, and how to add custom property pages to the Outlook environment. Before we discuss the COM add-in, we'll add two folder home pages to the Account Tracking application, the second of which uses the Outlook View control.

FOLDER HOME PAGES

Folder home pages are a feature in Outlook 2000 that enable you to link an HTML page to any folder in the Outlook environment. (You saw this capability in the Outlook Today feature discussed in Chapter 7.) Folder home pages support offline viewing capabilities, so you can request Outlook to synchronize an HTML file that is associated with a folder offline when a user synchronizes the folder. This ensures that your HTML page is available whether the user is working offline or online. We will look at two folder home pages in this chapter. One will use the Outlook View control.

The process of associating a folder home page with a folder is easy. Outlook 2000 provides a user interface for this connection, shown in Figure 10-1. You can access this Properties dialog box in Outlook 2000 by right-clicking on a folder and then choosing Properties.

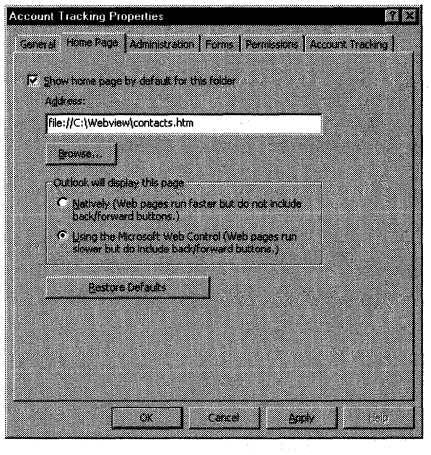


Figure 10-1. *Configuring a folder home page for a folder.*

Since Outlook 2000 hosts Microsoft Internet Explorer in-frame, when a user clicks on the folder, your folder home page can appear directly inside the Outlook client. Furthermore, you can make the folder home page the default view for a particular folder. In the Web pages for your folder home pages, you might want to include instructions for using the folder, a way to mail the folder owner, or a listing of links related to that folder or to other folders.

You can also add script to the folder home page to access the Outlook object model. Figure 10-2 shows the first example of a custom folder home page (Contacts.htm) for the Account Tracking application.

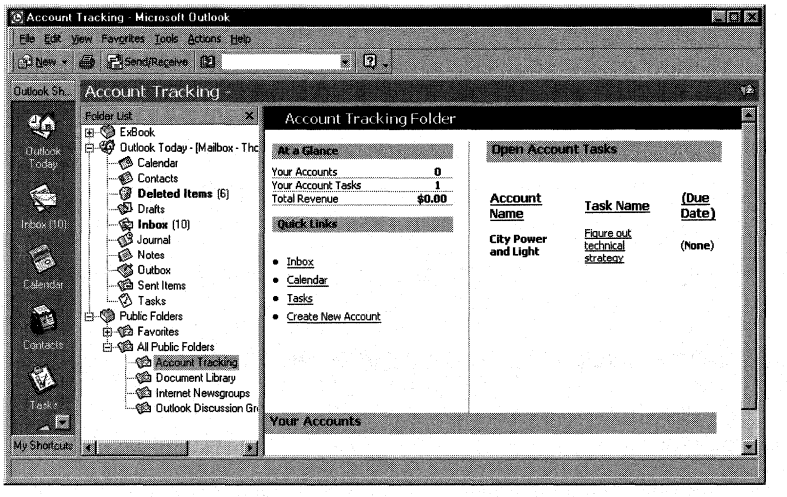


Figure 10-2. *The folder home page (Contacts.htm) for the Account Tracking application.*

Setting Up the First Folder Home Page

To test the folder home pages, you'll need a machine that has Outlook 2000 with the Visual Basic Scripting Support and Collaboration Data Objects components installed. Follow these steps to set up the first folder home page:

1. If you haven't already set up the Account Tracking application, do so now as explained in the section titled "Setting Up the Application" on page 157 in Chapter 6. If you want to use the Sales.mdb database, you will need to make an additional change to the code for the Account Tracking form. Since Office 2000 includes Data Access Objects (DAO) 3.6, you will need to change the line

   ```
   Set oDatabaseEngine = _
      item.application.CreateObject("DAO.DBEngine.35")
   ```

   to

   ```
   Set oDatabaseEngine = _
      item.application.CreateObject("DAO.DBEngine.36")
   ```

   If you want the application to create sales charts and print account summaries, install Microsoft Excel.

2. Copy the Webview folder from the companion CD to your local hard drive, and clear the read-only flag for the files contained in this folder.

3. Open the Webview\Contacts.htm file in Notepad.

4. Find the first occurrence of *oAccountFolder*, and modify the path to the location of your Account Tracking folder.

5. Save Contacts.htm, and close Notepad.
6. In Outlook, right-click on the Account Tracking folder, and choose Properties.
7. In the Address text box of the Home Page tab, specify the location of the Contacts.htm file—for example, *file://C:\Webview\Contacts.htm*.
8. Check the Show Home Page By Default For This Folder check box, and click OK.
9. Click the Account Tracking folder in Outlook to display the folder home page.

Example Script for the Folder Home Page

The following shows the script for the folder home page (Contacts.htm) displayed in Figure 10-2:

```
<SCRIPT ID=clientEventHandlersVBS LANGUAGE=vbscript>
'*****************************************************
'In-line code
'
'These lines of code are run when the browser reaches
'them while parsing the document. They set up the global
'variables that are needed throughout the application.
'*****************************************************
Set oApplication = window.external.OutlookApplication
Set oNS = oApplication.GetNameSpace("MAPI")

'Change this to your location for the Account Tracking folder
set oAccountFolder = oNS.Folders("Public Folders").Folders( _
   "All Public Folders").Folders("Account Tracking")

'Set some global vars for the EntryIDs
Dim arrTaskEntryIDs()
Dim oTasks       'Restricted collection of Tasks
Dim arrAccountEntryIDs()
Dim oAccounts    'Restricted collection of Accounts

'*****************************************************
'Sub CreateAccount
'
'This subroutine creates a new account info form and
'displays it for the user to fill in
'*****************************************************
Sub CreateAccount()
   set oAccount = oAccountFolder.Items.Add("IPM.Post.Account info")
   oAccount.Display()
End Sub
```

```
'*****************************************************
'Sub GetTask(lEntryID)
'
'This subroutine gets the task that the user clicked on
'in the HTML page and displays it. An index into an
'array of EntryIDs is passed to this subroutine.
'*****************************************************
sub GetTask(lEntryID)
   lTaskEntryID = arrTaskEntryIDs(lEntryID-1)
   for each oItem in oTasks
      if oItem.EntryID = lTaskEntryID then
         set otmpTask = oItem
      end if
   next
   otmpTask.Display()
end sub

'*****************************************************
'Sub GetAccount(lEntryID)
'
'This subroutine gets the account that the user clicked on
'in the HTML page and displays it. An index into an
'array of EntryIDs is passed to this subroutine.
'*****************************************************
sub GetAccount(lEntryID)
   lAccountEntryID = arrAccountEntryIDs(lEntryID-1)
   for each oItem in oAccounts
      if oItem.EntryID = lAccountEntryID then
         set otmpAccount = oItem
      end if
   next
   otmpAccount.Display()
end sub

sub Window_onLoad()
'*************************************************************
'All of the following lines are run when the HTML page is
'loaded
'*************************************************************
   'Put the name of the folder in the bar
   txtFolder.innerHTML = oAccountFolder.Name & " Folder"

'*************************************************************
'Figure out the account tasks for the current user
'*************************************************************
   RestrictString = ""
   RestrictString = "[Message Class] = ""IPM.Task""" & _
      " AND [Owner] = """ & oNS.CurrentUser.Name & """ AND _
```

(continued)

```
   [Complete] = FALSE"
Set oTasks = oAccountFolder.Items.Restrict(RestrictString)
oTasksCount = oTasks.Count
'Redim the EntryID array
ReDim arrTaskEntryIDs(oTasksCount-1)
strTaskList = "<TABLE Border=0 cellpadding=2 cellspacing=2 " & _
"class='calendarinfo'><TR><TD><strong><Font Size=2><U>" & _
"Account Name</u></STRONG></TD><TD>  </TD><TD>" & _
"<STRONG><Font size=2><U>Task Name</U></FONT></STRONG>" & _
"</TD><TD>  </TD><TD><STRONG><Font size=2><U>" & _
"(Due Date)</U></FONT></strong></TD></TR>"
'Count the tasks using counter
counter = 1
oTasks.Sort "[ConversationTopic]", False
For each oTask in oTasks
   boolOverDue = 0
   if oTask.DueDate = "1/1/4501" then
      strDueDate = "None"
   else
      strDueDate = oTask.DueDate
      'Check to see whether the task is overdue
      if DateDiff("d",CDate(strDueDate),Now) > 0 then
         boolOverDue = 1
      end if
   end if
   if boolOverDue then
      'Turn red
      strTaskList = strTaskList & "<TR><TD><FONT " & _
      "COLOR='#FF0000'><STRONG>" & oTask.ConversationTopic & _
      "</STRONG></FONT></TD><TD>  </TD><TD>" & _
      "<A HREF='' onclick=GetTask(" & counter & _
      ");window.event.returnValue=false>" & oTask.Subject & _
      "</a></TD><TD>  </TD><TD><FONT " & _
      "COLOR='#FF0000'>(<Strong>" & strDueDate & _
      "</Strong>)</FONT><BR></TD></TR>"
   else
      strTaskList = strTaskList & "<TR><TD><STRONG>" & _
      oTask.ConversationTopic & "</STRONG></TD><TD>" & _
      "  </TD><TD><A HREF='' onclick=GetTask(" & _
      "counter & ");window.event.returnValue=false>" & _
      oTask.Subject & "</a></TD><TD>  </TD>" & _
      "<TD>(<Strong>" & strDueDate & "</Strong>)<BR></TD></TR>"
   end if
   arrTaskEntryIDs(counter-1) = oTask.EntryID
   counter = counter + 1
next
TaskList.innerHTML = strTaskList & "</TABLE>"
```

```
'*************************************************************
'Figure out which accounts the current user is a team member of
'*************************************************************

   'Find accounts where this person is a team member.
   'First restrict to only account items.
   RestrictString = ""
   RestrictString = "[Message Class] = "IPM.Post.Account info""
   Set oAccounts = oAccountFolder.Items.Restrict(RestrictString)

   'Now find accounts where this person is a team member
   numFound = 0
   strCurrentUser = oNS.CurrentUser.Name
   numTotalRevenue = 0
   strAccountHTML = "<table border=0 width=100% cellpadding=3 " & _
   "cellspacing=0 ID='Home' style='DISPLAY: inline; " & _
   "MARGIN-TOP: 12px'>"
   strAccountHTML = strAccountHTML & "<TR><TD " & _
   "class='calendarinfo'><STRONG><FONT SIZE=2><U>Account Name" & _
   "</U></FONT></STRONG></TD></TR>"

   RestrictString = ""
   RestrictString = "[Message Class] = ""IPM.Post.Account info""" & _
   " AND [txtAccountConsultant] = """ & strCurrentUser & _
   """ OR [txtAccountExecutive] = """ & strCurrentUser & _
   """ OR [txtAccountSalesRep] = """ & strCurrentUser & _
   """ OR [txtAccountSE] = """ & strCurrentUser & _
   """ OR [txtAccountSupportEngineer] = """ & strCurrentUser & """"

   Set oAccounts = oAccountFolder.Items.Restrict(RestrictString)
   numFound = oAccounts.Count
   ReDim arrAccountEntryIDs(numFound)
   counter = 1
   For Each oAccount in oAccounts
      set oUserProps = oAccount.UserProperties
      arrAccountEntryIDs(counter-1) = oAccount.EntryID
      'Get the total revenue for the account for 1998 and 1999.
      'Get the revenue and add it to the total.
      num1998Total = oUserProps.Find("form1998ActualTotal")
      num1999Total = oUserProps.Find("form1999ActualTotal")
      if num1998Total <> "Zero" then
         numTotalRevenue = numTotalRevenue + num1998Total
      end if
      if num1999Total <> "Zero" then
         numTotalRevenue = numTotalRevenue + num1999Total
      end if
      strAccountHTML = strAccountHTML & "<TR><TD " & _
      "class='calendarinfo'><A Href='' onclick=GetAccount(" & _
```

(continued)

```
        "counter & ");window.event.returnValue=false>" & _
        oAccount.Subject & "</A></TD></TR>"
        counter = counter + 1
    next
    numTotalRevenue = CCur(numTotalRevenue)
    numTotalRevenue = FormatCurrency(numTotalRevenue)
    TotalRevenue.innerHTML = "<STRONG>" & numTotalRevenue & _
        "</STRONG>"
    strAccountHTML = strAccountHTML & "</TABLE>"
    Accounts.innerHTML = strAccountHTML
    YourTasks.innerHTML = "<Strong>" & oTasksCount & "</Strong>"
    YourAccounts.innerHTML = "<STRONG>" & numFound & "</STRONG>"
end sub   'Window_OnLoad
-->
</SCRIPT>
```

Looking at the code, you can see that you need to follow a few critical steps to access the Outlook object model. The first step is to retrieve the Outlook Application object. To do this, you use the *Window.External.OutlookApplication* syntax. Once you have the Application object, you can retrieve the rest of the Outlook objects. For example, by calling the *ActiveExplorer* method on the returned Application object, you can retrieve the Explorer object that is hosting the folder home page.

The folder home page is a dynamic environment for the viewing of a folder, as illustrated by the Account Tracking folder home page. In it, the user is presented with a summary of her accounts, account revenue, and tasks. The home page allows you to restrict account tasks to the person currently viewing the folder. (While you could create a similar view in Outlook, you wouldn't be able to specify a filter based on who is viewing the folder.) These account summaries are created by using the *Restrict* method on the Outlook Items collection for the folder. For the Tasks restriction, the code restricts only those messages in the folder that are tasks, where the current user is the owner and the task status is incomplete. After receiving the restricted set, the code sorts the tasks by their conversation topics, which are the names of the accounts the tasks are for. The code loops through each task to see whether it has a due date. If it does, the code checks to see whether the task is past due. Then the code generates the HTML, which will be placed in the Open Account Tasks list.

For the revenue summary, the code first finds all account items to tally a total. The code restricts the collection to all accounts for which the current user is a team member. Then the code loops through each account and retrieves the revenue, which is stored in the UserProperties collection as custom properties. Since the revenue properties are formula properties, they can contain text that indicates zero revenue from the account. To compensate for this, the code checks whether the value of the property is the string "*Zero*". The code then builds a string for the restricted list of account names and prints out the account revenue. The string of account names is hyperlinked, as are the tasks, so that a user can quickly go to a specific account or task.

From the code sample, you can see how you can include all the features of the Outlook object model, or any object model for that matter, inside the HTML page you create for your folders.

THE OUTLOOK VIEW CONTROL

The release of Outlook 2000 included an add-on product named the Outlook View control. The Outlook View control is an ActiveX wrapper around the Outlook views, such as the Table, Calendar, Card, and Timeline views. You can use this ActiveX control either inside a Web application, such as an Active Server Pages (ASP) application, or inside your folder home page. This control prevents you from having to rewrite significant portions of code to mimic Outlook functionality. Figure 10-3 shows the second folder home page example (FullContacts.htm) hosting the View control.

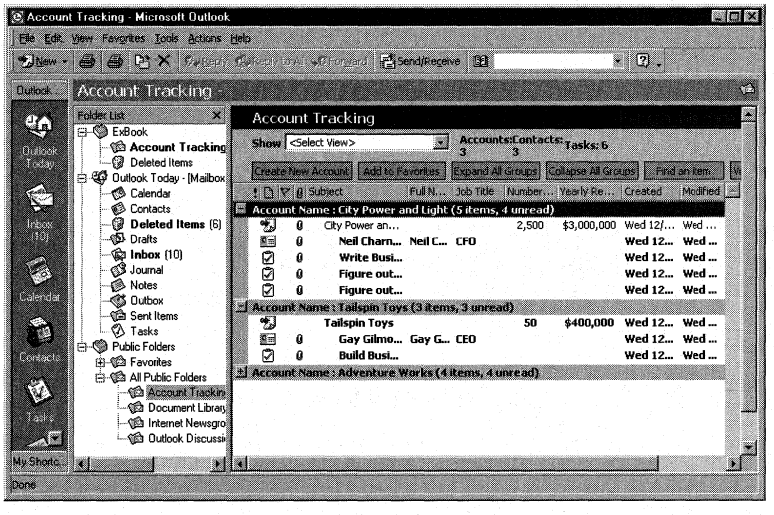


Figure 10-3. *A folder home page (FullContacts.htm) hosting the Outlook View control.*

The environment you place the View control in determines the control's functionality. For example, when you place the View control in a folder home page, the control provides full access to the Outlook object model as well as automatic merging of menu commands with the Outlook container, as shown in Figure 10-4. In contrast, in a stand-alone Web page scenario, the control does not allow access to any user data nor does it give you the entire Outlook object model. This restriction prevents the control from downloading all the Outlook data when a user accesses the Web page. In either scenario, the View control does require Outlook to be installed on the machine. The control does not install Outlook for you.

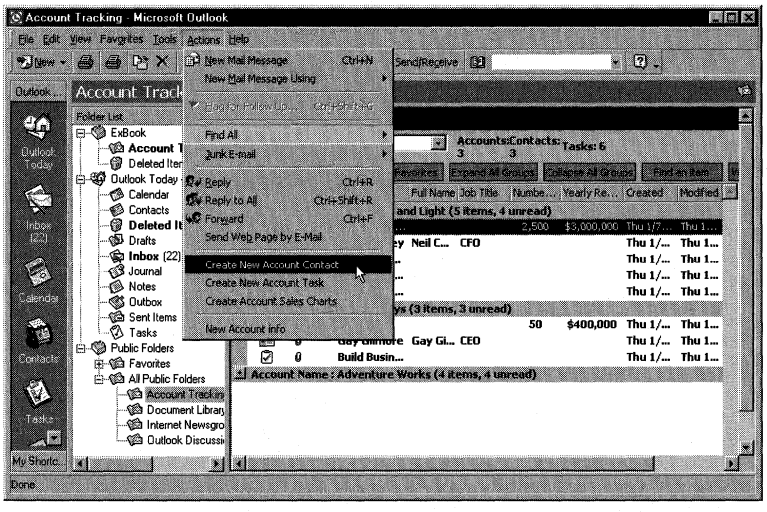


Figure 10-4. *When hosted in a folder home page, the View control automatically merges menu commands with its Outlook container. Notice that the custom actions appear in the Actions menu for an Account Tracking form selected in the control.*

The View control allows you to programmatically change control properties so that you can place more than one control on a single page in your application. For example, you might want to show a side-by-side view of two calendars, or maybe a contacts list and all tasks associated with the currently selected contact. When multiple View controls are on a single page, merging their menus is based on the control with the focus.

Setting Up the Second Folder Home Page

Using a machine that has Outlook 2000 with the Visual Basic Scripting Support and Collaboration Data Objects components installed, follow the next set of steps to set up the second folder home page, which uses the Outlook View control:

1. Make sure you have the Outlook View control installed. The easiest way to guarantee that the View control is installed on your machine is to install the Microsoft Outlook Team Folders Wizard.

   NOTE The Outlook View control shipped after Outlook 2000. Microsoft made the control available for downloading from the Outlook and Microsoft Exchange Server Web sites. The control is also available on the companion CD. To use it, install the Team Folders Wizard.

2. In Outlook, right-click on the Account Tracking folder and choose Properties.

3. In the Address text box of the Home Page tab, specify the location of the FullContacts.htm file—for example, *file://C:\Webview\FullContacts.htm*—and click OK. (FullContacts.htm is available on the companion CD.)
4. Click the Account Tracking folder to display the folder home page.

The following section outlines how to add the View control to a web page and how to program it.

Using the Outlook View Control

Adding a View control to your folder home page or web page is actually quite easy. All you need to do is add the <OBJECT> tag to your page and give the control an ID that you will use in your program. For the Account Tracking folder home page, this Object tag was inserted into the HTML page:

```
<OBJECT ID="oViewControl" WIDTH=100% HEIGHT=84%
style="border-bottom:1px silver solid"
CLASSID="CLSID:0006F063-0000-0000-C000-000000000046">
   <param NAME="Namespace" VALUE="MAPI">
   <param NAME="Folder" VALUE="">
   <param NAME="View" VALUE="Accounts">
</OBJECT>
```

This tag creates the View control object. Also notice the Param tags—you can use these tags to pass parameters to the control. In this example, I pass in *MAPI* for the *Namespace* parameter. I also pass in the folder. I pass a blank value for the folder so that the control defaults to the default folder the user is currently looking at. Finally, I pass in, as a string, the default view I want in the control. The Accounts view is the default view for the Account Tracking folder.

After you insert the control, you can add code in the folder home page to grab the Outlook Application object, Window.External.OutlookApplication, using the technique we saw earlier. Because I know this script must be running in the Account Tracking folder (as this is the folder home page for that folder), I set a variable to the current folder so that I can use that variable later in the script.

After the folder variable is set, my code needs to accomplish one more task. Recall that the View control is going to bring up the default folder that the user is viewing. This folder, however, might not be the Account Tracking folder. To ensure that the control displays the Account Tracking folder, the code finds the full path to the Account Tracking folder and passes this path as one of the control's properties, *Folder*. For example, if the Account Tracking folder were a top-level folder in the Favorites folders, the path would be \\Public Folders\Favorites\Account Tracking\. The code then fills in the total number of accounts, contacts, and tasks in the folder. The code for this process is shown on the following page.

```
<SCRIPT ID=clientEventHandlersVBS LANGUAGE=vbscript>

'*****************************************************
'In-line code
'
'These lines of code are run when the browser reaches
'them while parsing the document. They set up the global
'variables that are needed throughout the application.
'*****************************************************

Set oApplication = window.external.OutlookApplication
Set oNS = oApplication.GetNamespace("MAPI")
Set oCurrentFolder = oApplication.activeExplorer.currentfolder
Set oAccountFolder = oCurrentFolder
'AvailWidth = document.body.clientWidth

'*****************************************************
'Function StrFullPath
'
'This function creates and returns the full path to the
'folder
'*****************************************************
function StrFullPath()
   If oCurrentFolder Is Nothing Then
      strFolderName = ""
   End If
   Set olCollabFolder = oCurrentFolder
   strFolderName = ""
   Set olRoot = oCurrentFolder
   While (olRoot <> "Mapi")
      strFolderName = oCurrentFolder.Name & "\" & strFolderName
      Set olRoot = oCurrentFolder.Parent
      If olRoot <> "Mapi" Then
         Set oCurrentFolder = oCurrentFolder.Parent
      End If
   Wend
   strFullPath = "\\" & strFolderName
end function

'*****************************************************
'Sub FillTotals()
'
'This subroutine gets the count for the different types
'of items in a folder, such as accounts, contacts, and
'tasks. It also fills in the HTML page with this
'information.
```

```
'*************************************************
Sub FillTotals()
   RestrictString = ""
   RestrictString = "[Message Class] = ""IPM.Post.Account info"""
   Set oAccounts = oAccountFolder.Items.Restrict(RestrictString)
   oAcctCount = oAccounts.Count
   AccountTotal.innerHTML = "<STRONG>" & oAcctCount & "</STRONG>"
   RestrictString = ""
   RestrictString = _
      "[Message Class] = ""IPM.Contact.Account contact"""
   Set oContacts = oAccountFolder.Items.Restrict(RestrictString)
   oContactCount = oContacts.Count
   ContactTotal.innerHTML = "<STRONG>" & oContactCount & "</STRONG>"
   RestrictString = ""
   RestrictString = "[Message Class] = ""IPM.Task"""
   Set oTasks = oAccountFolder.Items.Restrict(RestrictString)
   oTasksCount = oTasks.Count
   TaskTotal.innerHTML = "<STRONG>" & oTasksCount & "</STRONG>"
End Sub

Fullpath = StrFullPath()
oViewControl.Folder = FullPath

'*************************************************
'Sub Window_onLoad()
'
'This subroutine is called when the HTML page is loaded
'*************************************************
Sub Window_onLoad()
   oViewControl.Folder = Fullpath
   'oViewControl.width = AvailWidth
   txtFolder.innerHTml = oAccountFolder.Name
   FillTotals()
End Sub
```

Now that some of the information for the HTML page is filled in, we need to add some buttons to the page to allow the user to call our subroutines, which automate the View control. I've left out the HTML code that actually creates the buttons (you can look at this code in the FullContacts.htm file on the companion CD), but we will take a look at the automation code that drives the View control from these buttons.

There are actually six buttons and a drop-down list from which the user can change the View control. The drop-down list enables the user to change the view of the control to one of the other views in the Outlook folder. Figure 10-5 shows another view of the Account Tracking folder home page.

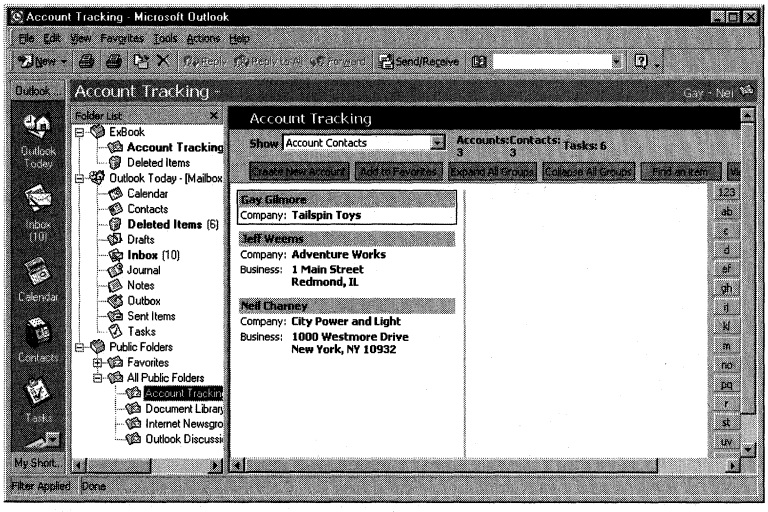


Figure 10-5. *The folder home page for the Account Tracking application, which contains the Outlook View control. This time the view is showing account contacts.*

The View control has no methods to create new views, so the views must already exist in the folder. To change the view by using code, set the *View* property on the control as the name of the desired new view. Since a fully functional Outlook application is running in the control, users can right-click on view columns to bring up the Field Chooser or customize the view directly in the page. The following code implements changing the views of the control:

```
'*****************************************************
'Sub WhatView_onChange
'
'This subroutine changes the view of the Outlook control
'depending on what the user picked in the drop-down list
'*****************************************************
Sub WhatView_onChange
   oViewControl.view = WhatView.value
   window.focus
end sub
```

Implementing the functionality for the buttons is actually pretty straightforward as well. From the buttons, the user can create new accounts, expand groups, collapse groups, add a folder to a favorites list, find an item in a folder, and view the address book. Most of these actions are already contained in the View control as methods. For example, to view the address book, all the code has to do is call the *AddressBook* method on the View control. Same thing for adding the folder to the favorites—all the code has to do is call the *AddtoFavorites* method on the View control. Here's the code for the buttons:

```
'*****************************************************
'Sub CreateAccount
'
'This subroutine creates a new account info form and
'displays it for the user to fill in
'*****************************************************
Sub CreateAccount()
   set oAccount = oAccountFolder.Items.Add("IPM.Post.Account info")
   oAccount.Display()
End Sub

'*****************************************************
'Sub Actions_onClick(Action)
'
'This subroutine executes the correct action depending
'on what the user picked in the Web page, such as
'finding an item, creating a new account, and so on.
'*****************************************************
Sub Actions_onClick(Action)
   Select Case Action
      case "AddressBook"
         oViewControl.AddressBook()
      case "AddtoFavorites"
         oViewControl.AddtoFavorites()
      case "ExpandAllGroups"
         oViewControl.ExpandAllGroups()
      case "CollapseAllGroups"
         oViewControl.CollapseAllGroups()
      case "AdvancedFind"
         set oExplorer = oApplication.ActiveExplorer
         set oCommandBar = _
            oExplorer.CommandBars.Item("Menu Bar")
         set oMenu = oCommandBar.Controls("Tools")
         set oAF = oMenu.Controls("Advanced Find...")
         oAF.Execute
      case "CreateAccount"
         CreateAccount()
   End Select
End Sub
```

You can also take advantage of other methods and properties in your applications that use the Outlook View control. To see a complete list of them, just add a reference to the Outlook View control either in the VBA that ships with Outlook 2000 or in Microsoft Visual Basic, and use the object browser. Most of the methods and properties are self-explanatory, such as the *ReplyInFolder* and *ReplyAll* methods. On the next page are a few of the more interesting properties and methods for the View control that we haven't discussed yet and that you can use in your code.

- *FlagItem method.* This method brings up the dialog box that flags an item with a reminder. It will not work unless the user has selected a valid item in the View control, such as a PostItem.
- *Categories method.* This method brings up the Categories dialog box in which the user can select item categories. This is the same dialog box that appears when the Categories button is clicked in an Outlook form.
- *CustomizeView method.* This method brings up the dialog box that lets a user select fields, sortings, the filter, the automatic formatting, and the grouping for the view. This is the same dialog box that is displayed by selecting the View\Current View\Customize Current View menu option.
- *ShowFields method.* This method brings up the Show Fields dialog box. Using this method, the user can quickly select the desired fields that the View control will display for the current view.
- *SynchFolder method.* This method attempts to synchronize the current folder in the background. (I use the word *attempts* because your program might call this method only to find that the connection to the Exchange server is not available.) Consider creating a button on your HTML form so that users can easily activate folder synchronization.
- *Restriction property.* This is a powerful property because it allows you to filter the items you want to display in your view. It takes the same string format as the *Restrict* method on the Items collection. For example, if you want to restrict the view so that only Task items appear, you would pass to the *Restriction* property the following string: *[Message Class] = "IPM.Task"*. You can also pass your restriction as a parameter by using the following syntax when creating your View control:

  ```
  <param NAME=Restrict VALUE="[Message Class] = 'IPM.Task'">
  ```

  By using the *Restriction* property, you can place two View controls on a single page and have one view control show a restricted set of items based on what the users pick in the other View control.

THE ACCOUNT TRACKING COM ADD-IN

In the rest of this chapter, we will look at a COM add-in for the Account Tracking application. We will start by looking at how the COM add-in works and then delve into the code that implements the COM add-in. The Account Tracking COM add-in has various features, which include the following:

- The add-in includes a property page that allows users to set different options.
- On startup, the add-in checks whether an Account Tracking group and a shortcut exist. If not, the COM add-in can automatically create the group and shortcut.
- When creating the shortcut, the add-in can enable a folder home page for the Account Tracking folder.
- The add-in includes custom buttons on the command bar.
- The add-in notifies users via e-mail when new tasks or accounts are assigned to them.
- The add-in notifies users when changes to an account are made.

Compiling and Registering the COM Add-In

To set up the Account Tracking COM add-in, we first have to compile and register it. To compile it, you will need a machine with Outlook 2000 and Visual Basic 6.0 installed. Follow these steps to compile the add-in:

1. Copy the Account Admin folder from the companion CD to your local hard drive, and clear the read-only flag for the files.
2. Open AccountPP.vbp in Visual Basic 6.0.
3. Make AccountPP.ocx. (This file is the ActiveX control property page. Compiling automatically registers it.)
4. Open AccountAdminDLL.vbp in Visual Basic 6.0.
5. Change the constant STRFOLDERHOMEPAGEPATH to the location of the FullContacts.htm file.
6. Search for the second occurrence of *oNS.Folders*, and change the statement

   ```
   Set oFolder = oNS.Folders("Public Folders").Folders( _
      "All Public Folders").Folders("Account Tracking")
   ```

   to the *location* of your Account Tracking folder.
7. Make AccountAdminDLL.dll.

   NOTE It might be necessary to specify the location of the AccountPP.ocx in the References dialog box. If AccountPP is displayed as MISSING in the References dialog box, uncheck it, click OK to close, reopen the References dialog box, and browse for the location of AccountPP.ocx.

8. Double-click on AccountAdminDLL.reg to add the appropriate entries in the registry.

9. Launch Outlook 2000.
10. From the Tools menu, select Options. Click on the Other tab, click the Advanced Options button, and then click the COM Add-Ins button. In the COM Add-Ins dialog box, make sure that AccountAdmin is checked as an available add-in.
11. Log off, and close Outlook.
12. Restart Outlook.

Testing the COM Add-In

To test some of the COM add-in options, you have to turn them on. To turn the options on, you use the new Account Tracking tab, as shown in Figure 10-6. This property page is the AccountPP control we compiled in Visual Basic 6.0. (Later in the chapter, we will examine how it is constructed.) To display this property page, choose Options from the Tools menu in Outlook 2000, and then click on the Account Tracking tab. Make sure all options are checked, and click OK. (The Account Tracking tab is also available from the folder Properties dialog box, which can be accessed by right-clicking on the Account Tracking folder and choosing Properties.)

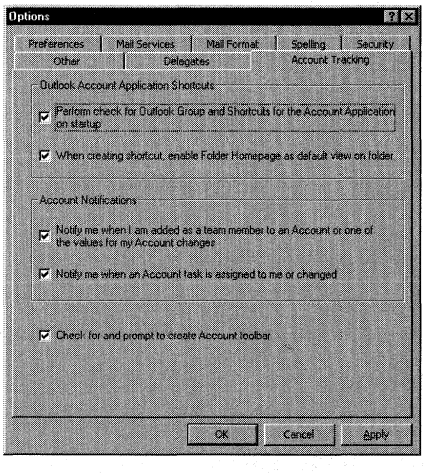


Figure 10-6. *The new Account Tracking tab in the Options dialog box. This tab makes it easy for users to select which features of the Account Tracking COM add-in they want to use.*

The Account Tracking settings selected by the user in either the Options or the Properties dialog boxes are automatically written to the registry so that the COM add-in can track the settings for each Outlook session. Figure 10-7 shows the registry with the user's settings for the COM add-in.

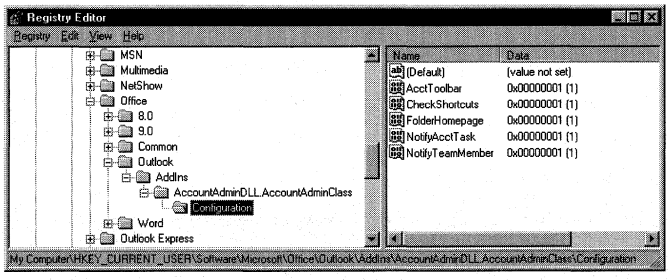


Figure 10-7. *The registry settings for the Account Tracking COM add-in. The add-in automatically persists the user settings to this portion of the registry.*

Now that the COM add-in is set up and the options are turned on, let's see how the add-in works. Select the Account Tracking folder in Outlook, and then choose New Account Info from the Actions menu. Fill out the information for a new account. Make sure you add yourself as a team member using your full name under the Account Team tab. Then click the Account Tasks tab, and add a new task.

> **NOTE** If you don't have the Outlook Visual Basic Scripting Support installed, a message indicating that scripting is not supported will be displayed when you try to display New Account Info. You can install the Outlook Visual Basic Scripting Support by re-running Setup for Outlook 2000.

If you left yourself as the task owner, once you add a new task, the COM add-in should send two e-mail messages to your Inbox. The first e-mail indicates that a new account has been created with you as a team member. The second e-mail indicates that a task was assigned to you. Figure 10-8 shows an example of the second e-mail.

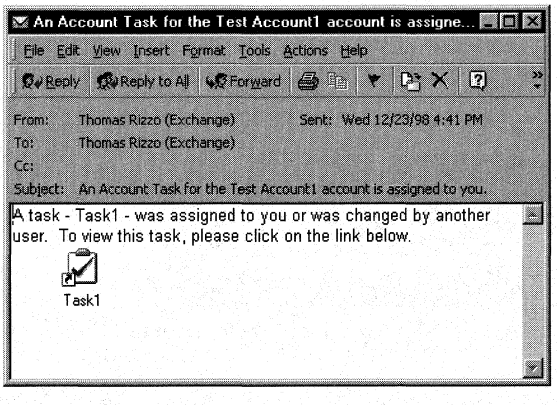


Figure 10-8. *An e-mail notification stating that a task either has been assigned to the current user or has changed in the event the current user is also the owner of the task.*

The COM add-in can notify a user when the user is added to the account team or, if the user is already a member of the account team, when the account information changes in the application. The information change notification can be triggered by changing the revenue of the account, the team members on the account, or the address of the account. Figure 10-9 shows the e-mail that is sent to the user when his account changes.

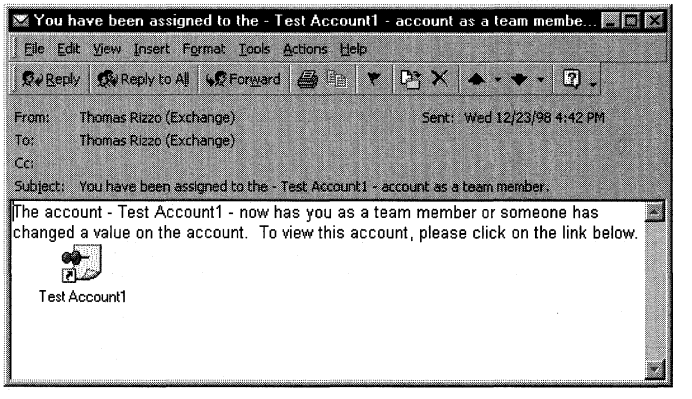


Figure 10-9. *An e-mail notification stating that account information has changed. This notification will be triggered only when the current user is a member of that account team.*

The Account Tracking COM add-in also includes the ability to automatically search for an Account Tracking Outlook group and shortcut. This option was activated on the Account Tracking tab of the Options dialog box. When the user starts Outlook and an Account Tracking group or shortcut does not exist, the user is prompted, as shown in Figure 10-10, about whether he wants to create a group or shortcut for the Account Tracking application. If the user chooses Yes, a new Account Tracking group and an Account Tracking shortcut are created on the Outlook Bar. If the Account Tracking Group dialog box is not displayed, try restarting Microsoft Windows and opening Outlook.

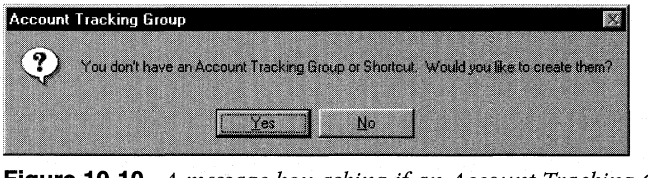


Figure 10-10. *A message box asking if an Account Tracking Group and Shortcut should be created.*

If when creating the shortcut, the check box named When Creating Shortcut Enable Folder Homepage As Default View On Folder is checked on the Account Tracking tab, the folder home page will be set as the default view for the folder. For the Account Tracking application, the folder home page is FullContacts.htm, which was specified earlier in the AccountAdminDLL project.

The COM add-in includes a new command bar and command buttons designed specifically for the Account Tracking application. After restarting Outlook, the COM add-in should display a message box asking if you want to create and display these new buttons, as shown in Figure 10-11. If the message box is not displayed, try clicking the newly created Account Tracking shortcut in the Account Tracking group.

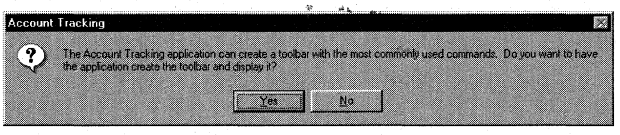


Figure 10-11. *A message box asking if the Account Tracking application should create new command buttons.*

The buttons make it easier for users to quickly create new accounts, contacts, or tasks. Figure 10-12 shows you the new command bar in Outlook 2000. If these buttons are not displayed, right-click on the command bar and select Account Tracking from the context menu.

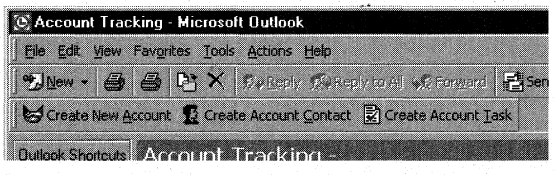


Figure 10-12. *A command bar created programmatically by the COM add-in.*

Implementing the COM Add-In

Let's review assumptions I made about implementation of the Account Tracking COM add-in. To make the code easier to digest, I assumed that users would always load the COM add-in at startup. For this reason, as you will see, the OnConnection event for the COM add-in is left pretty bare. Since loading at startup calls both the OnConnection and the OnStartupComplete events, most of the code is written in the OnStartupComplete event for simplicity. If you want to implement the ability to disconnect and reconnect the Account Tracking COM add-in, you will need to move some of the code out of the OnStartupComplete event into a subroutine, and then call that subroutine from both events. You will also need to revise the code to properly initialize some of the Outlook object variables.

The COM add-in options are stored in the registry. The registry path to these options is included in a code module with the COM add-in. Therefore, to change the registry location for the COM add-in options, you simply change this path.

Searching for the Account Tracking Group and Shortcut

The first section of the code we'll examine searches for the Account Tracking group and shortcut on the Outlook Bar. As mentioned earlier, this code occurs only when

the add-in is loaded and connected upon startup in Outlook 2000. The main portion of this code is implemented in the OnStartupComplete event for the COM add-in. Remember that you must place *Implements IDTExtensibility2* in your project before you can create code for this event. The following code is from the general declarations and the *OnStartupComplete* event procedure:

```
Implements IDTExtensibility2

Dim WithEvents oApplication As Outlook.Application
Dim WithEvents oNS As Outlook.NameSpace
Dim WithEvents oItems As Outlook.Items
Dim oFolder As Outlook.MAPIFolder
Dim WithEvents oExplorer As Outlook.Explorer
Dim oCommandBar As Office.CommandBar
Dim WithEvents oCreateAccountBHandler As Office.CommandBarButton
Dim WithEvents oCreateAcctContactBHandler As Office.CommandBarButton
Dim WithEvents oCreateAcctTaskBHandler As Office.CommandBarButton
Dim oAcctItem As Outlook.PostItem
Dim oFolders As Outlook.Folders
Dim prefLookForShortcuts As Integer
Dim prefMakeFolderHomepage As Integer
Dim prefNotifyWhenNewMember As Integer
Dim prefNotifyWhenNewTask As Integer
Dim prefEnableAcctToolbar As Integer
Dim oNewPage As Object

Const STRFOLDERHOMEPAGEPATH = _
"file://C:\Webview\fullcontacts.htm"
Private Sub IDTExtensibility2_OnStartupComplete(custom() As Variant)
   On Error Resume Next
   Set oNS = oApplication.GetNamespace("MAPI")
   'Replace with your folder location.
   'This is for offline users since they must put the folder
   'in their Favorites folder.
   Set oFolder = oNS.Folders("Public Folders"). _
      Folders("Favorites").Folders("Account Tracking")
   If oFolder Is Nothing Then
      'You may prefer to put in an EntryID here
      Set oFolder = oNS.Folders("Public Folders").Folders( _
         "All Public Folders").Folders("Account Tracking")
   End If
   If oFolder Is Nothing Then
      Set oFolders = oNS.Folders("Public Folders").Folders( _
         "All Public Folders").Folders

      'The following code can be used if you want to search
      'the entire public folder hierarchy for the folder.
      'For performance reasons, this code is commented out.
```

```
'     If Not (oFolders Is Nothing) Then
'        Set otmpFolder = oFolders.GetFirst
'        Do While Not (otmpFolder Is Nothing)
'           ListFolders otmpFolder
'           If otmpFolder Is Nothing Then
'              Set otmpFolder = oFolders.GetNext
'           Else
'              Exit Do
'           End If
'        Loop
'     End If
'     If you do use this code, you need to uncomment the
'     ListFolders subroutine as well as add a check here
'     to see if oFolder is nothing after finishing

      MsgBox "You have the Account Tracking COM add-in loaded" & _
         " but you have no Account Tracking folder. " & _
            "You may wish to unload the COM add-in.", _
            vbOKOnly + vbInformation, "Account Tracking COM add-in"
      Exit Sub
   End If
   Set oItems = oFolder.Items
   Set oExplorer = oApplication.ActiveExplorer
   'See if the user wants us to check for shortcuts
   If prefLookForShortcuts = 1 Then
      'The following code checks to see if the user has
      'an Outlook shortcut and group for the Account Tracking
      'application
      Dim oPane As OutlookBarPane
      Dim oOLBarStorage As OutlookBarStorage
      Dim oOLBarGroups As OutlookBarGroups
      Dim oOLBarGroup As OutlookBarGroup
      Dim oOLBarShortcuts As OutlookBarShortcuts
      Dim oOLBarShortcut As OutlookBarShortcut
      'Used if shortcut found with no group
      Dim otmpOLAccountBarGroupIndex As Integer
      Dim otmpOLBarGroup As OutlookBarGroup
      Dim otmpOLShortcuts As OutlookBarShortcuts
      Dim BarCounter As Integer
      Dim ShortcutCounter As Integer
      Set oPane = oExplorer.Panes("OutlookBar")
      Set oOLBarStorage = oPane.Contents
      Set oOLBarGroups = oOLBarStorage.Groups
      boolFoundAcctGroup = 0
      boolFoundAcctShortcut = 0
      BarCounter = 1
         For Each oOLBarGroup In oOLBarGroups
```

(continued)

```
        'For debugging purposes
            'MsgBox "Group: " & oOLBarGroup.Name
            If oOLBarGroup.Name = "Account Tracking" Then
               boolFoundAcctGroup = BarCounter
            End If
            Set oOLBarShortcuts = oOLBarGroup.Shortcuts
            ShortcutCounter = 1
            For Each oOLBarShortcut In oOLBarShortcuts
               'For debugging purposes
               'MsgBox oOLBarShortcut.Name
               Err.Clear
               If IsObject(oOLBarShortcut.Target) Then
                  'Check to see if this is the file target by
                  'checking error
                  If oOLBarShortcut.Target.Name = _
                  "Account Tracking" Then
                     If Err.Number = -2147319765 Then
                        'File Target
                     ElseIf Err.Number = 0 Then
                        'For Debugging purposes
                        'MsgBox _
                        "Account Tracking Folder: " & _
                        oOLBarShortcut.Target.Name
                        boolFoundAcctShortcut = _
                           ShortcutCounter
                        otmpOLAccountBarGroupIndex = _
                           BarCounter
                     End If
                  End If
               Else
                  'The target is a URL string
               End If
               ShortcutCounter = ShortcutCounter + 1
            Next
            BarCounter = BarCounter + 1
        Next
        'For debugging purposes
        'MsgBox boolFoundAcctShortcut & boolFoundAcctGroup.
        'Check to see whether shortcut exists without group.
        If (boolFoundAcctShortcut <> 0) And _
        (boolFoundAcctGroup = 0) Then
            'Check to see whether they want to remove the
            'shortcut and move it to a new group
            Response = MsgBox("You have an Account " & _
               "Tracking shortcut without an Account " & _
               "Tracking group. Would you like to create a " & _
               "new Account Tracking group and move the " & _
               "Account Tracking shortcut there?", & _
```

```
      vbYesNo + vbQuestion, "Account Tracking")
      If Response = vbYes Then
         'Delete the old Account Tracking shortcut.
         'Get the Outlook Bar for the shortcut.
         Set otmpOLBarGroup = _
            oOLBarGroups.Item(otmpOLAccountBarGroupIndex)
         Set otmpOLShortcuts = otmpOLBarGroup.Shortcuts
         otmpOLShortcuts.Remove boolFoundAcctShortcut
         Dim otmp2OLBarGroup As OutlookBarGroup
         Dim otmp2OLShortcuts As OutlookBarShortcuts
         'Create a new Account Tracking group
         Set otmp2OLBarGroup = oOLBarGroups.Add( _
            "Account Tracking", oOLBarGroups.Count + 1)
         'For debugging purposes
         'MsgBox "Group: " & otmp2OLBarGroup.Name
         Set otmp2OLShortcuts = otmp2OLBarGroup.Shortcuts
         otmp2OLShortcuts.Add oFolder, "Account Tracking"
         'Check to see whether they want us to create a
         'Web view
         If prefMakeFolderHomepage = 1 Then
          . 'Create the Web view
            oFolder.WebViewAllowNavigation = True
            oFolder.WebViewOn = True
            oFolder.WebViewURL = STRFOLDERHOMEPAGEPATH
         End If
      End If
   'Check to see whether group exists with no shortcut
   ElseIf (boolFoundAcctShortcut = 0) And _
   (boolFoundAcctGroup <> 0) Then
      'See if user wants to add shortcut to group
      Response = MsgBox("There is an Account " & _
         "Tracking Group without a shortcut to the " & _
         "Account Tracking folder. Do you want " & _
         "to add a shortcut to the Account Tracking " & _
         "folder in this group?", _
         vbYesNo + vbQuestion, "Account Tracking")
      If Response = vbYes Then
         Dim otmpOLGroup As OutlookBarGroup
         Set otmpOLGroup = _
            oOLBarGroups.Item(boolFoundAcctGroup)
         'For debugging purposes
         'MsgBox otmpOLGroup.Name
         Set otmpOLShortcuts = otmpOLGroup.Shortcuts
         otmpOLShortcuts.Add oFolder, "Account Tracking"
         'Check to see whether user wants us to create a
         'Web view
         If prefMakeFolderHomepage = 1 Then
```

(continued)

```
                    'Create the Web view
                    oFolder.WebViewAllowNavigation = True
                    oFolder.WebViewOn = True
                    oFolder.WebViewURL = STRFOLDERHOMEPAGEPATH
                End If
            End If
         'Check to see whether there is neither
         ElseIf (boolFoundAcctGroup = 0) And _
         (boolFoundAcctShortcut = 0) Then
            Response = MsgBox("You don't have an Account " & _
            "Tracking Group or Shortcut. Would you like to " & _
            "create them?", vbYesNo + vbQuestion, _
               "Account Tracking Group")
            If Response = vbYes Then
               Set otmpOLGroup = oOLBarGroups.Add( _
                  "Account Tracking", oOLBarGroups.Count + 1)
               'For debugging purposes
               'MsgBox otmpOLGroup.Name
               Set otmpOLShortcuts = otmpOLGroup.Shortcuts
               otmpOLShortcuts.Add oFolder, "Account Tracking"
               'Check to see whether user wants us to create a
               'Web view
               If prefMakeFolderHomepage = 1 Then
                  'Create the Web view
                  oFolder.WebViewAllowNavigation = True
                  oFolder.WebViewOn = True
                  oFolder.WebViewURL = STRFOLDERHOMEPAGEPATH
               End If
            End If
         'There is one other scenario with an Account Tracking
         'shortcut and an Account Tracking group.
         'In this scenario, do nothing.
      End If
   End If
End Sub
Sub ListFolders(objFolder)
'  On Error Resume Next
'  If Not (objFolder Is Nothing) Then
'     'Check to see whether Account Tracking folder
'     If objFolder.Name = "Account Tracking" Then
'        Set oFolder = objFolder
'        Exit Sub
'     Else
'        'Check for child folders
'        Set objFolders = objFolder.Folders
'        Set objFolder = objFolders.GetFirst
'        Do While Not (objFolder Is Nothing)
```

```
'           ListFolders objFolder
'           Set objFolder = objFolders.GetNext
'        Loop
'     End If
'  End If
End Sub
```

The first task the code performs is to set some of the variables in the application to their correct values. The commented parts in the code show how you can search for the Account Tracking public folder in the public folder hierarchy. Obviously, if the public folder hierarchy is large, completing the search could take a long time, and you might decide not to implement the code. Instead, you could replace the code in the event with code that retrieves the Account Tracking folder by EntryID, which allows you to eliminate any hard-coded paths to the folder.

The code then checks to see whether the user wants to look for the Account Tracking group and shortcut. This configuration information is pulled from the registry, as will be explained later in this section. If the user wants to search for the group and the shortcut, the code uses some of the new objects and collections in the Outlook 2000 object model.

The code grabs the OutlookBar pane from the current Explorer. Then the code retrieves the OutlookBarStorage object for the contents of that pane, and it retrieves the OutlookBarGroups in the storage object. From there, the code uses a For...Each loop to find the Account Tracking group.

> **NOTE** You could replace the For...Each loop code with a simpler version that uses the *Item* method on the OutlookBarGroups collection. By using the *Item* method, you can retrieve the group by name. But to show you how to use this collection, I used the For...Each loop.

In the For...Each loop, the code retrieves each shortcut in the group by using the OutlookBarShortcuts collection. The code then loops through each shortcut to determine whether the target for that shortcut is the Account Tracking folder by using the Account Tracking name. You could also compare the EntryIDs of the target with the original folder we set earlier in the code. It's really your choice how you want to implement this.

You can see some error handling code in the *IDTExtensibility2_OnStartupComplete* procedure. We have to make sure we do not error-out on file system targets. This error-handling code skips file system targets.

Counters in the code let the application know the index of the Account Tracking group as well as the index of the Account Tracking shortcut within that group, if the shortcut exists. The code uses these counters to check a number of scenarios, such as whether both the group and the shortcut exist. This checking scenario occurs when both counters are *0*. The code also checks to see whether the shortcut exists but no group does. If the shortcut does exist, the code can create a new group, remove the

existing shortcut, and place the shortcut in the new group. The code counts where the shortcut exists in a certain group to simplify removing the shortcut using its index. If the group exists without a shortcut, the code can create a shortcut in the group and associate the shortcut with the folder.

The code also checks the option settings to see whether it should make the default view on the folder the folder home page. If the check box that enables the folder home page on the Account Tracking tab of the Options dialog box is checked, the code uses the *WebViewAllowNavigation*, *WebViewOn*, and *WebViewURL* properties to set up the folder home page. *WebViewAllowNavigation* returns or sets the navigation mode for the folder if the user is viewing a folder home page. When this property is set to True, Outlook allows users to navigate using the Forward and Back buttons of the Microsoft Web Control. When this property is set to False, Outlook displays the folder home page in Native mode, which makes the Forward and Back buttons unavailable. By setting this property to True, the folder home page provides more functionality to the user, although it runs a bit slower.

The *WebViewOn* property returns or sets the folder home page state. If you set this property to True, as is done in the preceding code, Outlook displays the folder home page as the default view for the folder.

The *WebViewURL* property returns or sets the string that identifies the URL for the folder home page. Any valid URL can be used in this property, such as a file or an http URL. The application sets this property to a constant string, which is set in the declarations section of the program.

Using Events to Notify Users of Changes

The next section of the code we will take a look at tracks when users add or change account or task items in the folder. The application does not track deleted items since Outlook does not pass the deleted item in its ItemRemove event, making it difficult to figure out what was removed from the folder.

To track additions and changes to items, the code declares a variable *oItems* as an Outlook.Items collection by using the WithEvents keyword. The WithEvents keyword allows you to select the events you want to handle in the Visual Basic environment for the collection. The code for this application implements the ItemAdd and the ItemChange events for the Items collection. Let's first review the ItemAdd event, which is shown here:

```
Private Sub oItems_ItemAdd(ByVal Item As Object)
   Dim oUser As Variant
   Dim oMail As Outlook.MailItem
   Dim oAttach As Outlook.Attachment
   Dim oItem As Outlook.TaskItem
   Dim oAccountItem As Outlook.PostItem
   Dim oUserProps As Outlook.UserProperties

   oUser = oNS.CurrentUser.Name
```

```
'Check to see what type of item was just created
If Item.Class = olTask Then
   'Check to see whether user wants notification
   If prefNotifyWhenNewTask = 1 Then
      'Transform into TaskItem
      Set oItem = Item
      'Check to see whether the current user is the owner
      If oItem.Owner = oUser Then
         'Send to the user a message with a link to the item
         Set oMail = oApplication.CreateItem(olMailItem)
         With oMail
            .To = oUser
            .Subject = "New Account Task for the " & _
               Item.ConversationTopic & _
               " account is assigned to you."
            .Body = "A new task - " & Item.Subject & _
               " - was assigned to you.  " & _
               "To view this task, please click" _
               & " on the link below."
         End With
         Set oAttach = oMail.Attachments.Add(Item, _
            olEmbeddeditem)
         oMail.Recipients.ResolveAll
         oMail.Send
      End If
   End If
ElseIf Item.MessageClass = "IPM.Post.Account info" Then
   'Check to see whether user wants notification
   If prefNotifyWhenNewMember = 1 Then
      Set oAccountItem = Item
      boolAccountMember = 0
      Set oUserProps = oAccountItem.UserProperties
      If oUserProps.Find("txtAccountConsultant") = oUser Then
         boolAccountMember = 1
      ElseIf oUserProps.Find("txtAccountExecutive") = _
         oUser Then
         boolAccountMember = 1
      ElseIf oUserProps.Find("txtAccountSalesRep") = _
         oUser Then
         boolAccountMember = 1
      ElseIf oUserProps.Find("txtAccountSE") = oUser Then
         boolAccountMember = 1
      ElseIf oUserProps.Find("txtAccountSupportEngineer") = _
      oUser Then
         boolAccountMember = 1
      End If
      If boolAccountMember = 1 Then
         'Send to the user a message with a link to the item
```

(continued)

```
            Set oMail = oApplication.CreateItem(olMailItem)
            With oMail
               .To = oUser
               .Subject = "A New Account - " & _
                  Item.ConversationTopic & _
                  " - has been created with you as a " & _
                  "team member."
               .Body = "A new account - " & Item.Subject & _
                  " - was created with you as a team member." _
                  & " To view this account, please click" _
                  & " on the link below."
            End With
            Set oAttach = oMail.Attachments.Add(Item, _
               olEmbeddeditem)
            oMail.Recipients.ResolveAll
            oMail.Send
         End If
      End If
   End If
End Sub
```

The *oItems_ItemAdd* event procedure first retrieves the name of the current user. Then the code checks the class of the item that was added to the collection in the folder. If the item is a task, the code coerces the item into an Outlook TaskItem object before it attempts to call the methods and properties on that object type. If the current user is the owner of the new task, the application creates an e-mail message with the new task attached as a shortcut, and then sends the e-mail to the user. The user receives the notification e-mail in the Inbox.

The code used to notify users of a new account in the folder is similar to the task code, but instead of checking the owner property, the account code checks the item's custom properties that correspond to the names of the team members for the account. If the user is found in one of these properties, the code sends an e-mail to the user indicating that she has a new account for which she is a team member.

The only aspect of this subroutine and the next subroutine that you might want to change is the user who sends the item. In the current implementation, the user sends the update. You can change this functionality so that the public folder is the sender of the message by giving your users Send On Behalf Of permissions in your Exchange Administrator program for the folder. Then either expose the folder in the address list so that it can be added into the From field, or place the address of the folder in the From field. If you don't want the e-mail to come from the folder, you can create a mailbox and assign it Send On Behalf Of permissions in the Exchange Administrator program.

To notify the user that she has been assigned an existing task or that a task for which she is the owner has changed, the code uses the ItemChange event for the Items

collection. This event is also used to notify the user when she is added to an account team after the account is created, or when an account for which she is a team member has been changed. The following code implements the ItemChange event:

```
Private Sub oItems_ItemChange(ByVal Item As Object)
   Dim oUser As Variant
   Dim oMail As Outlook.MailItem
   Dim oAttach As Outlook.Attachment
   Dim oTaskItem As Outlook.TaskItem
   Dim oAccountItem As Outlook.PostItem
   Dim oUserProps As Outlook.UserProperties

   'Since the event doesn't show us how the item changed,
   'we need to notify the user of the change but not what
   'specifically changed on the item

   oUser = oNS.CurrentUser.Name
   'Check to see what type of item was just created
   If Item.Class = olTask Then
      'Check to see whether the user wants to be notified
      If prefNotifyWhenNewTask = 1 Then

         'Transform into TaskItem
         Set oTaskItem = Item

         'Check to see whether the current user is the owner
         If oTaskItem.Owner = oUser Then
            'Send to the user a message with a link to the item
            Set oMail = oApplication.CreateItem(olMailItem)
            With oMail
               .To = oUser
               .Subject = "An Account Task for the " & _
                  Item.ConversationTopic & _
                  " account is assigned to you."
               .Body = "A task - " & Item.Subject & _
                  " - was assigned to you or was changed " & _
                  "by another user.  " _
                  & "To view this task, please click" _
                  & " on the link below."
            End With
            Set oAttach = oMail.Attachments.Add(Item, _
               olEmbeddeditem)
            oMail.Recipients.ResolveAll
            oMail.Send
         End If
      End If
   ElseIf Item.MessageClass = "IPM.Post.Account info" Then
      Set oAccountItem = Item
```

(continued)

```
            boolAccountMember = 0
            Set oUserProps = oAccountItem.UserProperties
            If oUserProps.Find("txtAccountConsultant") = oUser Then
               boolAccountMember = 1
            ElseIf oUserProps.Find("txtAccountExecutive") = oUser Then
               boolAccountMember = 1
            ElseIf oUserProps.Find("txtAccountSalesRep") = oUser Then
               boolAccountMember = 1
            ElseIf oUserProps.Find("txtAccountSE") = oUser Then
               boolAccountMember = 1
            ElseIf oUserProps.Find("txtAccountSupportEngineer") = _
            oUser Then
               boolAccountMember = 1
            End If
            If boolAccountMember = 1 Then
               'Send to the user a message with a link to the item
               Set oMail = oApplication.CreateItem(olMailItem)
               With oMail
                  .To = oUser
                  .Subject = "You have been assigned to the - " & _
                     Item.ConversationTopic & _
                     " - account as a team member."
                  .Body = "The account - " & Item.Subject & _
                     " now has you as a team member or someone " & _
                     "has changed a value " _
                     & "on the account. To view this account, " _
                     & "please click on the link below."
               End With
               Set oAttach = _
                  oMail.Attachments.Add(Item, olEmbeddeditem)
               oMail.Recipients.ResolveAll
               oMail.Send
            End If
         End If
      End Sub
```

The code that handles the ItemChange event is very similar to the code for ItemAdd, so I won't cover it in detail. The only difference between the two event handlers is the text of the message that they send to the user. Since Outlook does not pass the property that was changed on the item as a parameter to ItemChange, the code can't know whether the user was assigned to an item or which property was changed. For this reason, the message text notifies the user only that a change to the item has occurred.

Adding and Handling Custom Command Bars and Buttons

The next section of code we'll take a look at adds a custom command bar and command buttons to the Outlook toolbar, and provides event handlers for the buttons when users click them. The code for this functionality is shown here:

```
Private Sub oExplorer_BeforeFolderSwitch( _
```

```
ByVal NewFolder As Object, Cancel As Boolean)
'Add CommandBar buttons to the Outlook User Interface for easy
'creation
Dim oTempFolder As Outlook.MAPIFolder
Dim oCommandBars As Office.CommandBars
Dim oCommandBar2 As Office.CommandBar
Dim oControls As Office.CommandBarControls
Dim oControl As Office.CommandBarButton
Dim otmpCommandBar As Office.CommandBar

'Make sure they want to do this
If prefEnableAcctToolbar = 1 Then
   'First check to see whether the folder is the
   'Account Tracking folder
   If Not (NewFolder Is Nothing) Then
      Set oTempFolder = NewFolder
      boolFoundCommandBar = 0
      'You might want to put in the EntryID here rather than
      'the name
      If oTempFolder.Name = "Account Tracking" Then
         'Check to see whether command bar already exists
         Set oCommandBars = _
            oApplication.ActiveExplorer.CommandBars
         For Each oCommandBar In oCommandBars
            If oCommandBar.Name = "Account Tracking" Then
               boolFoundCommandBar = 1
               Set otmpCommandBar = oCommandBar
               Exit For
            End If
         Next
         If boolFoundCommandBar = 0 Then
            'Need to create the command bar
            'Maybe add text box for searching for account
            'or contacts
            Response = MsgBox("The Account Tracking " & _
               "application can create a toolbar with " _
               & "the most commonly used commands. Do " & _
               you want to have the application create" _
               & " the toolbar and display it?", _
               vbYesNo + vbQuestion, "Account Tracking")
            If Response = vbYes Then
               'Create the command bar
               Set oCommandBar = oCommandBars.Add( _
                  "Account Tracking", Temporary:=False)
               Set oControls = oCommandBar.Controls
               'Create the buttons, and set the
               'event handler objects to the
               'buttons.
```

(continued)

```
            'Create the first button.
            Set oControl = oControls.Add( _
               Type:=msoControlButton, ID:=1, _
               Temporary:=False)
            oControl.Caption = "Create New &Account"
            oControl.FaceId = 609
            oControl.Style = msoButtonIconAndCaption
            Set oCreateAccountBHandler = oControl
            'Create the second button
            'Context menu
            Set oControl = oControls.Add( _
               Type:=msoControlButton, ID:=1, _
               Temporary:=False)
            oControl.Caption = "Create Account &Contact"
            oControl.FaceId = 607
            oControl.Style = msoButtonIconAndCaption
            Set oCreateAcctContactBHandler = oControl
            'Create the third button
            'Context menu
            Set oControl = oControls.Add( _
               Type:=msoControlButton, ID:=1, _
               Temporary:=False)
            oControl.Caption = "Create Account &Task"
            oControl.FaceId = 329
            oControl.Style = msoButtonIconAndCaption
            Set oCreateAcctTaskBHandler = oControl
            'Make the command bar visible
            oCommandBar.Visible = True
            oCommandBar.Position = msoBarTop
         End If
      Else
         'Account Tracking command bar already exists.
         'See if they want to do this.
         If prefEnableAcctToolbar = 1 Then
            'Check to see if visible; if not, make
            'visible
            Dim oCBControls As Office.CommandBarControls
            Dim oCBButton As Office.CommandBarButton
            If otmpCommandBar.Enabled = False Then
               otmpCommandBar.Enabled = True
            End If
            If otmpCommandBar.Visible = False Then
               otmpCommandBar.Visible = True
            End If
         End If
      End If
   Else
      'It's not the Account Tracking folder.
```

```
            'Look for the toolbar and disable it.
            On Error Resume Next
            Set oCommandBars = oApplication.ActiveExplorer. _
               CommandBars
            Set oCommandBar = oCommandBars("Account Tracking")
            oCommandBar.Enabled = False
         End If
      Else
         'It's a file system folder!
         'Disable toolbar.
         On Error Resume Next
         Set oCommandBars = _
            oApplication.ActiveExplorer.CommandBars
         Set oCommandBar = oCommandBars("Account Tracking")
         oCommandBar.Enabled = False
      End If
   End If
   Set oTempFolder = Nothing
   Set oCommandBars = Nothing
   Set oCommandBar = Nothing
   Set oControls = Nothing
   Set oControl = Nothing
End Sub
```

The application includes the Outlook Explorer object's BeforeFolderSwitch event, as shown in the preceding code. The *oExplorer_BeforeFolderSwitch* event procedure is passed, as a MAPIFolder, the folder that the user is trying to switch to. The code checks the folder's name to see if it is the Account Tracking folder. You could also perform this comparison by using the EntryID of the folder.

If the folder is the Account Tracking folder, the code searches the CommandBars collection of the Explorer object to see whether an Account Tracking command bar exists. If the code finds the Account Tracking command bar, it simply makes the command bar visible.

If the code doesn't find the Account Tracking command bar, it creates the command bar if the user selected to do this as a preference. The code adds a new CommandBar object to the CommandBars collection by passing the name of the command bar as well as the *Temporary* parameter. The *Temporary* parameter indicates that Outlook should persist the command bar between Outlook sessions. Then the code starts creating the buttons on the command bar.

To create the buttons, the code uses the Controls collection on the CommandBar object. The code then adds three button controls to the collection. The control type is identified with the msoControlButton constant. (You can create other types of controls on your command bars besides buttons, such as drop-downs, combo boxes, and popups.) The code also passes an ID of *1* for all the controls; this value indicates that the control is a custom control and not built in. The code passes the *Temporary*

parameter and sets it to False so that Outlook persists the buttons between sessions. Depending on the type of control you specify, the *Add* method will return an appropriate object, such as a CommandBarButton, a CommandBarComboBox, or a CommandBarPopup object.

After the code receives the CommandBarButton object from the *Add* method on the Controls collection, it starts setting properties on the CommandBarButton object. The first property it sets is the *Caption* property, which is a string containing the caption text for the control. Notice how you can place an ampersand before one of the letters in the control caption to provide a shortcut key to the control. This caption property is the default screen tip for the control.

The second property the code sets is *FaceId*, which specifies how the button face should look. Office 2000 has a number of built-in faces that you can use. If you want to use a custom face on your buttons, you must specify a 0 for this property and copy your custom face to the clipboard. Then you can use the *PasteFace* method on the CommandBarButton object to paste the face from the clipboard onto your control.

The final property the code sets is the *Style* property. This property can have many different values, such as the msoButtonIconAndCaption constant, which displays the button face as well as the caption text. Or you could choose msoButtonCaption to display only the caption. To enhance usability of the buttons, the code displays both the icon and the caption in them. For a list of all the style values, refer to the Office 2000 help file.

After the new CommandBarButtons are created and set, they are assigned to other variables such as *oCreateAccountBHandler*. If you take a look at the declarations section of the code earlier in the chapter, you'll notice that *oCreateAccountBHandler* is declared as an Office.CommandBarButton using the WithEvents keyword. The WithEvents keyword specifies that *oCreateAccountBHandler* is used to respond to events for a CommandBarButton. The following code shows the event handlers for the three buttons on the Account Tracking command bar:

```
Private Sub oCreateAccountBHandler_Click(ByVal Ctrl As _
Office.CommandBarButton, CancelDefault As Boolean)
   Dim oAccount As Outlook.PostItem
   Set oAccount = oFolder.Items.Add("IPM.Post.Account info")
   oAccount.Display
End Sub

Private Sub oCreateAcctContactBHandler_Click(ByVal Ctrl As _
Office.CommandBarButton, CancelDefault As Boolean)
   Dim oSelection As Outlook.Selection

   On Error Resume Next
   boolFoundAccountItem = 0
   Set oSelection = oExplorer.Selection
   For Each oItem In oSelection
```

```
      If oItem.MessageClass = "IPM.Post.Account info" Then
         boolFoundAccountItem = boolFoundAccountItem + 1
         'Set the item found to a global variable just in case
         'it is the only one found
         Set oAcctItem = oItem
      End If
   Next
   If boolFoundAccountItem = 0 Then
      MsgBox "You have no account selected. Please select " & _
         "an account and try again.", _
         vbOKOnly + vbExclamation, "Create Contact"
      Exit Sub
   ElseIf boolFoundAccountItem > 1 Then
      MsgBox "You have more than one account selected. " & _
         "Please select only one account and try again.", _
         vbOKOnly + vbExclamation, "Create Contact"
      Exit Sub
   ElseIf boolFoundAccountItem = 1 Then
      Set AccountContactForm = oAcctItem.Actions( _
         "Create New Account Contact").Execute
      AccountContactForm.Display (True)
   End If
End Sub
Private Sub oCreateAcctTaskBHandler_Click( _
ByVal Ctrl As Office.CommandBarButton, CancelDefault As Boolean)
   On Error Resume Next
   boolFoundAccountItem = 0
   Set oSelection = oExplorer.Selection
   For Each oItem In oSelection
      If oItem.MessageClass = "IPM.Post.Account info" Then
         boolFoundAccountItem = boolFoundAccountItem + 1
         'Set the item found to a global variable just in case
         'it is the only one found
         Set oAcctItem = oItem
      End If
   Next

   If boolFoundAccountItem = 0 Then
      MsgBox "You have no account selected. Please select " & _
         "an account and try again.", _
         vbOKOnly + vbExclamation, "Create Contact"
      Exit Sub
   ElseIf boolFoundAccountItem > 1 Then
      MsgBox "You have more than one account selected. " & _
         "Please select only one account and try again.", _
         vbOKOnly + vbExclamation, "Create Contact"
      Exit Sub
   ElseIf boolFoundAccountItem = 1 Then
```

(continued)

```
      Set AccountTaskForm = oAcctItem.Actions( _
         "Create New Account Task").Execute
      AccountTaskForm.Display (True)
   End If
End Sub

Private Function CheckSelection(strMessageClass) As Integer
   On Error Resume Next
   boolFoundAccountItem = 0
   Set oSelection = oExplorer.Selection
   For Each oItem In oSelection
      If oItem.MessageClass = strMessageClass Then
         boolFoundAccountItem = boolFoundAccountItem + 1
         'Set the item found to a global variable just in case
         'it is the only one found
         Set oAcctItem = oItem
      End If
   Next
   CheckSelection = boolFoundAccountItem
End Function
```

Notice that we use the standard Outlook object model to implement all three event handlers. The *oCreateAccountBHandler_Click* event handler is the simplest of the three since it only adds a new account form to the folder and displays this form to the user.

The other two event handlers, *oCreateAcctContactBHandler_Click* and *oCreateAcctTaskBHandler_Click*, also use some of the features in the Outlook 2000 object model. Before a user can create either an account contact or a task, the user must first select an account. The code for these handlers uses the new Selection collection on the Explorer object to determine what the user has selected in the user interface.

The *oCreateAcctContactBHandler_Click* and *oCreateAcctTaskBHandler_Click* event handlers both loop through the collection of selected items to see whether any are account items. Users can select multiple items in the user interface, so the code remembers how many account items it sees in the Selection collection. Both handlers set the last account item they see to a global variable just in case this account item is the only one in the selection. Because the add-in cannot guess for which account the user wants to create a new contact or task, the subroutines display error messages when the user has more than one account selected in the user interface. If no accounts are selected, the application displays an error message telling the user to select an account. If only one account is selected, the application calls the custom actions on the account form to create either a new account contact or a new task.

Adding Custom Property Pages and Storing User Settings

The final section of code implements the property pages that allow users to pick their custom settings for the application. We will also quickly look at how the registry is used to store these settings for the user. While we examine this code, you will see

some interesting objects implemented in Outlook 2000, such as the PropertyPage object, the PropertyPageSite object, and the PropertyPages collection object.

Custom property pages allow you to integrate your applications more tightly into the Outlook application. They also make it easier for your users to configure your application, because your customizations are part of the standard Outlook configuration pages. The following code implements the property page extension code in the COM add-in. Then we will look at the code for the ActiveX control, which creates the actual property page that appears.

```
Private Sub SetDefaultProps()
   oNewPage.prefLookForShortcuts = prefLookForShortcuts
   oNewPage.prefEnableAcctToolbar = prefEnableAcctToolbar
   oNewPage.prefMakeFolderHomepage = prefMakeFolderHomepage
   oNewPage.prefNotifyWhenNewMember = prefNotifyWhenNewMember
   oNewPage.prefNotifyWhenNewTask = prefNotifyWhenNewTask
End Sub

Private Sub oNS_OptionsPagesAdd(ByVal Pages As _
Outlook.PropertyPages, ByVal Folder As Outlook.MAPIFolder)
   If Folder.Name = "Account Tracking" Then
      'Add the Options page to the folder
      Set oNewPage = CreateObject("AccountPP.UCAdminPage")
      SetDefaultProps
      oNewPage.oAdminDLL = Me
      Pages.Add oNewPage
   End If
End Sub

Private Sub oApplication_OptionsPagesAdd(ByVal Pages As _
Outlook.PropertyPages)
   Set oNewPage = CreateObject("AccountPP.UCAdminPage")
   SetDefaultProps
   oNewPage.oAdminDLL = Me
   Pages.Add oNewPage
End Sub

Public Sub SetRegistryValues(prefShortcuts, prefAcctToolbar, _
prefFolderHomepage, prefNotifyMember, prefNotifyTask)
   'This subroutine is called by the Property page to have the
   'Options page persist its values
   boolSuccess = SetAppRegValue("CheckShortcuts", REG_DWORD, _
      prefShortcuts)
   boolSuccess = SetAppRegValue("AcctToolbar", REG_DWORD, _
      prefAcctToolbar)
   boolSuccess = SetAppRegValue("FolderHomepage", REG_DWORD, _
      prefFolderHomepage)
   boolSuccess = SetAppRegValue("NotifyTeamMember", REG_DWORD, _
```

(continued)

```
        prefNotifyMember)
    boolSuccess = SetAppRegValue("NotifyAcctTask", REG_DWORD, _
        prefNotifyTask)
End Sub
```

In this code, two subroutines handle the OptionsPagesAdd event. The first subroutine uses the NameSpace object. The OptionsPagesAdd event fires for the NameSpace object when the user clicks on a folder in the namespace you are monitoring and then selects Properties. The NameSpace *OptionsPagesAdd* event procedure is passed two parameters: *Pages*, which is a collection of Outlook PropertyPages; and *Folder*, which is the folder the user is trying to retrieve properties for.

The second subroutine uses the Outlook Application object. The OptionsPagesAdd event fires the Application object when a user selects Options from the Tools menu to configure the overall application settings for Outlook. Both *OptionsPagesAdd* event handlers call the same code because there is only one way to customize the Account Tracking application. However, for your add-ins, you could have two different property pages for these two different events, depending on your needs.

The first step both event handlers perform is creating an object. This object is an ActiveX control, which is the actual property page that the subroutine will add to the PropertyPages collection. We'll look at the code for the control later in this chapter.

The next step the event handler performs is to set the default properties for the new property page object we created. This is accomplished by setting some of the variables on the control to the values that are currently stored in the add-in. All of these values are originally retrieved from the registry.

After all properties are set for the controls on the form, the code passes a reference of the add-in to the new property page. You might be wondering why it does this. The main reason is to allow the ActiveX control property page call back into the add-in when a user makes a change and applies it. If the ActiveX control does not call back into the add-in, the add-in will not know that something has changed and, therefore, will not behave as expected.

The final step in the code is to add the new page to the PropertyPages collection. This is done using the *Add* method of the collection. You can call this method in two ways. The first way, which is exemplified in the code, passes an object to the method so that the object is displayed as a property page. The second way passes in the ProgID of the control as a string, which enables Outlook to create the control. If you use the second way, you can also pass an optional string that is the caption for the property page. We'll see how to set the caption when we pass an ActiveX control later on.

Now that we know how to add our pages to the PropertyPages collection, we need to look at what the actual page should implement. The following code implements the ActiveX control, which is the property page extension. Figure 10-13 on page 380 shows the interface for the control in Visual Basic 6.0 design mode.

```
Implements Outlook.PropertyPage
Private oSite As Outlook.PropertyPageSite
Dim m_prefLookForShortcuts As Integer
Dim m_prefMakeFolderHomepage As Integer
Dim m_prefNotifyWhenNewMember As Integer
Dim m_prefNotifyWhenNewTask As Integer
Dim m_prefEnableAcctToolbar As Integer
Dim m_fDirty As Boolean
Dim m_AdminDLL As Object
Private boolInitializing As Boolean

Private Sub SetDirty()
   If Not oSite Is Nothing Then
      m_fDirty = True
      oSite.OnStatusChange
   End If
End Sub

Public Sub RefreshControls()
   checkNotifyAccount.Value = m_prefNotifyWhenNewMember
   checkNotifyTask.Value = m_prefNotifyWhenNewTask
   checkPerformCheck.Value = m_prefLookForShortcuts
   CheckToolbar.Value = m_prefEnableAcctToolbar
   CheckWebShortcut.Value = m_prefMakeFolderHomepage
End Sub

Private Sub checkNotifyAccount_Click()
   If boolInitializing = False Then
      SetDirty
      m_prefNotifyWhenNewMember = checkNotifyAccount.Value
   End If
End Sub

Private Sub checkPerformCheck_Click()
   If boolInitializing = False Then
      SetDirty
      m_prefLookForShortcuts = checkPerformCheck.Value
   End If
End Sub

Private Sub checkNotifyTask_Click()
   If boolInitializing = False Then
      SetDirty
      m_prefNotifyWhenNewTask = checkNotifyTask.Value
   End If
End Sub

 Private Sub CheckToolbar_Click()
   If boolInitializing = False Then
      SetDirty
      m_prefEnableAcctToolbar = CheckToolbar.Value
   End If
End Sub
```

(continued)

```
Private Sub CheckWebShortcut_Click()
   If boolInitializing = False Then
      SetDirty
      m_prefMakeFolderHomepage = CheckWebShortcut.Value
   End If
End Sub

Private Sub PropertyPage_Apply()
   On Error GoTo PropertyPageApply_Err
   m_fDirty = False
   If Not m_AdminDLL Is Nothing Then
      m_AdminDLL.SetRegistryValues m_prefLookForShortcuts, _
         m_prefEnableAcctToolbar, m_prefMakeFolderHomepage, _
         m_prefNotifyWhenNewMember, m_prefNotifyWhenNewTask
      'Refresh the add-in DLL settings
      m_AdminDLL.CheckRegistryValues
   End If
   Exit Sub
PropertyPageApply_Err:
   MsgBox "Error in PropertyPage_Apply.  Err# " & Err.Number _
      & " and Err Description: " & Err.Description
End Sub

Private Property Get PropertyPage_Dirty() As Boolean
   PropertyPage_Dirty = m_fDirty
End Property

Private Sub PropertyPage_GetPageInfo(HelpFile As String, _
HelpContext As Long)
   HelpFile = "nothing.hlp"
   HelpContext = 102
End Sub

Private Sub UserControl_EnterFocus()
   boolInitializing = False
End Sub
Private Sub UserControl_Initialize()
   m_fDirty = False
   boolInitializing = True
End Sub

Private Sub UserControl_InitProperties()
   On Error Resume Next
   Set oSite = Parent
   RefreshControls
End Sub

Public Property Get prefLookForShortcuts() As Variant
   prefLookForShortcuts = m_prefLookForShortcuts
End Property
```

```
Public Property Let prefLookForShortcuts(ByVal vNewValue As Variant)
   m_prefLookForShortcuts = vNewValue
End Property

Public Property Get prefMakeFolderHomepage() As Variant
   prefMakeFolderHomepage = m_prefMakeFolderHomepage
End Property

Public Property Let prefMakeFolderHomepage( _
ByVal vNewValue As Variant)
   m_prefMakeFolderHomepage = vNewValue
End Property

Public Property Get prefNotifyWhenNewMember() As Variant
   prefNotifyWhenNewMember = m_prefNotifyWhenNewMember
End Property

Public Property Let prefNotifyWhenNewMember( _
ByVal vNewValue As Variant)
   m_prefNotifyWhenNewMember = vNewValue
End Property

Public Property Get prefNotifyWhenNewTask() As Variant
   prefNotifyWhenNewTask = m_prefNotifyWhenNewTask
End Property

Public Property Let prefNotifyWhenNewTask( _
   ByVal vNewValue As Variant)
   m_prefNotifyWhenNewTask = vNewValue
End Property

Public Property Get prefEnableAcctToolbar() As Variant
   prefEnableAcctToolbar = m_prefEnableAcctToolbar
End Property

Public Property Let prefEnableAcctToolbar( _
   ByVal vNewValue As Variant)
   m_prefEnableAcctToolbar = vNewValue
End Property

Public Property Get Caption() As Variant
   Caption = "Account Tracking"
End Property

Public Property Get oAdminDLL() As Variant

End Property

Public Property Let oAdminDLL(ByVal vNewValue As Variant)
   Set m_AdminDLL = vNewValue
End Property
```

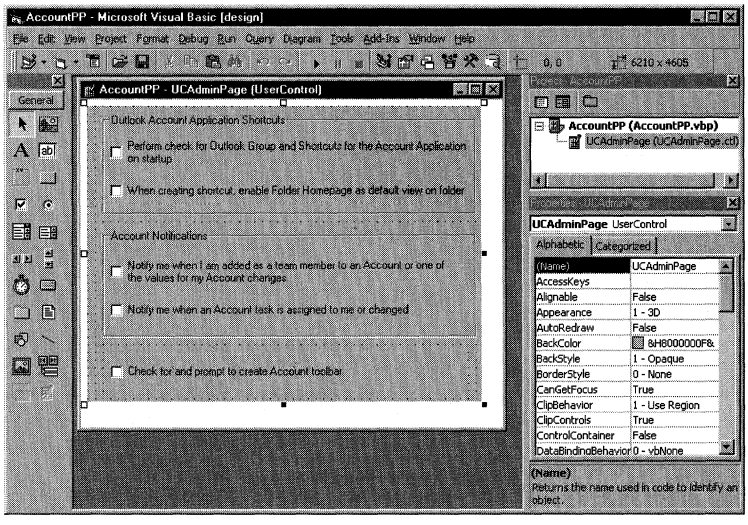


Figure 10-13. *The ActiveX control that implements the property page extension in Visual Basic 6.0 design mode.*

As you can see from the preceding code listing, not much code is implemented in the extension. Property page extensions are actually pretty easy to write. There are just a few important elements you need to implement.

The first of these elements, which is at the top of the code, implements the *PropertyPage* interface. To implement this interface, add a reference to the Outlook 2000 object model and then type *Implements Outlook.PropertyPage* in your declarations section. Once you do that, you will be able to select the different methods and properties you need to implement for your property page. You need to implement only two methods, *GetPageInfo* and *Apply*, and one property, *Dirty*.

The *GetPageInfo* method is called by Outlook to retrieve for your users help information about the property page. In this method, you can set two parameters: *HelpFile* and *HelpContext*. *HelpFile* is a string that points to your help file. (In the code, I point to a nonexistent help file.) The *HelpContext* parameter is a Long data type that specifies the context ID of the help topic associated with your property page.

The *Apply* method and the *Dirty* property work together with another method on the Outlook PropertyPageSite object, which we haven't discussed yet. The PropertyPageSite object, which in my code you can see declared in the declarations section, points to the container for your property page object. In this case, the container that holds your object is Outlook.

Once you declare a variable to be a PropertyPageSite, you need to initialize it. The best place to do this is in the intrinsic *InitProperties* event procedure of your ActiveX control. As you can see in the code, *oSite*, which is the *PropertyPageSite*

variable, is set to the intrinsic *Parent* property of the ActiveX control. The *Parent* property, in this case, returns an Outlook PropertyPageSite object.

Now that we know how to retrieve the PropertyPageSite, we can continue looking at the *Apply* method and the *Dirty* property. When users make changes to your page, the code must be able tell Outlook that the page has become dirty and the Apply button should become active. To do this, you can keep some private variable that tracks whether the user has changed an option—in essence, a dirty flag. When the user does change an option, you set this flag to True and then call the *OnStatusChange* method of the PropertyPageSite object.

The *OnStatusChange* method, in turn, forces Outlook to try to retrieve the *Dirty* property for your property page. You can see this implemented in the *Property Get PropertyPage_Dirty* procedure. The code sets the *PropertyPage_Dirty* variable to the value of the private dirty flag, which should be True, and returns that value to Outlook. Once Outlook receives a True value, it enables the Apply button.

So what happens when the user clicks this newly enabled Apply button? Well, your *PropertyPage_Apply* subroutine is called. In this subroutine, you should take whatever steps are necessary to apply the changes and also set your private dirty flag back to False. You can see in the code for my property page that I set the dirty flag to False, and then I attempt to save the values the user selected back to the registry.

You might be wondering what the m_AdminDLL object in this subroutine is. It is the reference to the add-in, which we passed to the property page when we created it. Since the registry functions are already implemented in the add-in, the property page just calls back to the add-in. The CheckRegistryValues call forces the add-in to refresh its internal values with the new values the user has selected.

The *Apply* method is also called when there is a dirty setting in your property page and the user clicks OK in the Properties or Options dialog box. The *Apply* method is not called when the user clicks Cancel.

The only other element that you must implement in your property page is its caption. To set the caption, you must add a property to your application. You can do this by selecting the Add Procedure option from Tools menu and then selecting the settings for creating a new public property, as shown in Figure 10-14.

Once you've added the *Caption* property procedure, select the Procedure Attributes option from the Tools menu, and select the property you just created from the drop-down list. Click the Advanced button, and select the Caption procedure ID in the Procedure ID dialog box, as shown in Figure 10-15. By doing this, you are associating your property with the identifier for the caption property on a control. Outlook will query this property ID for the caption for your property page. Then just implement the code to set this property to the value you want for your caption. An example of the *Caption* property procedure was shown in the previous code.

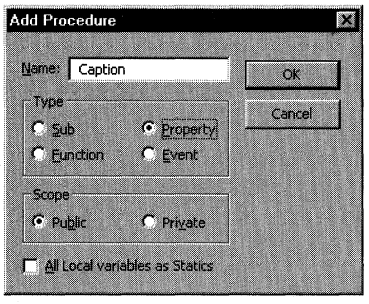


Figure 10-14. *Creating a new property for the caption of your property page.*

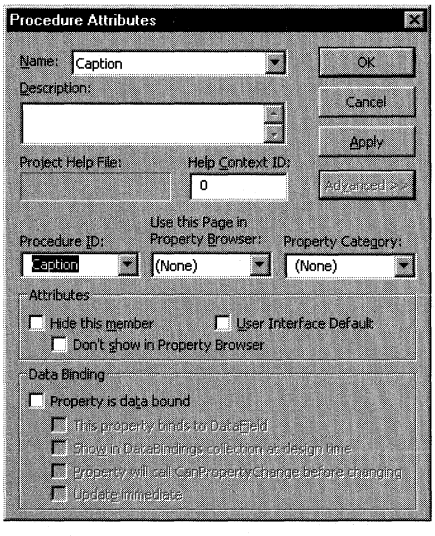


Figure 10-15. *Setting the Caption Procedure ID for your property.*

THE AVAILABILITY CHECKER SAMPLE APPLICATION

To make it easier for you to start building complex applications that leverage the development features in Outlook 2000, I've included some sample applications on the companion CD that are not covered in this chapter. One sample that you might

find particularly useful is the Availability Checker application. By combining COM add-ins, ActiveX controls, and custom Outlook forms, this sample allows you to synchronize the information on availability of users at different points in time (free/busy information) for all users in your organization. While working offline, the users can also create meeting requests that query the free/busy information for other users. Figure 10-16 shows the Property page you can use to set up the users you want to synchronize offline, and Figure 10-17 shows the custom Outlook form and ActiveX control that allow you to work with the synchronized free/busy information.

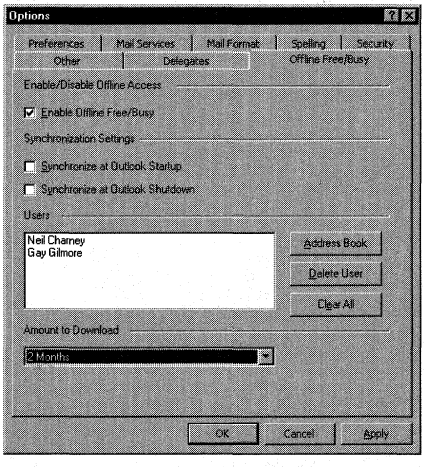


Figure 10-16. *The Property page of the Offline Free/Busy sample application.*

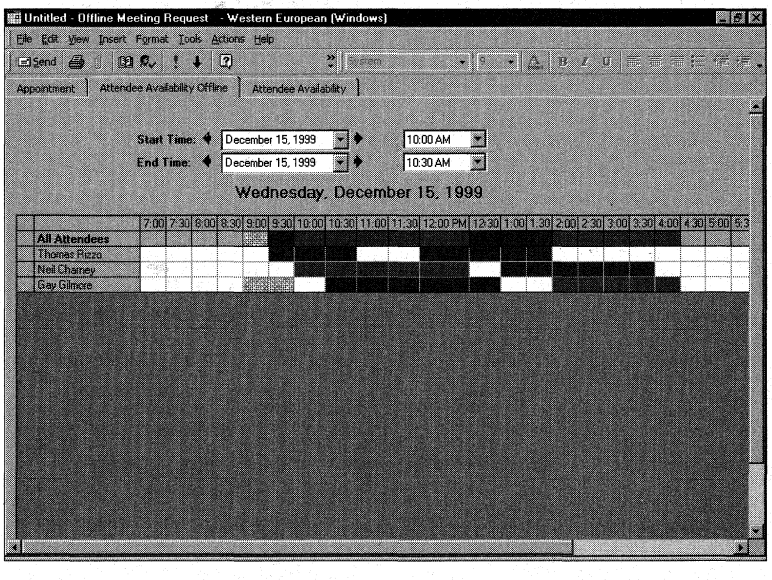


Figure 10-17. *The custom Outlook form and ActiveX control that allow you to browse the availability of users while you work offline.*

Chapter 11

Digital Dashboards

While the Digital Dashboard is an exciting concept, it's also very confusing. Your ideas about how to use a Digital Dashboard probably differ from mine, and my ideas will likely differ from another person's ideas, and so on. If you look at a Digital Dashboard at its most rudimentary level, you will see a set of dynamic Web pages that consolidate personal, team, corporate, and Internet information into a single unit. The way you implement those dynamic Web pages and configure your Digital Dashboard are up to you.

In this chapter, we'll look at the different technologies that you can use to create a Digital Dashboard as well as some Digital Dashboard samples. I'll suggest ways you can integrate Microsoft Exchange Server, Microsoft Outlook information, and business intelligence into your Digital Dashboard. After you've examined the ideas presented in this chapter, you'll be equipped to choose the best implementation for your Digital Dashboard.

PRELIMINARY CONSIDERATIONS

A Digital Dashboard is similar in concept to the dashboard of a car. Just as a driver relies on the dashboard to check the car's speed, oil level, and engine temperature, a computer user needs an easy way to quickly scan his computer for vital information. A Digital Dashboard combines an easy-to-use Web interface with a complex series of back-end systems such as Exchange Server, Microsoft SQL Server, or Web information systems.

When creating your Digital Dashboard, you should take into account the following considerations:

- *Customizability.* Imagine if the dashboard in your car displayed information about the voltage of the rear turn signal—you wouldn't consider that data terribly useful or important. In the same way that you want the dashboard of your car to display information that matters, you want your Digital Dashboard to display information that's pertinent to the current user. Allowing users to fully or partially customize the information contained in the Digital Dashboards makes their lives easier. If you're using Microsoft Windows 2000, Active Directory is the logical choice for holding your Digital Dashboard customization information.
- *Offline capability.* Usually the people who are the most interested in dashboards are executives. And most executives are mobile, meaning the ability to work offline is important to them. As you'll see later in the chapter, offline capability is one of the reasons the Digital Dashboard samples from Microsoft are hosted in Outlook 2000.
- *Scalability.* If you plan for 10,000 people to use your Digital Dashboard, you want to be sure that if 5,000 of them hit the same data source at the same time, your systems can handle the load. It's disheartening to try accessing content with a Digital Dashboard that's not up to the task.

Why Host a Digital Dashboard in Outlook?

You might be wondering why you should host your dashboard in Outlook rather than in a Web browser. There are a number of advantages and a couple of disadvantages to using Outlook to host a Digital Dashboard. However, I think the pros far outweigh the cons.

Outlook is usually the only application that users leave open the entire time they're working on their computer—which is the first reason you should host your Digital Dashboard there. What better way to encourage use of your dashboard than to integrate it into the productivity application that users work with a majority of the time? Besides, your users probably won't want to host a separate application to run a Digital Dashboard. To improve your dashboard's accessibility, you can provide links directly to it from the Outlook bar.

Second, the Outlook View control is locked down programmatically when it runs inside a Web browser. You saw this in the discussion of the Team Folder Wizard in Chapter 9. If you want to harness the full power of the View control—and for that

matter, the Outlook object model—you need to host your dashboard in Outlook. Also, the Outlook Databinding control that you'll see later in this chapter does not work outside the Outlook environment.

The ease of using COM add-ins is the third reason to host your Digital Dashboard in Outlook. For example, you could build an add-in that provides a wizard to customize your dashboard or that synchronizes content from databases or the Internet. You could build similar technologies in a Web browser–only environment, but you'd have to program them as ActiveX controls or server-side implementations. COM add-ins give you the benefits of component technologies and offload the processing to the client computer; in other words, your component can work in both a connected state and a disconnected state.

Finally, hosting your Digital Dashboard in Outlook allows for the offline synchronization of Exchange Server information and Web information. You can synchronize Folder home pages offline so that users can work with them when disconnected from the network. The only catch is that Outlook Today Web pages do not synchronize offline. You'll see how to get around this when we discuss deployment of the Digital Dashboard later in this chapter.

As I mentioned, there are a couple of compelling reasons *not* to host your Digital Dashboard in Outlook. Aside from not having the application, you might be inclined to host your dashboard elsewhere if you have a need for a roaming dashboard. For example, users who take a lot of business trips might frequently access information from more than one computer. If you can't guarantee that the computer being used by a dashboard user will have Outlook, you might want to host your dashboard directly in a browser.

EXAMPLE: THE FINANCE DIGITAL DASHBOARD

The Microsoft Digital Dashboard Starter Kit on the companion CD contains six Digital Dashboard samples. All the Digital Dashboards on the companion CD are built using Microsoft FrontPage, so you can open and modify them as needed. This starter kit also includes white papers on more advanced topics and some examples of styles and components you can use in your Digital Dashboard.

This starter kit isn't meant to be the standard you must follow for your own Digital Dashboards; the examples offer one interpretation of how you might build a dashboard. In fact, you could skip all the HTML and use Microsoft Visual Basic to build your dashboard as a client-side Microsoft ActiveX control hosted in a Web page. The important thing is that you understand how to get the data from your data sources and present that data coherently to your users.

In this section, we'll look at the Finance Digital Dashboard provided in the starter kit. Figure 11-1 shows the home page of the Finance dashboard.

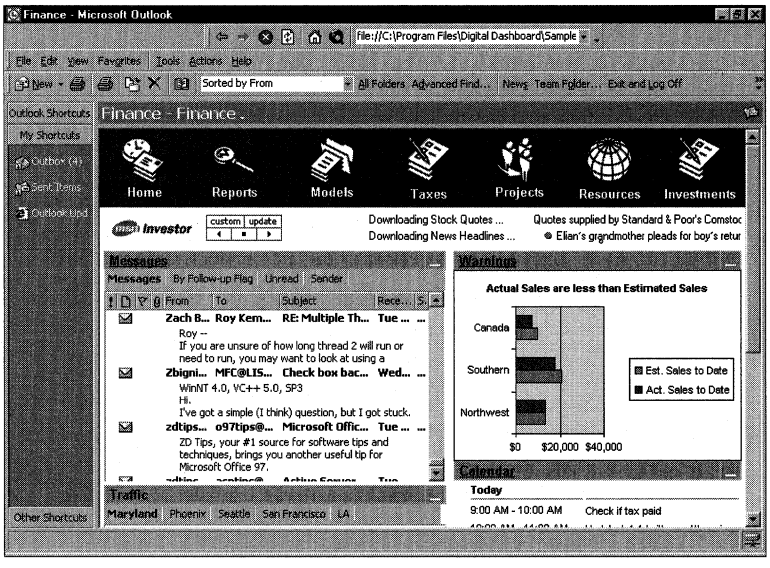


Figure 11-1. *The Finance dashboard home page.*

The home page of a Digital Dashboard includes a simple navigation bar at the top of the page, with the contents of the page below it. The home page of this dashboard includes the Microsoft Investor stock ticker, a chart created with the Microsoft Office 2000 Web Components (OWC) that shows sales figures, the Outlook View control showing the Inbox, traffic information from the Internet, and the Outlook Databinding control showing the calendar. You'll see how to utilize all these components when we discuss the Digital Dashboard architecture later in the chapter.

Figure 11-2 shows the Reports page of the dashboard, where the Office Web Components enable you to provide business information, specifically corporate sales, profits by warehouse, and costs by warehouse.

As you can see, it is possible to easily integrate business systems into the dashboard. The users can access the information they need without having to know the names of servers running SQL Server, connection strings, and other specific technical information.

THE OFFICE WEB COMPONENTS

The Office Web Components make up a set of ActiveX controls that provide scaled-down Office functionality. The set of controls consists of a data source, chart, pivot table, and spreadsheet component. You can embed these components in Web pages or Windows applications. The components are distributable, but to distribute them legally, you must procure an Office 2000 license on the client's machine.

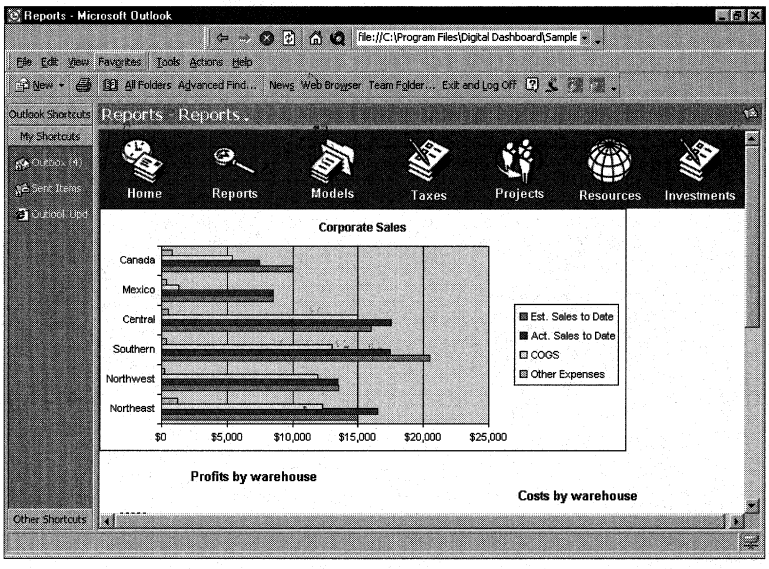


Figure 11-2. *The Reports page of the Finance dashboard.*

The Models page, shown in Figure 11-3, shows how you can use the PivotTable control of the Office Web Components. This page allows savvy users to cut and slice the data from the business systems and drill down to specific information. The PivotTable component could be using the data stored in a local replica of an OLAP cube or directly to a SQL database. It all depends on how you build your dashboard.

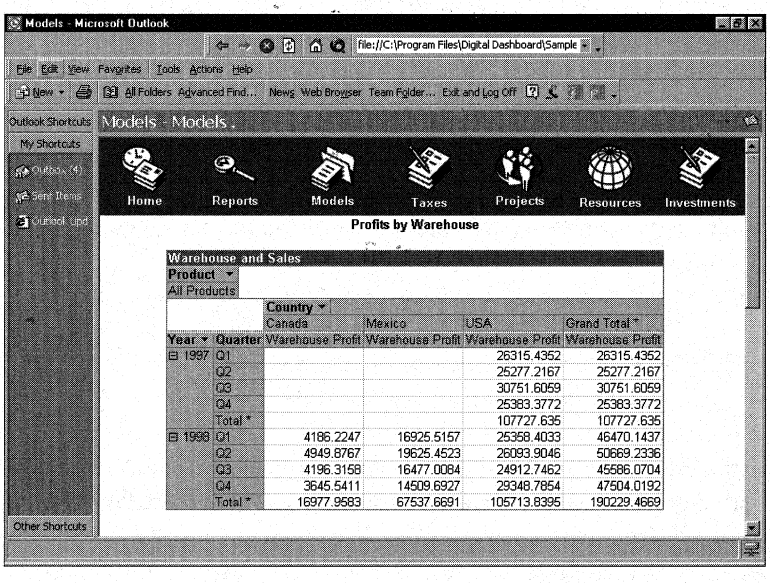


Figure 11-3. *The Models page of the Finance dashboard.*

Figure 11-4, the Taxes page, shows the Outlook View control pointing to an Exchange Server folder. Here users can click on the hyperlinks at the top of the View control to see different views of the information. By using the View control, users can drag and drop content into the dashboard pane or quickly change how they view information in the dashboard.

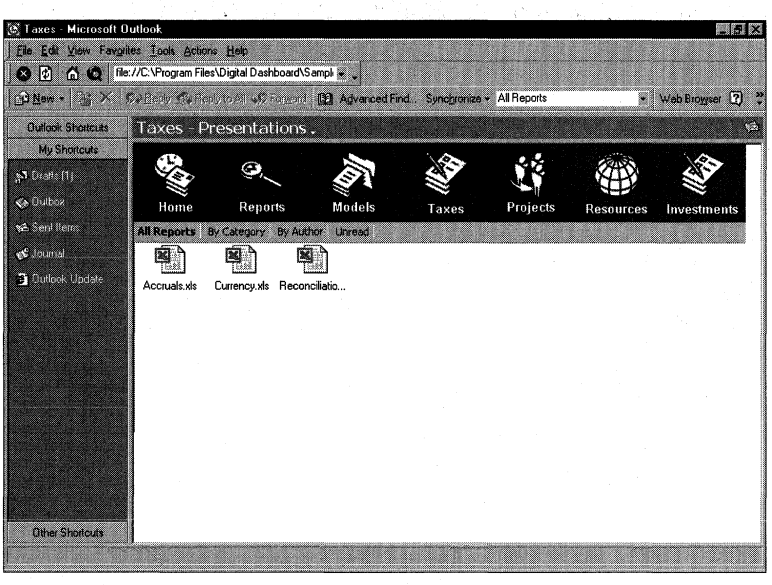


Figure 11-4. *The Taxes page of the Finance dashboard.*

The Projects page shown in Figure 11-5 integrates with the Team Folders Wizard. The projects on this page actually correspond to team project templates created by the wizard. You can build powerful business applications by combining the power of the Team Folders Wizard and the Digital Dashboard.

The Resources page in Figure 11-6 shows an easy way to provide the users of the dashboard with common hyperlinks to widely used Web sites. Finance buffs can quickly navigate from this page to MSN, MSNBC, and even Microsoft Expedia. The Resources page is an excellent example of integrating Web content into the Digital Dashboard.

Figure 11-7 on page 392, which shows the Investments page from the Finance dashboard, illustrates how you can integrate the functionality of Office 2000 into a Web page. The Resources page uses Microsoft Excel to display investment information and features current prices pulled from the Internet.

Figure 11-8 on page 392 shows how to edit the Finance dashboard in FrontPage. As mentioned earlier, all the Digital Dashboard examples on the companion CD were built with FrontPage. However, this doesn't mean you can't build your dashboard with another program, such as Microsoft Visual InterDev. Notice that none of the pages we just discussed require Microsoft Active Server Pages (ASP). This is because the Digital Dashboard Starter Kit needs to run on client machines that don't require much server infrastructure so that users can use the dashboards without setting up a server.

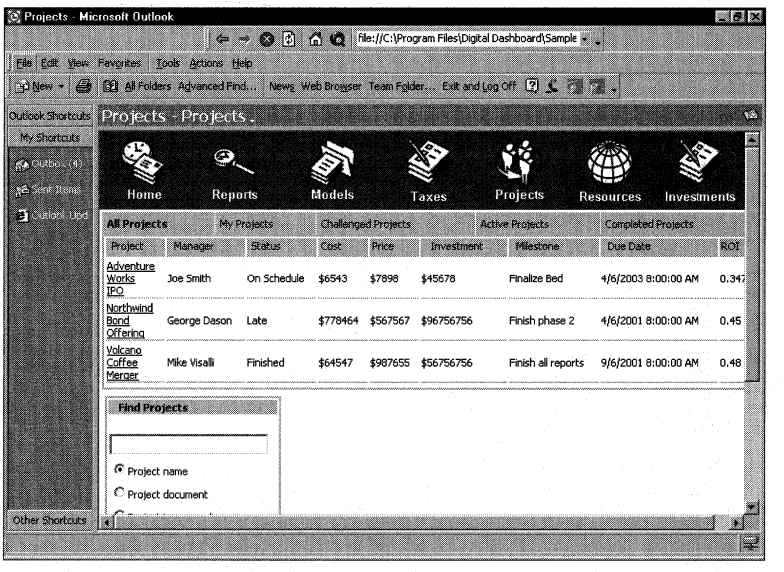


Figure 11-5. *The Projects page of the Finance dashboard.*

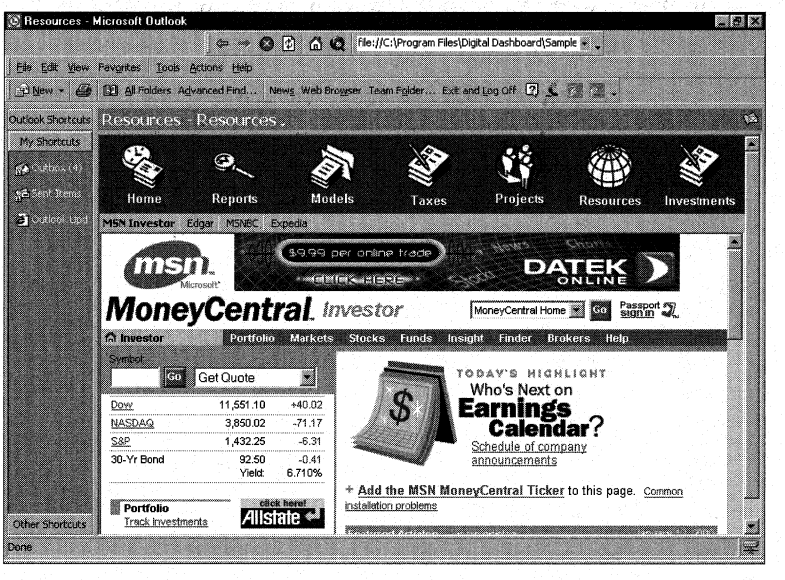


Figure 11-6. *The Resources page of the Finance dashboard.*

The other five Digital Dashboard examples in the starter kit offer variations of the features available in the Finance example. Rather than running through five more examples here, you can look at them on the companion CD at your leisure. Now let's take a look at how to actually build a Digital Dashboard.

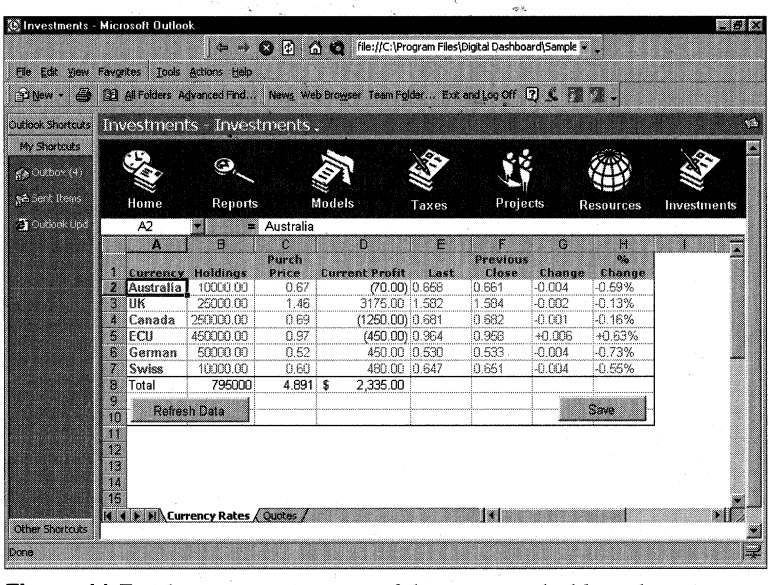


Figure 11-7. *The Investments page of the Finance dashboard.*

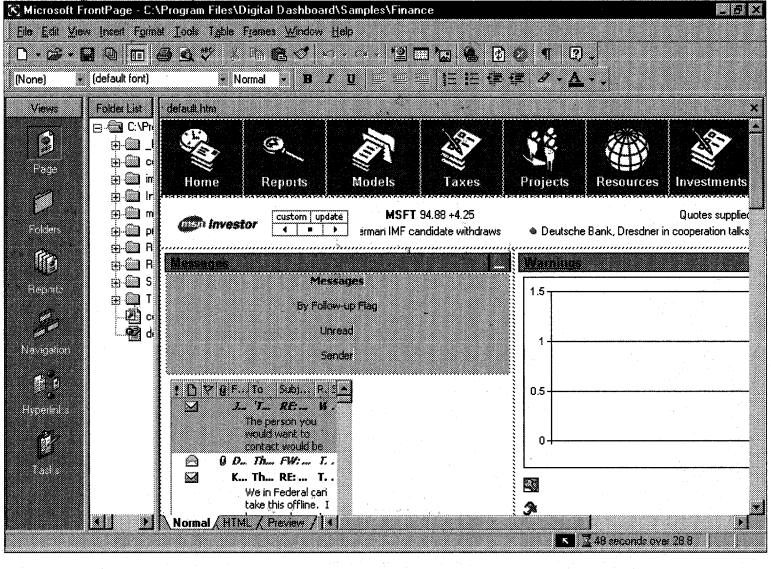


Figure 11-8. *Editing the Finance dashboard in FrontPage.*

BUILDING A DIGITAL DASHBOARD

In addition to the Digital Dashboard samples, the Digital Dashboard Starter Kit contains two COM add-ins that extend Outlook and Microsoft FrontPage and that you'll undoubtedly find useful. The first of these add-ins allows you to easily change the

Outlook Today URL without having to hack the registry. When you load this add-in, you'll find a new option in the Tools menu called Set Outlook Today Page. When you click on this menu option, you'll see the dialog box shown in Figure 11-9, in which you can set the URL you want to use for your custom Outlook Today page. However, this add-in is no way to deploy your Digital Dashboard to hundreds of users if you plan to change the default Outlook Today page for everyone. Rather, you need to use System Management Server (SMS), a setup program, or a logon script to modify the registry setting for Outlook Today programmatically.

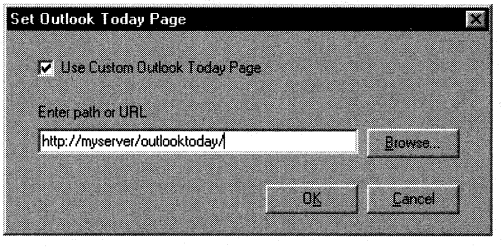


Figure 11-9. *The Set Outlook Today Page dialog box.*

The second COM add-in extends FrontPage, making it easier to insert Outlook information into your Web pages using either the Outlook View control or the Outlook Databinding control. Go to the Insert menu and select Outlook Controls, as shown in Figure 11-10. From the drop-down menu, you can select Outlook elements for insertion.

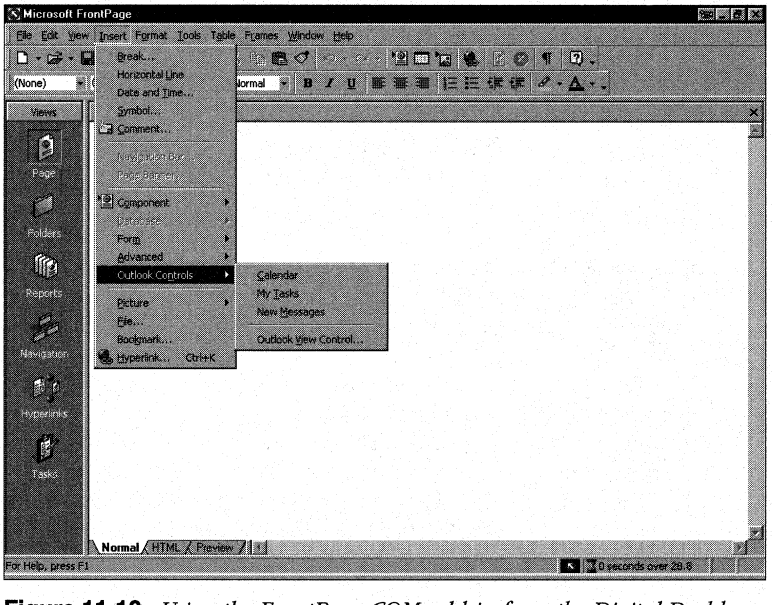


Figure 11-10. *Using the FrontPage COM add-in from the Digital Dashboard Starter Kit.*

Digital Dashboard Architecture

The Digital Dashboard examples included in the starter kit on the companion CD are predominantly composed of *information nuggets* that are exposed using DHTML *nugget windows,* or uniform graphical interfaces in the form of small windows. You can think of nuggets as self-contained applications or information. Client-side scripts either expose or hide these nuggets, depending on whether the user maximizes or minimizes the nugget window. Information nuggets usually consist of DHTML script and possibly an <OBJECT> tag for ActiveX control nuggets.

Every nugget has a unique name. It's a good idea to use some sort of convention to name your nuggets, such as nug_nuggetname_X. For example, if your nugget displays information from the user's Inbox, you might want to name it nug_inboxmessages_1.

Building Information Nuggets into the Dashboard

To better understand what a nugget is, let's look at some code. The following HTML code shows the Inbox nugget from the Finance Digital Dashboard. Notice the <DIV> tag that uniquely identifies the nugget as nug_messages_1. Also notice that the nugget uses the Outlook View control to display its information.

```
<div class="wholeNugget" ID="nug_messages_1" href="http://
www.microsoft.com">
   <table CELLPADDING="1" CELLSPACING="0" BORDER="0" WIDTH="100%">
      <tbody>
      <tr TITLE="Messages" STYLE="height:17px;
         font:bold 10pt arial" WIDTH="100%">
         <td NOWRAP ID="title" CLASS="NuggetBar"
            STYLE="padding-left:5px;border-left-style:solid;
            border-left-width:1px; border-top-style:solid;
            border-top-width:1px; border-bottom-style:solid;
            border-bottom-width:1px; text-decoration:underline">
            <span ID="text"><A href="outlook:inbox">Messages</A>
            </span></td>
         <td NOWRAP ID="drag" CLASS="NuggetBar"
            STYLE="border-top-style:solid; border-top-width:1px;
            border-bottom-style:solid; border-bottom-width:1px"> 
            </td>
         <td NOWRAP ID="disp" TITLE="Hide" CLASS="NuggetBar"
            onclick="displayNugget(nug_messages_1)"
            STYLE="border-top-style:solid; border-top-width:1px;
            border-bottom-style:solid; border-bottom-width:1px;
            border-right-style:solid; border-right-width:2px;
            border-left-style:solid; border-left-width:2px;width:16px">
            <img SRC="images/close.gif" width="17" height="13"></td>
         </tr>
      </tbody>
```

```
    </table>
    <div ID="content" CLASS="Nugget" STYLE="display:block;
            padding:0px; position:relative; top:-2; width: 100%;
            overflow-x:auto; margin:1px; border-top-width:0px">
       <table id="tblTopNavBar" border="0" cellpadding="0"
          cellspacing="0" style="width:100%;">
          <tr>
             <td valign="top" nowrap id="tdNavBar">
             <span class="topBarSpan"><span id="spanMessages4"
             onclick="HighPriorityMessages.view='Messages';
             changeMessagesTab(this);" xonblur="LoseFocus"
             xonFocus="changeFocus" id="FolderBtn2" class="btnFolder"
             style="font-weight:bold">Messages</span>
             <span id="spanMessages3"
                onclick="HighPriorityMessages.view='By Follow-up Flag';
                changeMessagesTab(this);" xonblur="LoseFocus"
                xonFocus="changeFocus" id="FolderBtnHome"
                class="btnFolder" style="width:0px; ">By
                Follow-up Flag</span> <span id="spanMessages5"
                onclick="HighPriorityMessages.view='Unread Messages';
                changeMessagesTab(this);" xonblur="LoseFocus"
                xonFocus="changeFocus" id="FolderBtn5"
                class="btnFolder">Unread</span>
             <span id="spanMessages2"
                onclick="HighPriorityMessages.view='By Sender';
                changeMessagesTab(this);" xonblur="LoseFocus"
                xonFocus="changeFocus" id="FolderBtn3"
                class="btnFolder">Sender</span>
             </span><!--End Folder Button Bar-->
          </td>
       </tr>
    </table>
       <div style="HEIGHT:200px;MARGIN-BOTTOM:0px;MARGIN-LEFT:0px">
          <object ID="HighPriorityMessages"
             CLASSID="CLSID:0006F063-0000-0000-C000-000000000046"
             style="width:100%;height:100%"
             codebase="../outlctlx.CAB#ver=9,0,3024" width="192"
             height="250">
             <param NAME="View" VALUE>
             <param NAME="Folder" VALUE="Inbox">
             <param NAME="Namespace" VALUE="MAPI">
             <param NAME="Restriction" VALUE>
             <param NAME="DeferUpdate" VALUE="0">
          </object>
       </div>
    </div>
</div>
```

You should take note of another <DIV> tag in this code: the <Content> tag. You can use this <DIV> tag in conjunction with some client-side JavaScript to hide or display the content of the nugget when the user clicks the Minimize or Maximize button. Here is the JavaScript code:

```
//-----------------------------------------------------------------
//  Function:       displayNugget
//
//  Description:    Alternately displays and hides the content
//     of the nugget
//
//  Arguments:      none
//
//  Returns:        nothing
//-----------------------------------------------------------------

function displayNugget(oNug)
{
   var e = oNug.all("disp");
   var f = oNug.all("content");

   ContentRegKey = "DD_CONTENT_" + document.title + "_" + oNug.id;

   // If contents are hidden, show them
   if (f.style.display == "none")
   {
      e.title = "Hide";
      e.children(0).src = "images/close.gif";
      f.style.display = "block";
      display = "0";
      // If the page is not run in the correct security context,
      try { window.external.SetPref(ContentRegKey,"display");}
      catch (exception) {}

   }
   // If contents are showing, hide them
   else
   {
      e.title = "Show";
      e.children(0).src = "images/open.gif";
      f.style.display = "none";
      display = "1";
      try { window.external.SetPref(ContentRegKey,"hide");}
      catch (exception) {}
   }
}
```

This code changes the image to reflect whether the content is visible or hidden. Also, the code updates the style for the HTML, indicating whether to display the nugget. Furthermore, the *SetPref* method, which is provided by Outlook so that you can easily write preferences to the registry, is called on window.external. *SetPref* takes two parameters: a string that specifies the key to which you want to write and a string that specifies the value for that key. The Digital Dashboard uses the registry to store the state of the nuggets as users open and close them. Figure 11-11 shows the registry location where these keys are stored. We'll discuss customizing the Digital Dashboard in more detail later in this chapter.

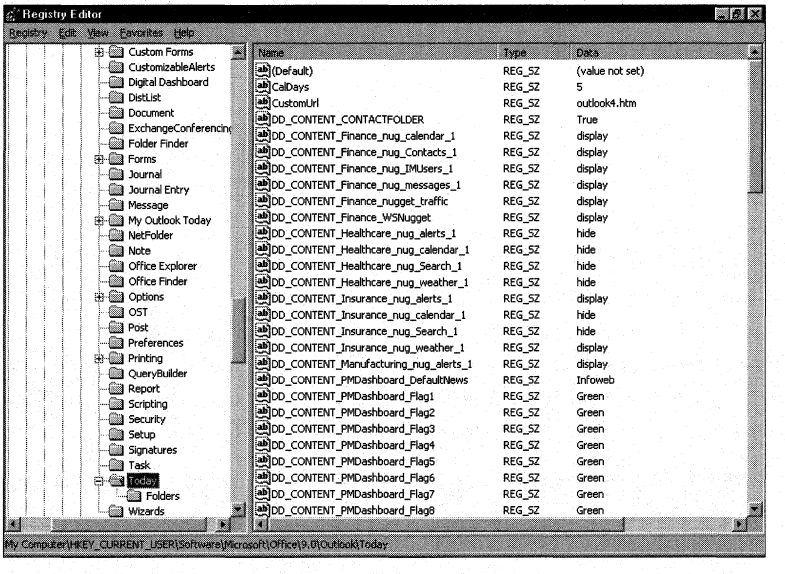


Figure 11-11. *Registry location for preferences stored using Outlook's* SetPref *method.*

As you can see, the dashboard is basically a wrapper for information nuggets. Your goal should be to create useful nuggets to plug into your Digital Dashboard. You can even create nuggets using <IFRAME> HTML tags rather than <DIV> tags.

> **WARNING** Each <IFRAME> tag you use will open a new instance of Microsoft Internet Explorer. If resources are limited on your client computers, use the <DIV> tag instead.

As I mentioned earlier, the dashboards described in this chapter are only examples to help you get started. You could scrap the entire idea of information nuggets and implement your Digital Dashboard using DCOM rather than HTML and JavaScript. With that said, let's take a look at how to build some new nuggets and add them to an existing dashboard.

Creating New Nuggets

Creating new nuggets is as simple as creating new HTML sections in your dashboard code. All you need to do to add your new nuggets of information is plug into the existing nugget framework that we looked at earlier. In this section, we'll look at adding two types of nuggets. The first is an ActiveX control provided in Windows 2000 that you can embed in your dashboard to view the information generated by the performance monitor in a Web application. The second nugget type is the MSN Messenger client for Microsoft Instant Messaging. The MSN Messenger client will illustrate how we can take advantage of real-time collaboration as part of our dashboard.

The System Monitor Nugget

We're going to add the system monitor nugget to the Finance dashboard; you've already seen the Finance dashboard, so the process will be easier for you to follow. The techniques illustrated in our example can be used to add the nugget to any of the dashboards. Figure 11-12 shows the System Monitor control integrated into the Finance dashboard. The nugget has its own top-level button on the dashboard banner so that users can quickly get to system information. The great thing about the System Monitor ActiveX control is that you can display your own system information or a remote server's system information. This makes it a helpful addition to a dashboard used by an administrator who has to monitor Windows 2000 servers, Exchange servers, or any other Microsoft Windows NT–based services that expose performance monitor counters. You will need Windows 2000 to use the System Monitor control since it ships as part of that operating system.

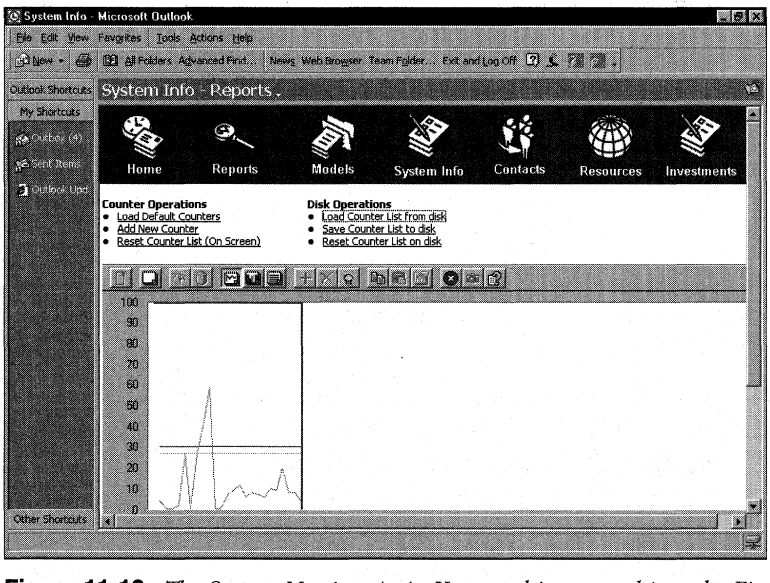


Figure 11-12. *The System Monitor ActiveX control integrated into the Finance dashboard.*

Creating the nugget for the System Monitor control is very straightforward. The nugget is a full-screen nugget, so we don't have to create a <DIV> tag because users will not be hiding or showing the nugget. (You'll learn how to enable hiding or showing of nuggets with the Instant Messaging nugget.) The System Monitor nugget does, however, use VBScript client-side code to load, save, and reset information in the registry by using the *GetPref* and *SetPref* methods. These methods allow you to set up the control the way you want to and then reload those settings every time you revisit the page that displays the nugget.

You set up of the System Monitor ActiveX control by using its object model, which is quite extensive. The System Monitor control supports methods and properties that allow you to lock down the control, add new counters, and retrieve the pathnames of the existing counters, as well as change the colors used throughout the control. Rather than review all features of the System Monitor control, I will point you to the Platform SDK to learn more about them.

Take a look at the following code for the nugget to quickly learn how to use the System Monitor ActiveX control in your dashboards:

```
<HTML>
<HEAD>
<TITLE>Reports</TITLE>
<link rel="stylesheet" type="text/css"
   href="../common/DigitalDashboard.css">

<meta name="Microsoft Border" content="t">
</HEAD>

<BODY><table border="0" cellpadding="0" cellspacing="0"
   width="100%"><tr><!--msnavigation--><td valign="top">

<table border="0" width="100%" height="77">
   <tr>
      <TD width=31% align=left height="77">
      <B>Counter Operations</B>
      <li><A href="" onclick="LoadDefaultCounters()">
         Load Default Counters</A>
      <LI><A href="" onclick="AddNewCounter()">Add New Counter</A>
      <li><A href="" onclick="Reset()">
         Reset Counter List (On Screen)</A>
      </TD>
      <TD width=69% align=left height="77">
      <B>Disk Operations</B>
      <li><A href="" onclick="LoadCounterListDisk()">
         Load Counter List from disk</A>
      <li><A href="" onclick="SaveCounterListDisk()">
         Save Counter List to disk</A>
```

(continued)

```
      <li><A href="" onclick="ResetCounterListDisk()">
         Reset Counter List on disk</A>
      </TD>
      </tr>
   </table>
   <table width=100%>
      <tr>
      <td width="50%" height="100%">
      <object
      classid="CLSID:C4D2D8E0-D1DD-11CE-940F-008029004347"
      name=SystemMonitor
      id=SystemMonitor1 v:shapes="_x0000_s1026"
      class=shape width=687 height=405>
      <param name="_Version" value="196611">
      <param name="_ExtentX" value="18177">
      <param name="_ExtentY" value="10716">
      <param name="DisplayType" value="1">
      <param name="ReportValueType" value="1">
      <param name="MaximumScale" value="100">
      <param name="MinimumScale" value="0">
      <param name="ShowLegend" value="-1">
      <param name="ShowToolbar" value="-1">
      <param name="ShowScaleLabels" value="-1">
      <param name="ShowHorizontalGrid" value="0">
      <param name="ShowVerticalGrid" value="0">
      <param name="ShowValueBar" value="-1">
      <param name="ManualUpdate" value="0">
      <param name="Highlight" value="0">
      <param name="ReadOnly" value="1">
      <param name="MonitorDuplicateInstances" value="-1">
      <param name="UpdateInterval" value="1">
      <param name="BackColorCtl" value="-2147483633">
      <param name="ForeColor" value="-1">
      <param name="BackColor" value="-1">
      <param name="GridColor" value="8421504">
      <param name="TimeBarColor" value="255">
      <param name="Appearance" value="-1">
      <param name="BorderStyle" value="0">
      <param name="GraphTitle" value>
      <param name="YAxisLabel" value>
      <param name="LogFileName" value>
      <param name="AmbientFont" value="-1">
      <param name="LegendColumnWidths"
      value="6.63650075414781E-02   7.08898944193062E-02
      0.085972850678733   9.20060331825038E-02   7.69230769230769E-02
      7.69230769230769E-02   9.80392156862745E-02">
      <param name="LegendSortDirection" value="0">
      <param name="LegendSortColumn" value="0">
```

```
      <param name="CounterCount" value="0">
      <param name="MaximumSamples" value="100">
      <param name="SampleCount" value="0">
      </object>
   </td>
   </tr>
   </table>
   <p> <!--msnavigation--></td></tr>
   <!--msnavigation--></table></BODY>


<Script Language=VBScript>


Sub LoadCounterListDisk()
   on error resume next
   'Load from the registry
   'Could also store this in the AD
   'or in a hidden message
   CounterCount = 0
   CounterCount = _
      window.external.getpref("DD_SYSMON_COUNTERCOUNT")
   if CounterCount = 0 then
      msgbox "There are no counters saved!"
   else
      'Clear the control
      SystemMonitor1.Reset
      for i = 1 to CounterCount
         'Retrieve the registry path for each counter,
         'and add it to the control
         CounterPath = window.external.getpref( _
            "DD_SYSMON_COUNTER_" & i & "_PATH")
         SystemMonitor1.Counters.Add CounterPath
      next
   end if
   window.event.Returnvalue = False
End Sub

Sub SaveCounterListDisk()
   on error resume next
   'Scroll through each counter in the control and save it out
   'First get the total and write it out
   iCount = SystemMonitor1.Counters.Count
   window.external.SetPref "DD_SYSMON_COUNTERCOUNT", CStr(iCount)
   'Scroll through each counter, and write it out
   if iCount = 0 then
      msgbox "No counters to save!"
```

(continued)

```
   else
      i = 1
      for each oCounter in SystemMonitor1.Counters
         window.external.setpref "DD_SYSMON_COUNTER_" & _
            i & "_PATH", oCounter.Path
         i=i+1
      next
   end if
   window.event.Returnvalue = False
End Sub

Sub ResetCounterListDisk()
   on error resume next
   'Reset the counter list
   'First get the total and see if we need to reset anything
   iCount = window.external.getpref("DD_SYSMON_COUNTERCOUNT")
   if iCount <> 0 then
      'Let's clear it out!
      for i = 1 to iCount
         window.external.setpref "DD_SYSMON_COUNTER_" & _
            i & "_PATH", ""
      next
   end if
   window.external.setpref "DD_SYSMON_COUNTERCOUNT", 0
   window.event.Returnvalue = False
End Sub

Sub Reset()
   SystemMonitor1.Reset
   window.event.Returnvalue = False
End Sub

Sub AddNewCounter()
   SystemMonitor1.BrowseCounters
   window.event.Returnvalue = False
End Sub

Sub LoadDefaultCounters()
   'You could pull the defaults from a network location
   'or hidden message if you want
   SystemMonitor1.Counters.Add "\Processor(_Total)\% Processor Time"
   SystemMonitor1.Counters.Add "\LogicalDisk(_Total)\Free Megabytes"
   SystemMonitor1.Counters.Add "\Objects\Processes"
   SystemMonitor1.Counters.Add "\Objects\Threads"
   window.event.Returnvalue = False
End Sub

</Script>
</HTML>
```

Instant Messaging Nugget

One interesting nugget that you can add to your dashboard provides instant messaging capabilities. The MSN Messenger client includes an ActiveX control as well as an object model that you can use so that you can automate the client to log on, check who is currently online, and send and receive messages from other online users. By integrating this functionality into your dashboard, your users can monitor when other team members come online and communicate with them. Figure 11-13 shows the Instant Messaging nugget integrated into the Finance dashboard.

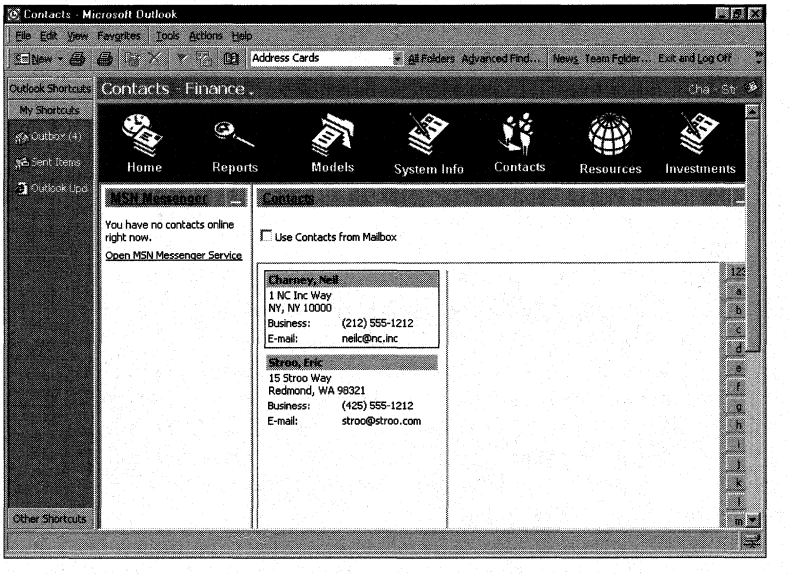


Figure 11-13. *The Instant Messaging nugget integrated into the Finance dashboard.*

Creating the instant messaging nugget involves creating an HTML <DIV> tag that will contain the contents of the nugget. The reason we use a <DIV> tag is so that we can hide and show the nugget according to the user's preferences. This nugget has some helper files written in both VBscript and JavaScript that manipulate the MSN Messenger object model. Note that Exchange 2000 will support Instant Messaging and Presence technology, so you can use this nugget for both the internal and external tracking of buddies. Instant Messaging is a great tool for corporations that want to use instant communications.

In addition to showing the Instant Messaging nugget, the page in Figure 11-13 shows the Contacts nugget. The Contacts nugget shows either the contacts in the current folder or the contacts from your default Contact folder by using the Outlook View control. These two nuggets are used together, so if you have Instant Messaging buddies who are in your Contact folders, you can quickly retrieve information about them as well as see if they are available online. The Contacts nugget is also implemented as a <DIV> tag, so the user can hide the nugget on the screen. The

following listing shows the code for both the Instant Messaging nuggets and the Contacts nugget. Note that I've left out the MSN Messenger code. You can find this code on the companion CD.

```
<TABLE BORDER=1 WIDTH=102% CELLPADDING=4 CELLSPACING=0
   BGCOLOR='#ffffff' height=650>

<!------------------------ BEGIN HORIZONTAL PANE ------------------>

   <TR>

<!------------------------ BEGIN LEFT PANE ------------------------>

   <TD width=15% id=LeftColumn vAlign=top height=87%>
   <font face='verdana,arial,helvetica' size=1>
   <div class="wholeNugget" ID="nug_IMUsers_1"
   href="http://www.microsoft.com">
      <table CELLPADDING="1" CELLSPACING="0" BORDER="0" WIDTH="100%">
      <tbody>
         <tr TITLE="Messages" STYLE="height:17px;
            font:bold 10pt arial" WIDTH="100%">
         <td NOWRAP ID="title" CLASS="NuggetBar"
            STYLE="padding-left:5px;border-left-style:solid;
            border-left-width:1px; border-top-style:solid;
            border-top-width:1px; border-bottom-style:solid;
            border-bottom-width:1px; text-decoration:underline">
            <span ID="text"><A href="http://www.microsoft.com">MSN
         Messenger</A></span></td>
         <td NOWRAP ID="drag" CLASS="NuggetBar"
            STYLE="border-top-style:solid; border-top-width:1px;
            border-bottom-style:solid;
         border-bottom-width:1px"> </td>
         <td NOWRAP ID="disp" TITLE="Hide" CLASS="NuggetBar"
            onclick="displayNugget(nug_IMUsers_1)"
            STYLE="border-top-style:solid; border-top-width:1px;
            border-bottom-style:solid; border-bottom-width:1px;
            border-right-style:solid; border-right-width:2px;
            border-left-style:solid;
            border-left-width:2px;width:16px">
         <img SRC="../images/close.gif" width="17" height="13"></td>
         </tr>
      </tbody>
      </table>
   <div ID="content" CLASS="Nugget" STYLE="display:block;
      padding:0px; position:relative; top:-2; width: 100%;
      overflow-x:auto; margin:1px; border-top-width:0px">
   <div style="HEIGHT:200px;MARGIN-BOTTOM:0px;MARGIN-LEFT:0px">
      <br>
```

```
      <script language="javascript">
         var s
         s='';
         s+=prepare_messenger()
         document.write(s);
      </script>
   </div>
   </div>
   </div>

      <!--End Nugget-->



      </font>


   </TD>

<!------------------------ END LEFT PANE -------------------------->

<!------------------------ BEGIN RIGHT PANE ----------------------->

   <TD id=RightColumn VALIGN=top width=97% height=100%>


   <div class="wholeNugget" ID="nug_Contacts_1">
      <table CELLPADDING="1" CELLSPACING="0" BORDER="0" WIDTH="100%">
      <tbody>
      <tr TITLE="Contacts" STYLE="height:17px;
      font:bold 10pt arial" WIDTH="100%">
         <td NOWRAP ID="title" CLASS="NuggetBar"
         STYLE="padding-left:5px;border-left-style:solid;
         border-left-width:1px; border-top-style:solid;
         border-top-width:1px; border-bottom-style:solid;
         border-bottom-width:1px "><span ID="text"
         STYLE="overflow:hidden">
         <A ID=ContactLink href="" target="_blank">
         <span style="overflow: hidden">Contacts</span></A></span></td>
         <td NOWRAP ID="drag" CLASS="NuggetBar"
         STYLE="border-top-style:solid; border-top-width:1px;
         border-bottom-style:solid; border-bottom-width:1px"> </td>
         <td NOWRAP ID="disp" TITLE="Hide" CLASS="NuggetBar"
         onclick="displayNugget(nug_Contacts_1)"
         STYLE="border-top-style:solid; border-top-width:1px;
         border-bottom-style:solid; border-bottom-width:1px;
         border-right-style:solid; border-right-width:2px;
```

(continued)

```
            border-left-style:solid; border-left-width:2px;width:16px">
            <img SRC="../images/close.gif" width="17" height="13"></td>
            </tr>
         </tbody>
         </table>
      <div  id="content">
      <form method="POST" action="">
      <p><input type="checkbox" name="C1" value="ON"
         onclick="LoadContacts()">Use Contacts from Mailbox</p>
      </form>
      <p>
      <object classid="clsid:0006F063-0000-0000-C000-000000000046"
      id="OVCtl1" width=100% height=500>
         <param name="View" value>
         <param name="Folder" value="Contacts">
         <param name="Namespace" value="MAPI">
         <param name="Restriction" value>
         <param name="DeferUpdate" value="1">
      </object>
      </div>
   </div>
         <!--End Nugget-->



   </td>

   <!------------------------ END RIGHT PANE ------------------------>
      </TR>

   </table>
    <!--msnavigation--></td></tr><!--msnavigation-->
   </table></BODY>

   <script language="vbscript">
   'on error resume next
   Dim ContactContentValue
   ApplyNuggetsStates
   ApplyContactFolder
   SetCheckBoxValue

   Sub SetCheckBoxValue
      document.all.C1.checked = ContactContentValue
   End Sub

   Sub LoadContacts
      strChecked = document.all.C1.checked
      'Set the registry, and apply the changes
```

```
   Window.external.SetPref "DD_CONTENT_CONTACTFOLDER",strChecked
   ApplyContactFolder
End Sub


Sub ApplyContactFolder
   Dim strFolder
   'Check to see if they want the standard contact folder
   'or this one
   ContactContentValue = _
      Window.external.GetPref("DD_CONTENT_CONTACTFOLDER")
   if ContactContentValue = "" then
      'First time set it to False
      Window.external.SetPref "DD_CONTENT_CONTACTFOLDER",False
   end if
   if ContactContentValue = "True" then
      'Show the default one
      oVCTL1.Folder = "contacts"
      strFolder = "outlook:contacts"
   else
      oVCTL1.Folder = "\\Digital Dashboard (Enhanced)\Finance\Contacts"
      strFolder = "outlook://Digital Dashboard " & _
         "(Enhanced)/Finance/Contacts"
   end if
   oVCTl1.DeferUpdate = 0
   'Set the hyperlink path
   document.all.ContactLink.href = strFolder
End Sub

Sub ApplyNuggetsStates()
Dim MyElement
for each MyElement in document.all
if MyElement.tagname="DIV" then
   if MyElement.classname="wholeNugget" then
      ContentRegKey = "DD_CONTENT_" & document.title & _
         "_" & MyElement.id
      ContentValue=Window.external.GetPref(ContentRegKey)
      select case ContentValue
         case "display"
            MyElement.all("content").style.display="block"
            MyElement.all("disp").children(0).src = _
               "../images/close.gif"
            MyElement.all("disp").title = "Hide"
         case "hide"
            MyElement.all("content").style.display="none"
            MyElement.all("disp").children(0).src = _
               "../images/open.gif"
```

(continued)

```
            MyElement.all("disp").title = "Show"
      end select
   end if
end if
next
End sub
</script>
```

USING OTHER COMPONENTS IN INFORMATION NUGGETS

There is a wealth of other components you can use as the basis for your information nuggets. In this section, we'll examine using the Office 2000 Web Components, the Outlook View control, and the Outlook Databinding control to build information nuggets. We'll also look at using Microsoft ActiveX Data Objects (ADO) as components for accessing data in your nuggets.

Office Web Components

Office 2000 enables you to publish spreadsheets, charts, and databases to the Web via a set of COM controls called the Office Web Components. You can use these components in any container that supports the hosting of ActiveX controls. However, you'll typically use these controls inside Web applications as client components, or in ASP applications to generate charts and graphs for your Web applications. The Finance Digital Dashboard that you saw earlier in the chapter used the Office Web Components.

This set contains four controls, called components: Data Source, Spreadsheet, PivotTable, and Chart. These components have an object library that you can program with to create applications. We'll quickly look at each of these components in this section; for more information, see the Office 2000 documentation.

Data Source Component

The Data Source component allows you to connect to external data sources to retrieve information for the other components to display. Although this control is invisible, it can establish data bindings to external data sources such as SQL Server or OLAP cubes. Usually when you create interactive Web applications using Office 2000, this control is automatically added and set up for you.

Spreadsheet Component

This component is like a miniature version of an Excel spreadsheet. It has a similar user interface and can support many of the functions that Excel supports. The Spreadsheet control enables you to load data programmatically using the Data Source con-

trol or any URL that points to an Excel spreadsheet saved as HTML. Once you retrieve the data, you can sort or filter it according to your application needs.

The Spreadsheet control is useful in applications in which you want a grid-based data entry model and calculation functionality such as expense reports or budgets. This component even supports real-time updates that you could use to display real-time information such as stock prices that are being fed from the Internet.

PivotTable Component

The PivotTable component is a small version of the Excel PivotTable. This component can retrieve data from OLE DB data sources or even OLAP data sources. Users of your application can model their data by filtering, grouping, and slicing it in numerous ways. One of the nice features of the PivotTable component is that you can link it with the Chart component (discussed next) so that when a user slices the data, the Chart component automatically updates to reflect the filtering or grouping that the user has selected.

Chart Component

The Chart component is the most flexible of the Office Web Components. The Chart component provides charting functionality similar to that of Excel. It can retrieve its data either by being filled with literal data, or by connecting to an ADO recordset or to the Spreadsheet or PivotTable components. As implied earlier, once you bind the Chart control to a data source, it will change whenever that data source is updated.

In addition to using the Chart control as a client-side component, you can use it as a server-side component in ASP applications. Many Web applications strive to provide data charting capabilities. Before the Chart control was developed, you either had to buy a third-party tool to chart information or you had to try automating Excel on your Microsoft Internet Information Services server, which is not a good idea. The Chart control enables you to connect to a data source in your ASP, chart the information, and then export that chart as a GIF image that you can embed in your web page. The image won't be as interactive as if you'd run the Chart component client-side in your application, but this solution works for browsers that do not support running ActiveX controls.

> **WARNING** Check your Office 2000 license to see if you need to buy any special license in order to use the Chart component on your IIS servers.

Outlook View Control

I won't cover the View control extensively here because we looked at it in great detail in Chapter 10. The View control is a great source of information for the nuggets you create for your Digital Dashboard. For more information on this control, refer back to Chapter 10.

Outlook Databinding Control

In Outlook Today, Outlook uses an ActiveX data source control similar to the Data Source control of the Office Web Components—the Databinding control. The difference is that the Outlook Databinding control works only in Outlook and can connect only to Outlook data. Furthermore, the Databinding control supports only a limited subset of fields from an Outlook data source. However, you can use the Databinding control within your Digital Dashboard to display nuggets of Outlook information. Figure 11-14 shows a not-so-pretty but functional example of hosting the Outlook Databinding control in a folder home page.

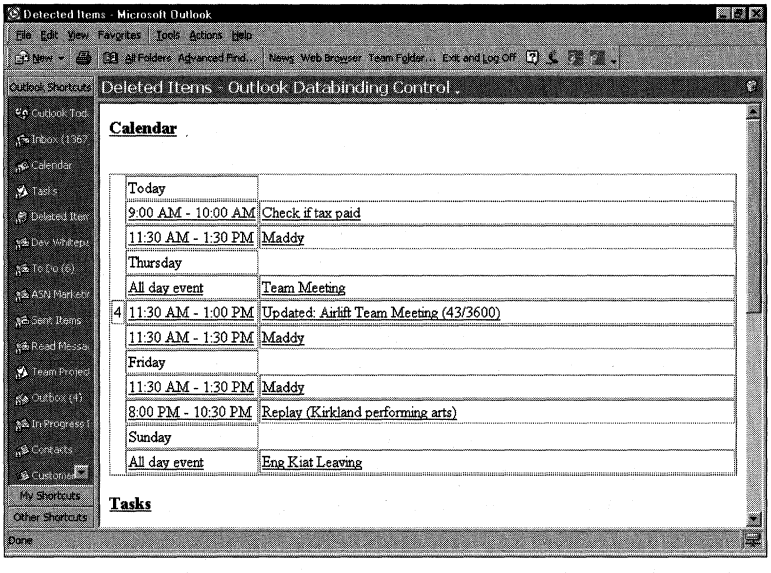


Figure 11-14. *The Outlook Databinding control in a folder home page.*

The number *4* appearing on the calendar is actually a placeholder for the arrow that normally points to the user's next appointment. You can use cascading style sheets (CSS) to format this data so that it appears as it does in the standard Outlook Today page. The code for this example page follows:

```
<html>
<HEAD>
   <title>Outlook Databinding Control</title>

   <OBJECT ID="CalList"


   CLASSID="CLSID:0468C085-CA5B-11D0-AF08-00609797F0E0">
   <PARAM NAME="Module" VALUE="Calendar"></OBJECT>
```

```
   <OBJECT ID="TaskList"


   CLASSID="CLSID:0468C085-CA5B-11D0-AF08-00609797F0E0">
   <PARAM NAME="Module" VALUE="Tasks"></OBJECT>

   <OBJECT ID="MailList"


   CLASSID="CLSID:0468C085-CA5B-11D0-AF08-00609797F0E0">
   <PARAM NAME="Module" VALUE="Inbox"></OBJECT>

</HEAD>
<BODY>

   <H3>Calendar</H3><BR>
   <table id=CalendarLiveTable border=1 cellspacing=1 cellpadding=2


      valign=top width=100% name="CalendarCol" datasrc="#CalList"


      style="display:;">
      <tr>
      <td nowrap valign=top width=10px align=left><div datafld="Next"


      DATAFORMATAS="html"></DIV></TD>
      <td valign=top nowrap><DIV DATAFLD="StartEnd" DATAFORMATAS="html"


      class=CalendarStartEnd > </DIV></TD>
      <td valign=top width=100%><div datafld="SubjectLocation"


      DATAFORMATAS="html" class=CalendarSubjectLocation> </DIV></TD>
      </tr>
   </table>

   <P><H3>Tasks</H3><BR>
   <TABLE border=1 name="TaskCol" cellspacing=0 id=TasksLiveTable


   datasrc="#TaskList" width=100% style="display:;">
   <TBODY>
      <TR>
   <TD width=1px><INPUT TYPE=checkbox DATAFORMATAS="Text"
```

(continued)

```
    DATAFLD="Complete" height=20px></TD>
    <TD width=1px><DIV DATAFLD="Importance" DATAFORMATAS="html"


    class=TaskImportance></DIV></TD>
    <TD><DIV DATAFLD="Subject" DATAFORMATAS="html"


    class=TaskSubject></DIV></TD>
       </TR>
    </TBODY>
    </TABLE>

    <P><H3>Folder List</H3><BR>
    <table border=1 name="MailCol" id=InboxLiveTable datasrc="#MailList"


    cellspacing=0 style="display:;">
    <TBODY>
    <tr>
    <td align=left valign=top class=borderBottom><DIV  DATAFLD="Name"


    DATAFORMATAS="html" class=Folder></DIV></td>
    <td nowrap valign=top class=borderBottom align=right><DIV


    DATAFLD="Count" DATAFORMATAS="html" class=InboxCount></DIV></td>
    </tr>
    </TBODY>
    </table>

</body>
</html>
```

The code creates three Outlook Databinding objects of the same type that have different module parameters. One object points at the calendar, another at the tasks folder, and the third at the Inbox. The nice thing about using databinding objects is that if the underlying Outlook folder changes, you don't need to refresh the page—the databinding objects automatically refresh it for you.

These three databinding objects correspond to the three HTML tables in the code. Each of the HTML rows has a <DIV> tag with a DATAFLD element. This DATAFLD element tells the Databinding control which field to place in that row of the table. The data sources of the Outlook Databinding control are limited to the fields shown in this table:

Data Source	*Fields*
Calendar	Next, StartEnd, and SubjectLocation
Tasks	Complete, Importance, and Subject
Mail	Name, Count

NOTE You cannot change these fields nor their grouping or filtering by using code in your application. This is one of the limitations of the Outlook Databinding control.

You can customize the data source objects for the Calendar and Task data sources to point to specific folders. Doing so allows you to quickly display public calendar or task information in your nuggets. To customize the data source objects, add the *Path* parameter to the object tag for the control. The *Path* parameter takes as its value the path to the folder to which you want the Databinding control to bind. The following example points the Databinding control at a calendar folder:

```
<object ID="CalList" CLASSID="CLSID:0468C085-CA5B-11D0-AF08-00609797F0E0">
   <param NAME="Module" VALUE="Calendar">
   <param NAME="Path"
      VALUE="\\Public Folders\Favorites\Marketing Calendar">
</object>
```

ActiveX Data Objects

ADO is a high-level interface to OLE DB data sources. You can use ADO to connect to a multitude of data sources that have OLE DB providers. In this section, we'll briefly look at using ADO to connect to SQL databases. This section covers ADO 2.1 only. When we discuss Exchange 2000 Server in Chapter 18, we'll examine ADO 2.5 and its new features, one of which is the ability to understand hierarchical data sources such as those used in Exchange 2000 Server. In Chapter 18, we'll also discuss using ADO 2.5 and the Exchange 2000 Server OLE DB provider to connect to Exchange 2000 Server data sources.

The ADO object model consists of seven objects—Connection, Command, Recordset, Parameter, Field, Property, and Error—and four types of collections—Fields, Properties, Parameters, and Errors. We won't cover each of these objects and collections extensively in this chapter. Instead, I'll quickly walk you through the basics of connecting to OLE DB data sources from your Digital Dashboard. For more information on ActiveX Data Objects, refer to the ADO documentation.

Connection Object

Before you can manipulate data in a database, you need to connect to that database. The Connection object enables you to do this. You specify your connection criteria, such as the name of the database and the server it resides on, and then you attempt

to connect. The following code sample shows how to create a connection object, set the connection properties, and then open the connection to the database:

```
Dim oConnection As ADODB.Connection
Set oConnection = CreateObject("ADODB.Connection")
With oConnection
   .Provider = "SQLOLEDB"
   .ConnectionString = "User ID=sa;Password=;" _
      & "Data Source=Server1;Initial Catalog=MyDB"
End With
oConnection.Open
```

> **NOTE** The provider for SQL Server is SQLOLEDB. The provider for ADSI is ADSDSOObject. The provider for Exchange 2000 Server and ADO 2.5 is EXOLEDB.

Recordset Object

Once you have a connection, you can create a Recordset object that will contain the records you want to retrieve from the data source. If the data source supports different cursor and lock types, you can specify them—for example, in the *Open* method. An example of a cursor type is a static cursor type, which allows you to see only a snapshot of the data; any additions, changes, or deletions made to the data source after you open the Recordset object won't be visible. Two lock type examples are optimistic and pessimistic locking. You can also pass a SQL query to the *Open* method on the Recordset object to filter or sort the Recordset before it returns to the server or client. For more information on this, take a look at the ADO documentation or *Programming ADO* (Microsoft Press, 2000).

The following code sample creates an ADO Recordset object, opens the user's table from the MyDB database used in the previous code, and closes both the Recordset and the Connection. You should close both when you complete your set of functions on each object.

```
Dim oConnection As ADODB.Connection
Set oConnection = CreateObject("ADODB.Connection")
With oConnection
   .Provider = "SQLOLEDB"
   .ConnectionString = "User ID=sa;Password=;" _
      & "Data Source=Server1;Initial Catalog=MyDB"
End With
oConnection.Open

Dim oRS As ADODB.Recordset
Set oRS = CreateObject("ADODB.RecordSet")
oRS.Open "Users", oConnection, adOpenDynamic, adLockOptimistic
oRS.Close
oConnection.Close
```

Getting at Recordset object data

Once you open your Recordset, you'll probably want to retrieve data from it. ADO provides a Fields collection that represents the columns in a row from the data source. Using the Fields collection, you can quickly and easily display data from a database in your Digital Dashboard. The following code uses the Fields collection to retrieve information from the database:

```
Dim oConnection As ADODB.Connection
Set oConnection = CreateObject("ADODB.Connection")
With oConnection
   .Provider = "SQLOLEDB"
   .ConnectionString = "User ID=sa;Password=;" _
      & "Data Source=Server1;Initial Catalog=MyDB"
End With
oConnection.Open

Dim oRS As ADODB.Recordset
Set oRS = CreateObject("ADODB.RecordSet")
oRS.Open "Users", oConnection, adOpenDynamic, adLockOptimistic

MsgBox oRS.Fields("UserName").Value
MsgBox oRS.Fields("UserEmail").Value
oRS.Close
oConnection.Close
```

Moving through the Recordset object

In addition to displaying information from the Recordset, you might want to move from one row to another in the Recordset object. ADO provides five methods that allow you to do this: *Move*, *MoveFirst*, *MoveLast*, *MoveNext*, and *MovePrevious*. The *Move* method moves the number of records that you specify from the current record. The *MoveFirst* and *MoveLast* methods move the cursor to the first or the last row of the Recordset, respectively. *MoveNext* and *MovePrevious* methods move the cursor to the next or the previous row, respectively.

For each of these methods, you need to use the *BOF* and *EOF* properties. The *BOF* property is a Boolean that specifies whether the current cursor position precedes the first record in the Recordset. The *EOF* property is a Boolean that specifies whether the current cursor position follows the last record in the Recordset. The following example shows how you can add navigation buttons to your application by using some of these methods and properties:

```
Dim oRS As ADODB.Recordset
Dim oConnection As ADODB.Connection
Private Sub ConnectToDB()

   Set oConnection = CreateObject("ADODB.Connection")
```

(continued)

```
    With oConnection
        .Provider = "SQLOLEDB"
        .ConnectionString = "User ID=sa;Password=;" _
            & "Data Source=Server1;Initial Catalog=MyDB"
    End With
    oConnection.Open


    Set oRS = CreateObject("ADODB.RecordSet")
    oRS.Open "Users", oConnection, adOpenDynamic, adLockOptimistic
End Sub

Private Sub cmdNext_Click()
    oRS.MoveNext
    If oRS.EOF Then
        'Moved too far
        oRS.MoveLast
    End If
End Sub

Private Sub cmdPrevious_Click()
    oRS.MovePrevious
    If oRS.BOF Then
        'Moved too far
        oRS.MoveFirst
    End If
End Sub
```

Filtering the Recordset object

You can use the *Filter* property to filter the rows returned in the Recordset. You pass to the *Filter* property a criteria string that specifies the filter you want to impose on the Recordset. You'll probably use the *Filter* property with the *RecordCount* property, which specifies how many records the Recordset contains. The following example sets the *Filter* property to filter only users from the United States. It also displays a warning if the value of the *RecordCount* property is 0. You can disable the filter by setting the *Filter* property to the constant adFilterNone.

```
Dim oRS As ADODB.Recordset
Dim oConnection As ADODB.Connection
Private Sub ConnectToDB()

    Set oConnection = CreateObject("ADODB.Connection")
    With oConnection
        .Provider = "SQLOLEDB"
        .ConnectionString = "User ID=sa;Password=;" _
            & "Data Source=Server1;Initial Catalog=MyDB"
    End With
    oConnection.Open
```

```
   Set oRS = CreateObject("ADODB.RecordSet")
   oRS.Open "Users", oConnection, adOpenDynamic, adLockOptimistic
   DoFilter
End Sub

Private Sub DoFilter()
   oRS.Filter = "Country = 'United States'"
   If oRS.RecordCount = 0 Then
      MsgBox "No records meet this criteria!"
   End If
End Sub
```

Command Object

You can use the Command object to build and execute commands on your data source. These commands can return Recordsets, execute bulk operations, or modify the structure of the data source. In your Digital Dashboard, your Command object will usually be returning Recordsets. The Command object has six properties—*ActiveConnection*, *CommandText*, *CommandTimeOut*, *CommandType*, *Prepared*, and *State*—and three methods—*Cancel*, *CreateParameter*, and *Execute*.

You'll probably use only the *CommandText*, *CommandTimeOut*, *CommandType*, and *Prepared* properties. *CommandText* contains a string that is a SQL statement or the name of a stored procedure you want to run on the data source. *CommandTimeOut* indicates how long, in seconds, you want the code to wait for the command to execute before terminating the attempt and displaying an error.

The *CommandType* property is a string that specifies the type of command contained in the *CommandText* property. *CommandType* is provided to optimize performance so that ADO doesn't have figure out the type of command you want to perform. This property can have the following values: *adCmdText*, indicating that *CommandText* contains a SQL statement; *adCmdStoredProc*, indicating that *CommandText* contains the name of a stored procedure; or *adCmdUnknown*, if *CommandText* is of unknown form.

The *Prepared* property is a Boolean value that specifies whether the provider should compile the command for reuse. By setting this property to True for commands you plan to reuse, you'll boost performance.

The *Cancel* method cancels the command that's currently executing. The *CreateParameter* method creates a Parameter object that you can use with stored procedures. The *Execute* method executes the command, whether it's a stored procedure or a SQL statement. Let's take a closer look at the capabilities of this method.

Execute Method

You can use the *Execute* method directly on the Connection object to add, delete, or update data in your data source or in methods on the Recordset object. We'll discuss both techniques so that you can decide which one works best for your applications.

Updating records using the Connection object requires you to write some SQL code. The following code snippet shows how to update the Users database to change the type of user:

```
Set oConnection = CreateObject("ADODB.Connection")
With oConnection
   .Provider = "SQLOLEDB"
   .ConnectionString = "User ID=sa;Password=;" _
      & "Data Source=Server1;Initial Catalog=MyDB"
End With

oConnection.Open
strSQL = "UPDATE Users SET Type = " & _
   "'Extranet' WHERE Type = 'Business Partner'"
oConnection.Execute strSQL
```

You can achieve the same result with the Recordset object by scrolling through each record and updating the field manually, as shown here:

```
Set oRS = CreateObject("ADODB.RecordSet")
oRS.Open "Users", oConnection, adOpenDynamic, adLockOptimistic
oRS.MoveFirst
Do While Not (EOF)
   If oRS.Fields("Type") = "Extranet" Then
      oRS.Fields("Type") = "Business Partner"
   End If
Loop
oRS.Update
```

Inserting new records can again be achieved with a SQL statement on the Connection object. On the Recordset object, the object model supplies the *AddNew* method for inserting new records. The following code example shows how to use the *Execute* method on the Connection object to create a new record:

```
strSQL = "Insert Users(Name, Type, Country) " _
   & "Values ('Tom','Internal','United States')"
   oConnection.Execute strSQL
```

This code sample uses the Recordset object to achieve the same result:

```
Set oRS = CreateObject("ADODB.RecordSet")
   oRS.Open "Users", oConnection, adOpenDynamic, adLockOptimistic
   oRS.AddNew
   With oRS
      .Fields("Name") = "Tom"
      .Fields("Type") = "Internal"
      .Fields("Country") = "United States"
   End With
   oRS.Update
```

Deleting a record yet again can be achieved by using a SQL statement for the Connection object, while the Recordset object provides the *Delete* method, which deletes the current record in the Recordset. The following code shows how to delete a record by using the Connection object:

```
Set oConnection = CreateObject("ADODB.Connection")
With oConnection
   .Provider = "SQLOLEDB"
   .ConnectionString = "User ID=sa;Password=;" _
      & "Data Source=Server1;Initial Catalog=MyDB"
End With
oConnection.Open
strSQL = "DELETE FROM Users Where Name = 'Tom Rizzo'"
oConnection.Execute strSQL
```

Using the *Delete* method on the Recordset object will delete the current record. Make sure that you move the Recordset to a valid record after performing the deletion. The following code deletes the current record and then moves the Recordset to a valid record:

```
Set oRS = CreateObject("ADODB.RecordSet")
oRS.Open "Users", oConnection, adOpenDynamic, adLockOptimistic
oRS.Delete
oRS.MoveNext
If oRS.EOF Then
   oRS.MoveLast
End If
```

This has been a quick overview of using ADO to access and manipulate relational database records. We'll take a closer look at using ADO with Exchange 2000 Server in Chapter 18. For more information on using ADO with SQL Server databases, refer to the ADO documentation.

STORING CUSTOM INFORMATION FOR THE DIGITAL DASHBOARD

Even though the dashboards in the starter don't provide personalization features, you'll probably want to enable users to personalize their features to some degree. Personalization can range from providing simple ways for the user to filter out items in their Inboxes on the dashboard, to providing wizards that allow users to customize the display and location of their nuggets. Once you provide personalization capabilities, you'll need to store the personal information in a particular location. Since you'll probably build your dashboard as a Web page, you can store the personalization information for the dashboard in several ways. This section outlines some of your options.

Using the Registry

You can personalize the content of the dashboard by using the registry, which is easy because Outlook already allows you to get and set information in the registry by using *GetPref* and *SetPref* methods. The starter kit uses the registry to remember the state of the different nugget windows, that is, whether they were maximized or minimized.

There are a couple of problems with using the registry, however. First, if you lock down the registry so that users cannot modify the registry at all, using *GetPref* and *SetPref* will not work. (However, I've never found anyone who completely locks down the entire registry so that the user population can't access it.) The second problem is with roaming users. Unless you set up your client machines to support a roaming registry, your settings will be machine-specific. Such settings are not useful if you have people who check in and out laptops for travel or move to different machines throughout the day.

Using Site Server Personalization and Membership

Another option for enabling personalization on the dashboard is to use the personalization capabilities of Microsoft Site Server. Site Server Personalization and Membership allow you to set up rules for personalizing Web content, and they allow users to customize the way Web pages are displayed. Such personalization information is stored for the user in a central repository. In your dashboard, you can query Site Server to figure out what content should be displayed, where it should be displayed, and for whom it should be displayed. Site Server Membership supports ADSI, so you can query the preferences of the user using that interface.

> **NOTE** If you're planning to implement Active Directory, you might just want to use it as the default for storing your preferences.

Using the Active Directory

The Active Directory is Microsoft's strategic directory store, and you should use it to store personalized information for your dashboard if it is available. Based on the technology from the Exchange directory, the Active Directory provides a wealth of features to store, query, and secure information about users, groups, and the organization. For example, you can query for a user's postal code or department to provide customized content without the user ever having to enter any personal information. The Active Directory is also extensible. This means that in addition to the basic information it maintains, you can extend the directory with your own custom content or classes.

Using Cookies

Since your dashboard will probably be a Web page, you can use cookies to store user preferences. The inherent problems with this approach are the possibility that your users turn off cookies for security reasons and that users roam but their cookies don't.

Using XML

Another possibility is to use XML to store preferences. You can then parse the XML code to figure out what content to display on the dashboard as well as where to display it. You can also use XSL to format the XML stream. Depending on your needs, you could store the XML stream in Exchange Server, in the file system, or even in the Active Directory.

Using Exchange Server

Storing the preferences directly in Exchange Server might be appealing if you don't have the Active Directory installed on your system. This option allows you to use the object models that you're used to—for example, the Collaboration Data Objects model (CDO). CDO allows preferences to roam with the user if the preferences are stored in the user's mailbox or a public folder. With CDO, these preferences can also be used offline. By using hidden messages in a user's mailbox or by storing hidden or visible preferences in a public folder, you can allow users to customize the dashboard and its content. For an example of storing configuration or personalization information as hidden messages, take a look at the Offline Free/Busy application on the companion CD. It uses hidden messages in your calendar folder to synchronize the free/busy information for selected users when you work offline. It also allows you to query that offline free/busy information by using CDO. Following is some code from the Offline Free/Busy application. It retrieves configuration information from a hidden message.

```
Private Function GetHiddenMessage(MsgClass As String) As Object
   'Won't work online
   'Set oCalendar = oSession.GetDefaultFolder(0)
   On Error Resume Next
   Set oMailbox = oSession.InfoStores( _
      "Mailbox - " & oSession.CurrentUser.Name)
   Set oCalendar = oMailbox.RootFolder.Folders("Calendar")
   Set oHidden = oCalendar.HiddenMessages
   'Clear the Filter
   oHidden.Filter = Nothing
   Set oFilter = oHidden.Filter
   oFilter.Type = MsgClass
   If oHidden.Count <> 0 Then
      'Return the message
```

(continued)

```
            Set GetHiddenMessage = oHidden.GetFirst
        Else
            'Nothing. Try to do a hard search on it.
            'Reset the filter
            oHidden.Filter = Nothing
            Debug.Print oHidden.Count
            For Each oHiddenMsg In oHidden
                If oHiddenMsg.Type = MsgClass Then
                    Set GetHiddenMessage = oHiddenMsg
                    Exit Function
                End If
            Next
            Set GetHiddenMessage = Nothing
        End If
        Err.Clear
    End Function

    Private Sub GetAppSettings(boolFromPropPage As Boolean)
        On Error GoTo errHandler
        Err.Clear
        'Get the hidden message
        Set oFBMsg = GetHiddenMessage("IPM.Post.FBSetting")
        If oFBMsg Is Nothing Then
            'Create the message
            Set oMailbox = oSession.InfoStores("Mailbox - " & _
            oSession.CurrentUser.Name)
            Set oCalendar = oMailbox.RootFolder.Folders("Calendar")
            Set otmpHidden = oCalendar.HiddenMessages
            Set oFBMsg = otmpHidden.Add(Subject:="FB Setting", _
                Type:="IPM.Post.FBSetting")
            'Add a Boolean to hold the setting
            oFBMsg.Fields.Add "Enabled", 11, False
            boolEnabled = oFBMsg.Fields("Enabled").Value
            oFBMsg.Update True, True
        Else
            'Get the setting
            boolEnabled = oFBMsg.Fields("Enabled").Value
        End If
    . . .
    End Sub
```

DEPLOYING A DIGITAL DASHBOARD

Depending on the technology you use to develop a Digital Dashboard, the method for deploying the dashboard will differ—for example, locating your dashboard in Outlook Today will demand a deployment that is different from one in which you use Internet Explorer or an Outlook Folder Homepage. Let's take a look at these different scenarios and some best practices for deploying your dashboard.

Outlook Today Digital Dashboard

When you create your dashboard as a replacement for the Outlook Today page, you can force Outlook to start up in your dashboard by telling Outlook to always start in the Outlook Today folder. Outlook supports starting up only in its default folders or in Outlook Today, so unless the Folder Homepage dashboard is associated with a default Outlook folder such as Contacts or Tasks, you cannot force your dashboard to start up in the Folder Homepage.

When customizing Outlook Today, you can choose from a number of options for deploying your solution. Your first option is to distribute your applications as a local or remote DLL. By placing your Web resources in a DLL, you'll gain the benefit of having your page load faster. However, deploying your DLL will be more difficult and redeploying after making changes will be a problem, especially if you use a local DLL. Furthermore, you will need Microsoft Visual Studio to create the resource DLL file.

Your second deployment option is to host your Outlook Today page on a Web server. Doing so makes your dashboard easy to deploy and change, plus you can leverage Active Server Pages. The only issue with Outlook Today and remote Web servers is offline use. Outlook Today does not synchronize offline in the way that a Folder Homepage does. If you need to use your dashboard offline, you can still use a Web server and Outlook Today, but you will need to make the Folder Homepage point to your dashboard so that the Folder Homepage can synchronize the Web information offline. When working with offline Web pages, Outlook will look to the Internet Explorer cache, which doesn't distinguish the Outlook folder associated with the Web pages. Once you get the pages into the Internet Explorer cache, you can use these pages both from the Folder Homepage or your Outlook Today page.

Folder Homepage Digital Dashboard

By making your dashboard a Folder Homepage for a private or public folder, you get the benefit of automatic offline synchronization. However, this benefit comes at a price: you can't force Outlook to start in your dashboard as the default unless the dashboard is associated with a standard Outlook folder. That said, you can use the same deployment techniques for Outlook Today and for Folder Homepages such as a DLL or Web server.

Recommendations for Accessing Exchange Server

The Starter Kit uses the Outlook object model extensively, which you might want to avoid if performance is a concern for your dashboard. The following recommendations will help you build dashboards that perform very well with Exchange data sources.

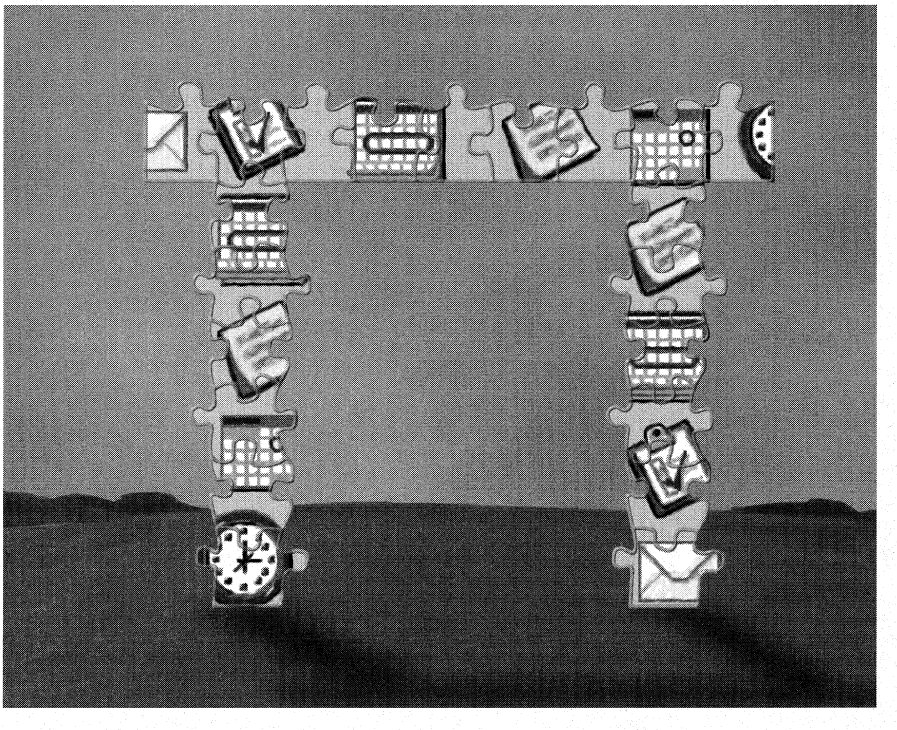
- *Use the databinding controls whenever possible.* You'll want to take advantage of the Outlook Databinding control when you need to display calendar or task information and you don't need a way for the user to drag and drop information contained in the HTML. The Databinding control streams information quickly into the page and eliminates some of the overhead of using the Outlook View control when displaying Outlook information. If your users require interaction with the data in a robust way, however, you'll want to use the View control in your application.
- *Use CDO whenever possible.* CDO is very good at accessing Exchange data, both online and offline, quickly, so try to use it wherever possible to access, update, or create information in your dashboard. If you need to interact with certain types of information in Outlook such as tasks or contacts, you'll want to use the Outlook object model or the View control. Both of these technologies expose a much richer interface to data types like tasks or contacts.
- *Be smart about client-side scripting.* Too much client-side script can slow your dashboard to a crawl. If possible, leverage ASP or some form of server component to display or retrieve data.
- *Use COM add-ins.* If you're going to host your dashboard in Outlook 2000, you won't want to forget about COM add-ins. By offloading some work to the add-in, you can take advantage of the speed with which the compiled code in the add-in can run. Plus, your add-ins can perform work during an entire Outlook session, which makes your dashboard performance even better. For example, in your dashboard, your add-in could synchronize with an OLAP cube in the background rather than in the foreground.
- *Don't be afraid of ActiveX controls.* If you're going to be hosting your dashboard in Outlook, this is especially true. Why spend countless hours tweaking HTML when you could write an ActiveX control that creates a better layout or has more functionality than you could provide in a Web page? Let your VB skills loose and write ActiveX controls when it makes sense for your dashboard. The benefit is that you can leverage these controls in other scenarios as well. The only drawback is deploying the controls and securing them. You can deploy your controls from a Web page, and if you use ActiveX control signing, you shouldn't have a problem with security.

WHAT ABOUT EXCHANGE 2000?

We're going to cover Exchange 2000 in much more detail in a later chapter. However, I wanted to point out that any of the nuggets you generate in Exchange 5.5 will work in Exchange 2000. This means the Outlook View control, the Databinding control, and the CDO and Outlook object model code you write will work in Exchange 2000. You can enhance your dashboard with Exchange 2000 technology because it offers richer security, Outlook Web Access, and new workflow capabilities.

Part III

Collaboration with Microsoft Exchange



Chapter 12

Collaboration Data Objects

As you saw in Chapter 7, you can use Microsoft Active Server Pages (ASP) to develop powerful Web applications that are not dependent on the capabilities of the browser. This power of ASP comes not from the built-in libraries of the ASP object model but rather from Active Server components you call from your ASP programs. This chapter covers one of those Active Server components—Collaboration Data Objects, or CDO—which enables you to develop messaging and collaboration applications for the Web. You can use CDO for other purposes as well, such as in client-based or server-based applications that you develop. In this chapter, you will learn what CDO is and how it compares to other technologies, what objects are in the CDO library, and how you can start developing ASP and Microsoft Visual Basic applications that take advantage of this library.

WHAT IS CDO?

CDO is an object library that exposes the interfaces of the Messaging Application Programming Interface (MAPI), but instead of requiring the C/C++ language as MAPI does, CDO can be programmed using any development tool that creates COM objects, such as ASP, Visual Basic, and Microsoft Visual C++.

CDO has had several incarnations, and previous versions shipped with different names and functionality. For example, in Microsoft Exchange Server version 4.0, CDO was named OLE Messaging, and in Exchange Server 5.0, CDO was named Active

Messaging. With the advent of Exchange Server version 5.5 and Microsoft Outlook 98, the library was renamed Collaboration Data Objects to better describe its services—CDO provides much more than messaging functionality. Even though the names have changed from version to version, any older applications using a previous version of CDO are compatible with the latest version of the library. (At the time this book goes to press, the latest released version of CDO is CDO for Microsoft Windows 2000.)

You install CDO by installing either Outlook on your machine or Outlook Web Access (OWA) on your Web server. CDO is actually divided into two dynamic link libraries (DLLs): CDO.dll and CDOHTML.dll. CDO.dll contains the core collaborative functionality of CDO, such as sending messages, accessing the directory, and viewing free/busy calendar information. CDOHTML.dll is the CDO Rendering library. This library allows you to automatically convert information stored inside Exchange Server to HTML by using custom views, colors, and formats. The CDO Rendering library is installed when you install OWA on your Web server. Throughout this chapter, you will learn how to use both CDO libraries in your applications.

CDO and the Outlook Object Library

CDO and the Outlook object library are complementary technologies. The Account Tracking application discussed in Chapter 6 illustrates how the CDO library is used in conjunction with the Outlook object library. You might be wondering when to use each library. To help you make a decision, take into consideration account criteria such as where the application will run and what type of information the application will access. As you develop applications, you will find that deciding whether to use the Outlook library or the CDO library will almost never be simple. Use the Outlook object library to do the following:

- Access special information stored in Outlook, such as Tasks and Journal items, that CDO does not support
- Open another user's information, such as the Calendar or Inbox
- Sort or filter complex Outlook properties

Use the CDO library to do the following:

- Render objects or data into HTML
- Create multiuser server-based applications
- Access detailed information stored in the directory or display address books for users to pick from

CDO and the CDO for Windows 2000

When you install Microsoft Internet Information Services (IIS) 5.0, you have the option to install an SMTP (Simple Mail Transfer Protocol) component and an NNTP (Network News Transfer Protocol) component on your Web server. These components are subsets, functionally, of the CDO library named Collaboration Data Objects for Windows 2000 (CDOSYS). The CDO for Windows 2000 library allows you to quickly build applications that do not require the complete functionality of CDO. For example, if on your Web page you wanted to create a simple way for users to send comments through e-mail, you would use the CDO for Windows 2000 object library rather than CDO. If your application required looking up a user in a directory server, however, you would want to use the advanced functionality of the full CDO library. Another difference between CDO and CDO for Windows 2000 is that CDO for Windows 2000 uses only SMTP and NNTP to communicate with a server. The use of these protocols to talk with the server limits the functionality that the CDO for Windows 2000 library can provide. With that said, you should use the CDO for Windows 2000 library to do the following:

- Send unauthenticated e-mail from a Web page
- Send bulk mailings via e-mail
- Support MHTML (Mime HTML)

Use the CDO library to do the following:

- Use authenticated or anonymous access to information, but not anonymous e-mails
- Access or create calendaring information
- Access a directory and its information

OVERVIEW OF THE CDO LIBRARY

The CDO library is a hierarchical library consisting of objects and collections. Throughout this hierarchy, you can create child objects from their parent objects. As you read the chapter, you'll find the same collection name for collections containing different objects. While these collections might have the same name, the information each accesses is specific to the object each refers to.

In the CDO library, the Session object is the highest-level parent of all the other objects and contains all objects and collections. This makes sense since you need some type of session, either an Exchange Server session or an offline session, to start accessing information stored in an Exchange Server database. Figure 12-1 shows the hierarchy of the CDO library, which begins with the Session object.

NOTE If you're familiar with ActiveX Data Objects (ADO), it might be helpful to know that the Session object is similar to the ADO Connection object. However, do not confuse the Session object in ASP with the Session object in CDO. They are two entirely different objects.

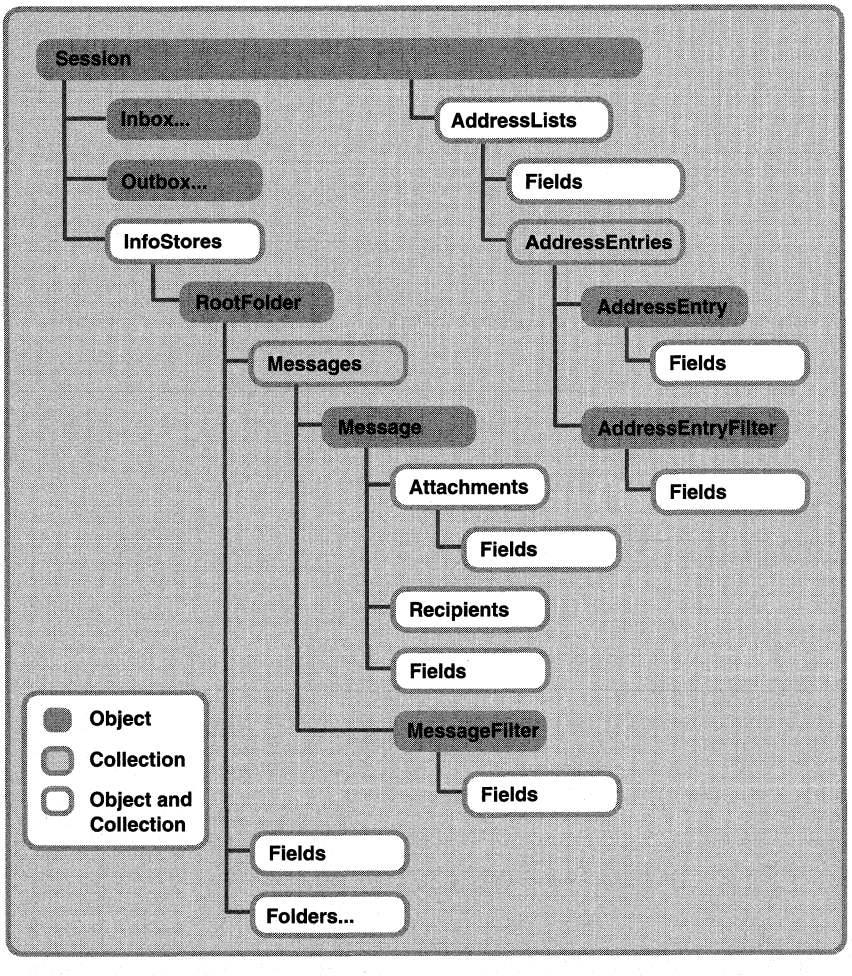


Figure 12-1. *The CDO library hierarchy. Notice how all other objects are created as children of the CDO Session object.*

As you can see from Figure 12-1, the CDO library is quite logical in its layout of collections and objects. Below the Session object are the major collections and objects of the CDO library, such as the InfoStores collection, which contains the data stores for your application, and the AddressLists collection, which contains the address entries your application can use. Below these major collections are other collections

such as the Folders collection, which contains the folders for a particular InfoStore, and the Messages collection, which contains the messages for a particular folder. The CDO library is one of the most approachable Microsoft object libraries and allows you to quickly build powerful collaborative applications.

Getting Help with the CDO Library

While this chapter provides an overview of the CDO library, you might also want to look at the CDO help file, which provides useful information as well as code samples. The CDO Windows Help file (cdo.hlp) can be found on the Exchange Server CD or on the Outlook CD. The CDO compiled HTML Help file (cdo.chm) is available on the companion CD. It is also available as part of the Platform Software Development Kit (SDK) section of the MSDN Library, which can be accessed online at *http://msdn.microsoft.com/developer/sdk/*. For additional CDO information, refer to the "Programming Outlook and Exchange Supplement" on the companion CD. Use it to learn about the CDO library methods and properties not discussed in this chapter. Also refer to the section at the end of this chapter, which contains some tips and tricks for building CDO applications.

Background for Four Sample Applications That Use CDO

The easiest way to learn any new object library is to look at the objects in action. For this reason, the rest of this chapter shows you four sample applications that demonstrate different technologies in the CDO library: a Helpdesk application, a Calendar of Events application, an Intranet News application, and a CDO Visual Basic application. From these four samples, you will learn how to use the CDO library in your applications and become aware of the technical considerations encountered when building CDO applications. Before we dive into the details of these four sample applications, we'll first look at the necessary Exchange Server logon step.

Using the CDO Session Object

Whether you build CDO applications by using ASP or by using some other development tool, the most important point to remember is that you cannot create any other objects in the CDO library if you do not successfully create a Session object. Furthermore, you cannot access any data in Exchange Server unless you successfully log on to the server using the Session object. Before we can look at the code in the CDO applications, you need to understand how to log on to an Exchange server by using the Session object.

The Session object is the top-level object in the CDO hierarchy. It contains session-wide settings and properties that return top-level objects. When using the *CreateObject* method in your applications, you use the ProgID of the Session object—MAPI.Session—to create a CDO object. CDO does not allow you to access any other objects in the library until you have successfully logged on using the *Logon* method of the Session object. The only exception to this is the *SetLocaleIDs* method, which sets the Locale and CodePage IDs for the user.

Using the *Logon* Method

The *Logon* method takes a number of parameters, as shown in the following code; the parameter you use depends on the needs of your application:

```
objSession.Logon( [ProfileName] [, ProfilePassword] [, ShowDialog]
[, NewSession] [, ParentWindow] [, NoMail] [, ProfileInfo] )
```

The two common ways to log on to a CDO session are by passing in a MAPI profile name and by passing in the specific information CDO needs to dynamically generate a profile. Dynamically generated profiles are the preferred method when building ASP applications with CDO. Since ASPs cannot access client profiles, CDO has no way to pull information from a profile located on the user's machine.

Authenticated Logon Using a Profile

To log on using a profile, pass the profile name as the first parameter, *ProfileName*, to the *Logon* method. If you don't know which profile name to use, set the *ShowDialog* parameter to True, and CDO will prompt the user to pick a profile. The second parameter, *ProfilePassword*, specifies the profile password. You could leave this parameter blank and set the *ShowDialog* parameter to True, and CDO will prompt the user for a password. By setting the *NewSession* parameter to False, you can have CDO take advantage of an existing MAPI session, eliminating the unnecessary overhead from creating a new MAPI session on the user's machine. The following code snippet shows you how to use the *Logon* method with a profile named *MS Outlook Settings*:

```
oSession.Logon ProfileName:="MS Outlook Settings", _
   showDialog:=True, NewSession:=True
```

Authenticated Logon Using a Dynamically Generated Profile

When your application is running in an environment where profiles or the ability to prompt a user for a profile might not be available, CDO allows you to dynamically generate a profile for the user by passing in the user's server name and mailbox name to the *Logon* method. To get this information, you can have your application prompt the user for his server name and mailbox name. Alternatively, CDO can pull the default Exchange Server name from the registry by using the *ConfigParameter* properties in

the CDO Rendering library, which you will learn about later in this chapter. For now, the code sample assumes that you know at least one Exchange Server name in your organization. The following code shows you how to log on to CDO using a dynamically generated profile:

```
strServer = "Exchange Server Name"
strMailbox = "User Alias Name (Not Display Name)"
strProfileInfo = strServer + vbLf + strMailbox
oSession.Logon "", "", False, True, 0, True, strProfileInfo
'Check for a valid logon
set oInbox = oSession.GetInbox

if err.number <> 0 then  'Not Successful
   oSession.Logoff
   Response.write "Unsuccessful Logon!"
end if
```

> **NOTE** For the user's mailbox name, don't use the display name, such as *Thomas Rizzo*. Instead use the alias of the user, such as *thomriz*. Also, when using the *ProfileInfo* parameter, attempt to access an item in a CDO message store, such as the first message in the Inbox, since the *Logon* method will return success even if the parameters in *ProfileInfo* are incorrect. If attempting to access items returns an error, the user was not successfully logged on.

Anonymous Access

CDO also allows users to anonymously access the Exchange Server public folder store as well as the Exchange Server directory. Anonymous access must be enabled by the administrator of the Exchange Server system. Also, the administrator or developer can control which folders and which directory entries the anonymous user can see by setting some options in the Exchange Administrator program. These options are discussed in more detail throughout this chapter.

To use anonymous access, you must pass in to the *ProfileInfo* parameter the distinguished name of the Exchange Server and the Anon account. You do this by using the following format:

```
server distinguished name & vbLf & vbLf & "anon"
```

The server's distinguished name takes the form of:

```
/o=enterprise/ou=site/cn=Configuration/cn=Servers/cn=server
```

The *enterprise* parameter corresponds to the Exchange Server organization, and the *site* parameter corresponds to the Exchange Server site you want to access. The following code shows you how to log on using anonymous access:

```
strProfileInfo = "/o=" & "Your Exchange Org" & "/ou=" & _
   "Your Site" & "/cn=Configuration/cn=Servers/cn=" & _
```

(continued)

```
   "Your Server" & vbLF & vbLF & "anon"
oSession.Logon "", "", False, True, 0, True, strProfileInfo
if err.number <> 0 then
   oSession.Logoff
   response.write "Unsuccessful Logon!"
end if
```

HELPDESK APPLICATION

Now that we know how to successfully log on to the Exchange Server, let's take a look at our first sample application, the Helpdesk application. The Helpdesk application is a Web-based application that allows users to submit new help requests. Help technicians can, in turn, use their Web browser to view and answer help requests as well as schedule meetings with the users to go on site to solve their problems. This application allows the technicians to use different views for the help tickets stored in the system so that they can quickly see who the ticket is from, when it was sent, and its status. (A help ticket for the Helpdesk application is shown in Figure 12-2.) When browsing help tickets, the technicians are presented with machine-configuration information from a Microsoft Access database, making it easier for them to track down whether the issue is related to hardware, software, or a user error.

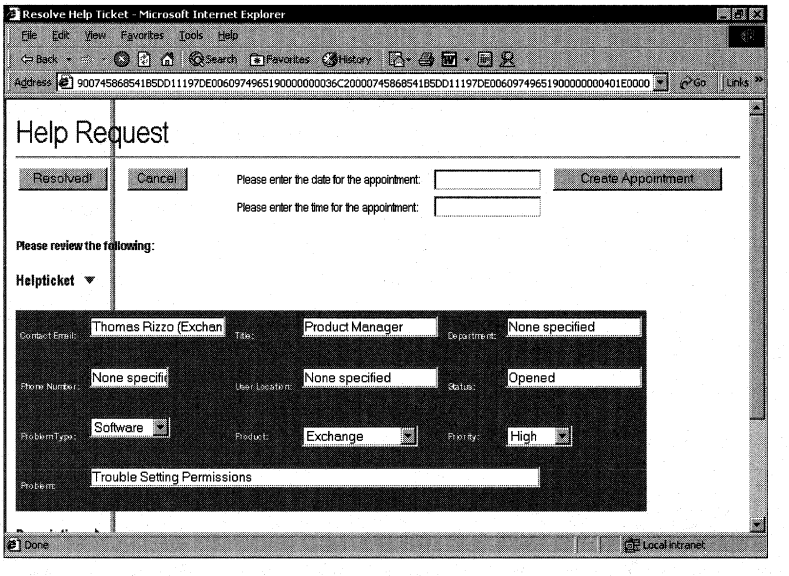


Figure 12-2. *A help ticket in the Helpdesk application. This is the Dynamic HTML version of the application.*

This application is the most complex of all the sample applications in this book, but the code for it is easy to follow and shows you how to use many of the objects in the CDO library. There are actually two versions of the application on the companion

CD. The version you use depends on the Web browser you want to target. One version implements a user interface for the help tickets by using Dynamic HTML (DHTML). The other version uses HTML tables. A screen from the non-DHTML version is shown in Figure 12-3.

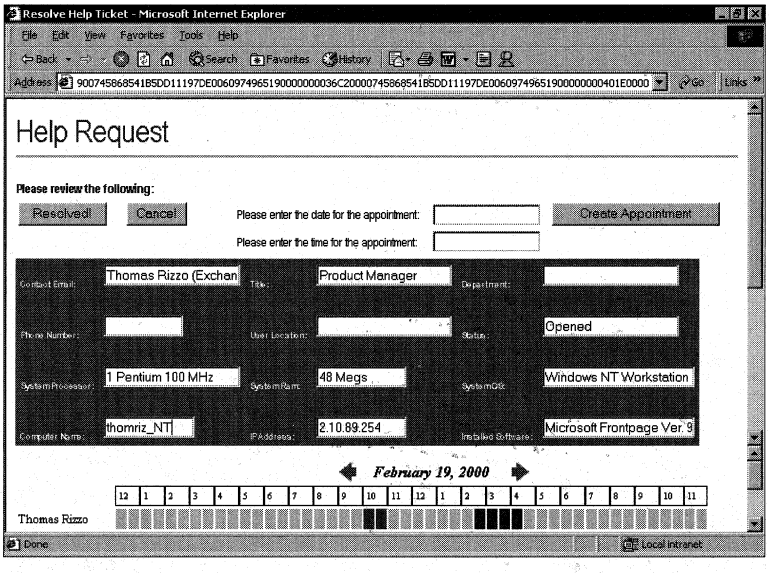


Figure 12-3. *The non-DHTML version of a help ticket.*

Setting Up the Helpdesk Application

Before you can install the application, you must have a Windows NT 4.0 Server or Windows 2000 server and a client with certain software installed. Table 12-1 outlines the installation requirements.

To use the Helpdesk application, you will need to have some e-mail users. You can either select currently set up e-mail users or add new ones using the Exchange Administrator program. Be sure to fill in the directory information for your users; this information should include their office locations, phone numbers, titles, and departments. The Helpdesk application dynamically retrieves this information using the CDO library. If the information is not available in the directory, the application will display the text "None specified" for these fields.

To install the application, copy the Helpdesk folder from the companion CD to your Web server where you want to run the application. If you are going to use a browser that does not support DHTML, such as Microsoft Internet Explorer 3.0, copy the three .asp files from the Nondhtml subfolder to the Helpdesk folder, overwriting the current files. These files will replace the DHTML versions of the Helpdesk with the HTML versions.

Required Software	*Installation Notes*
Exchange Server 5.5 with Service Pack 1 or later installed with Outlook Web Access	Service Pack 3 is recommended.
IIS 3.0 or higher with Active Server Pages	IIS 4.0 is recommended.
CDO library (cdo.dll) and CDO Rendering library (cdohtml.dll)	Exchange Server 5.5 Service Pack 1 installs CDO library 1.21 and CDO Rendering library 1.21. Outlook installs CDO library 1.21.
ActiveX Data Objects	IIS 4.0 installs ADO 1.5. Visual Basic 6.0 installs ADO 2.0. For more information on ADO, consult *http://www.microsoft.com/data/*.
Access 97 (optional)	Install Access if you want to utilize the database access feature. Access 2000 is recommended.
For the client:	
A Web browser or Outlook	For the Web browser, Internet Explorer 5.0 is recommended. You can run the client software on the same machine or on a separate machine.

Table 12-1. *Installation requirements for the Helpdesk application.*

Start the IIS administration program. The user interface you see depends on what version of IIS you have. Create a virtual directory that points to the location where you copied the helpdesk files, and name the virtual directory *helpdesk*. Make sure you enable the Execute Permission for the virtual directory or you will receive an http error when attempting to access the application. You will be able to use the following URL to access your helpdesk: *http://*yourservername*/helpdesk*.

Included with the helpdesk files on the companion CD is a sample Access 2000 database (smsdata.mdb) that allows the application to query for system information about the current user. To use this database, you must set up a system DSN for it on the server machine by using the ODBC Data Source Administrator in the Control Panel. Point the system DSN at the smsdata.mdb file, and name the DSN Helpdesk. Make sure that the read-only flag is unchecked for the smsdata.mdb file. Open smsdata.mdb in Access, and edit the UserID fields for all three tables to reflect the display names of the users you have in Exchange.

> **NOTE** Files copied from CDs have their read-only flags set. When setting up the applications included with this book, be aware that you may need to clear this read-only flag after copying files from the companion CD.

Launch Microsoft Outlook. (You can launch it on any machine because you are going to create a public folder that will contain the help tickets for the application. The only requirement is that the Exchange server with which the IIS server is communicating can access the public folder you create.) Create a new public folder

under All Public Folders. Name the folder *Helpdesk*, and select Task Items as the default item type for the folder.

> **NOTE** You must install the Helpdesk folder under All Public Folders or the application will not work. If you cannot install the application there, you can modify the code contained in the Helpdesk application so that it looks for the folder in another location, or you can retrieve the folder by using its EntryID.

Among the helpdesk files, you will find a file named helpdesk.fdm. This file is a form definition file that will be used to import the correct fields needed by the Helpdesk application. In this case, the form is a Help Request task form with multiple custom fields.

To install the Help Request form, you need to import this file into the Helpdesk public folder. To do this, right-click on the Helpdesk public folder and select Properties. Click on the Forms tab, and click Manage to display the Forms Manager, as shown in Figure 12-4.

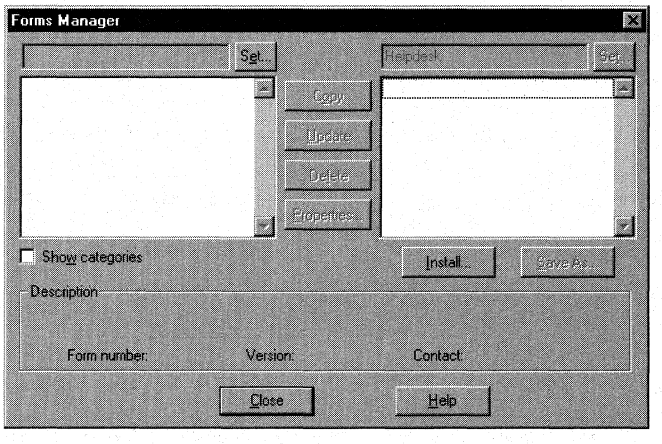


Figure 12-4. *The Forms Manager dialog box.*

In the Forms Manager dialog box, click Install to display the Open dialog box. Select the All Files option from the Files Of Type drop-down list so that Outlook does not search only for files with a .cfg extension. Locate and double-click on the helpdesk.fdm file to display the Form Properties dialog box. Click Cancel, and then click Close and OK. When you have the Helpdesk public folder selected, you should see a new option on the Actions menu named New Helpdesk Request.

You will need to create some views in the Outlook client so that these views are available to the Web browser client. This is one of the powerful features of CDO. These views will include some of the custom fields from the Help Request form just installed. Using the Define Views dialog box and the information in Table 12-2, create the helpdesk, from, and status views for the Helpdesk public folder. (For information about creating views, see the section titled "Views" starting on page 71 in Chapter 3.)

View Name	*Type*	*Fields*	*Group By*
Helpdesk	Table	Flag, From User (select from Help Request form), Received, Subject (select from All Mail Fields)	None
From	Table	Flag (select from Help Request form), Received, Subject (select from All Mail Fields)	From User, in ascending order
Status	Table	From User (select from Help Request form), Received, Subject (select from All Mail Fields)	Flag, in ascending order

Table 12-2 *Information for creating helpdesk, from, and status views.*

Using the Properties dialog box in Outlook for the Helpdesk public folder, set the permissions for your users. Give regular users who will submit help tickets Create Items permission. Give technicians who can submit, view, and resolve help tickets Create Items and Read Items permissions. Figure 12-5 shows a sample permissions setup for the Helpdesk folder. As you will see, the application checks the permissions of the current user for the Helpdesk folder to determine whether the user is a technician or just a regular user. If you do not give yourself at least Read Items permissions, you will not be able to see the menu option View Current Help Tickets in the Helpdesk application.

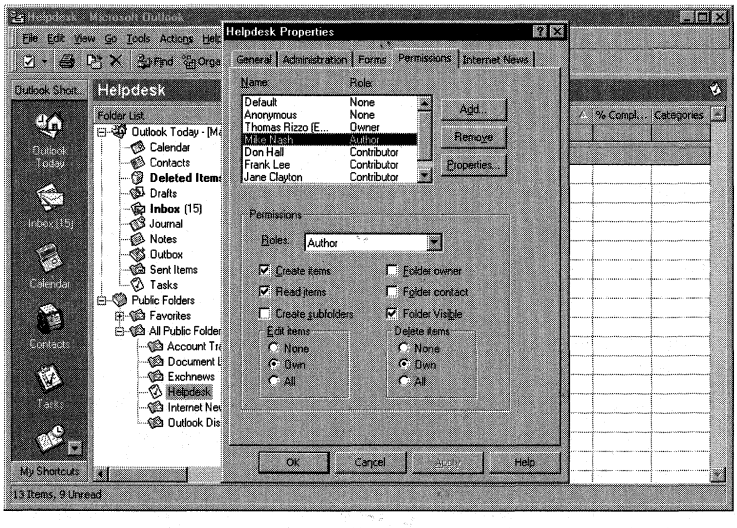


Figure 12-5. *The Permissions tab of the Properties dialog box for the Helpdesk public folder. Regular users have Create Items permissions, and technicians have Create Items and Read Items permissions.*

You're finished setting up the application. Try connecting your browser to *http://* yourservername*/helpdesk* to access the application.

Helpdesk CDO Session Considerations

You need to be aware of certain issues when building ASP-based applications with CDO, like the Helpdesk application. Recall from Chapter 7 that the ASP Session object, which is created when a new user connects to your Web application, maintains the user state for Web applications. ASP, in turn, runs the *Session_OnStart* subroutine in the Global.asa file. When a user's session times out or the user abandons the session, ASP runs the *Session_OnEnd* subroutine in the Global.asa file. This might seem simple enough, but the most common problem developers run into with CDO applications is not putting the correct code in the *Session_OnStart* and *Session_OnEnd* subroutines in the Global.asa. If you do not put the correct code in these subroutines, you could get an ASP 0115 error, which indicates that a trappable error has occurred in an external object. Your ASP application will cease working until you restart the Web server. To help you better understand what you need to do as a CDO developer, and to help you avoid this error in your application, let me explain in more detail exactly what happens when a user logs on and off an Exchange server in a CDO ASP application. I'll do this by showing you the Global.asa file from the Helpdesk application:

```
<SCRIPT LANGUAGE=VBScript RUNAT=Server>
Sub Application_OnStart
   On Error Resume Next
   Set objRenderApp = Server.CreateObject("AMHTML.Application")
   If Err = 0 Then
      Set Application("RenderApplication") = objRenderApp
   Else
      Application("startupFatal") = Err.Number
      Application("startupFatalDescription") = _
         "Failed to create application object<br>" & _
         Err.Description
   End If
   Application("hImp") = 0
   'Load the configuration information from the registry
   objRenderApp.LoadConfiguration 1, _
      "HKEY_LOCAL_MACHINE\System\CurrentControlSet\" & _
      "Services\MSExchangeWeb\Parameters"
   Application("ServerName") = objRenderApp.ConfigParameter("Server")
   Err.Clear
End Sub

Sub Application_OnEnd
   Set Application("RenderApplication") = Nothing
End Sub
```

(continued)

```
Sub Session_OnStart
   On Error Resume Next
   'This is a handle to the security context.
   'It will be set to the correct value when a
   'CDO session is created.
   Session("hImp") = 0
   Set Session("AMSession") = Nothing
End Sub
'While calling the Session_OnEnd event, IIS doesn't call us in
'the right security context.
'Workaround: current security context is stored in Session
'object and then gets restored in Session_OnEnd event handler.
'
'All CDO and CDO Rendering library objects stored in the
'Session object need to be explicitly set to Nothing between
'the two objRenderApp.Impersonate calls below.
Sub Session_OnEnd
   On Error Resume Next
   set objRenderApp = Application("RenderApplication")
   fRevert = FALSE
   hImp = Session("hImp")
   If Not IsEmpty(hImp) Then
      fRevert = objRenderApp.Impersonate(hImp)
   End If
   'Do our cleanup. Set all CDO and CDOHTML objects inside
   'the session to Nothing.
   'The CDO session is a little special because we need to do
   'the Logoff on it.
   Set objOMSession = Session("AMSession")
   If Not objOMSession Is Nothing Then
      Set Session("AMSession") = Nothing
      objOMSession.Logoff
      Set objOMSession = Nothing
   End If
   If (fRevert) Then
      objRenderApp.Impersonate(0)
   End If
End Sub
</SCRIPT>
```

The *Application_OnStart* subroutine is run only once, when the first user connects to the application. As you can see in the code, the first step is to create a new CDO rendering application from the CDO Rendering library. The ProgID for the CDO rendering application is AMHTML.Application. The rendering application is stored in an application-level variable so that you avoid creating multiple AMHTML objects. Your application will perform better if you create the AMHTML object once and then use it throughout all user sessions.

> **NOTE** CDO was formerly named Active Messaging, so you will see both CDO and AM used throughout the CDO library. Consider any objects that are prefixed with AM to be CDO objects.

The *Application_OnStart* subroutine initializes some variables and then uses a method on the CDO rendering application object to retrieve the name of the Exchange server from the Web server's registry. This method allows portability of the code. You don't have to hard-code the name of the Exchange server; the Helpdesk application pulls the information from the registry.

After the first user connects and *Application_OnStart* is finished running, ASP runs the *Session_OnStart* subroutine for all users. *Session_OnStart* clears a session variable named *hImp*. This *hImp* variable is a handle to the security context for the current user. Remember that when a user first browses a Web page in IIS, she is running in the security context of the IUSR_*servername* account. While this account is useful for browsing Web pages anonymously, it is not useful for accessing Exchange Server objects and information because it has no implicit permissions on Exchange Server objects. So when building an authenticated CDO application, you need to force IIS to challenge the current user for the user's Microsoft Windows NT credentials. IIS and CDO can use the Windows NT security context for that user to attempt a logon to the Exchange server. The following code, taken from the Logon.inc file of the Helpdesk application, checks the http variables to make sure that the current user is authenticated by the Web server using the user's Windows NT credentials. In the Helpdesk application, this section of code is called by every ASP page, just in case the ASP session of the user has timed out.

```
Public Function BAuthenticateUser
   On Error Resume Next
   'Response.Write( "In BAuthenticateUser<br>")
   BAuthenticateUser = False
   bstrAT = Request.ServerVariables("AUTH_TYPE")
   If InStr(1, "_BasicNTLM", bstrAT, vbTextCompare) < 2 Then
      Response.Buffer = TRUE
      Response.Status = ("401 Unauthorized")
      Response.AddHeader "WWW.Authenticate", "Basic"
      Response.End
   Else
      BAuthenticateUser = True
   End If
End Function
```

This function searches the AUTH_TYPE server variable to see whether either "Basic" or "NTLM" is contained anywhere in the string. If neither is, the user is unauthenticated and the script sends back a "401 Unauthorized" response and a header that will force the browser to challenge the user for credentials.

Once authenticated, the security context of the user must be saved as a *Session* variable. You must save this security context because IIS uses multiple threads of execution and you cannot be guaranteed that the thread that tries to destroy the session when the user logs off will be the same thread used to create the session. In the *Session_OnEnd* subroutine, the script checks to see whether *hImp* is not empty, implying it is a valid handle. If *hImp* is not empty, *hImp* is used to specify the Windows NT security context to impersonate. Once the CDO and CDO Rendering objects are set to *Nothing* and the session is logged off, the CDO Rendering application object reverts from the authenticated thread to the unauthenticated thread by calling the *Impersonate* method, with *0* as the parameter.

As we step through the Helpdesk application, you will see that every page in the application checks to see whether the ASP session, and therefore the CDO session, has been abandoned or has timed out. If the session has been abandoned or has timed out, the application redirects the user to the logon page so that he can restart his ASP and CDO sessions. Figure 12-6 shows the logon page.

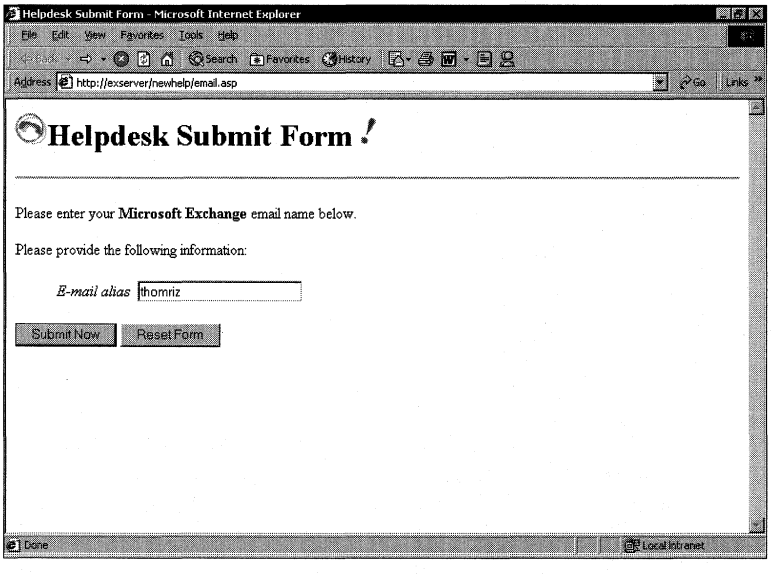


Figure 12-6. *The logon page for the Helpdesk application.*

Logging On to the Helpdesk

You've seen that both the Global.asa and the Logon.inc files help to authenticate users and maintain sessions. We have not yet discussed, however, how you actually use the *Logon* method of the CDO Session object in the Helpdesk application to create

a valid session with the Exchange server. The *Logon* method must be called before you attempt to access any other CDO objects. The script that implements user logons is contained in the Logon.inc file, shown here:

```
<%
'logon.inc. VBScript methods to create and check an
'ActiveMessaging session

'============================
'ReportError
'
'============================
Function ReportError( bstrContext)
   ReportError= False
   If Err.Number <> 0 Then
      Response.Write( "Error: " & bstrContext & " : " & _
         Err.Number & ": " & Err.Description & "<br>")
      Err.Clear
      ReportError= True
   End If
End Function

'============================
'BAuthenticateUser
'Ensures user is authenticated. Note that this implies that
'Basic authentication is enabled on the IIS server.
'============================
Public Function BAuthenticateUser
   On Error Resume Next
   'Response.Write( "In BAuthenticateUser<br>")
   BAuthenticateUser = False
   bstrAT = Request.ServerVariables("AUTH_TYPE")
   If InStr(1, "_BasicNTLM", bstrAT, vbTextCompare) < 2 Then
      Response.Buffer = TRUE
      Response.Status = ("401 Unauthorized")
      Response.AddHeader "WWW.Authenticate", "Basic"
      Response.End
   Else
      BAuthenticateUser = True
   End If
End Function
'============================
'CheckAMSession
'
'Checks for and returns the AM session in the Session object.
'If not found, calls NoSession.
```

(continued)

```
'Call this before emitting any HTML or any redirects,
'authentication, and so on.
'Returns True if session exists or can be created.
'============================
Public Function CheckAMSession()
   On Error Resume Next
   'Response.Write( "In GetSession<br>")
   Dim amSession
   CheckAMSession= False
   Set amSession= Session( "AMSession")
   If amSession Is Nothing Then
      NoSession
      amSession= Session( "AMSession")
   End If
   If Not amSession Is Nothing Then
      CheckAMSession= true
   End If
End Function

'============================
'NoSession
'
'Called when the AM session cannot be found.
'Either creates a session or gets more info from the user.
'Returns only if the session was created.
'
'============================
Sub NoSession()
   On Error Resume Next
   Dim bstrMailbox
   Dim bstrServer
   Dim bstrProfileInfo
   Dim objAMSession1
   Dim objRenderApp
   Dim objInbox
   bstrServer = Application("ServerName")
   if Session("mailbox") is Nothing then
      bstrMailbox = Request.QueryString("Contact_Email")
   else
      bstrMailbox = Session("mailbox")
   end if
   'Must be authenticated to successfully log on
   BAuthenticateUser
   Err.Clear
   Set objAMSession1 = Server.CreateObject("MAPI.Session")
```

```
   If Not ReportError( "create MAPI.Session") Then
      set objRenderApp = Application( "RenderApplication" )
      'Construct the ActiveMessaging profile from server
      'and mailbox name
      bstrProfileInfo = bstrServer + vbLF + bstrMailbox
      Err.Clear
      objAMSession1.Logon "", "", False, True, 0, True, _
         bstrProfileInfo
      If Not ReportError( "CDO Logon") Then
      'To ensure that we are really logged on, we need to
      'try retrieving some data
         Err.Clear
         Set objInbox = objAMSession1.Inbox
         'If ReportError( "Get Inbox"), then
         'the logon is no good. We'll do a little
         'cleanup here.
         If err.number <> 0 then
            objAMSession1.Logoff
            Set objAMSession1 = Nothing
            %>
            <META HTTP-EQUIV="REFRESH" CONTENT="0;
            URL=email.asp"; TARGET="_top">
            <%
            'response.redirect "default.htm"
         End If
         'This will be retrieved in CheckSession.
         'Note that if the logon failed, this is set to 'Nothing'.
         Set Session("AMSession") = objAMSession1
         'Need this to re-create the proper security context
         'in Session_OnEnd.
         Session("hImp") = objRenderApp.ImpID
      End If    'objAMSession1.Logon
   End If   'Server.CreateObject()
End Sub
```

The first function called on every ASP page in the Helpdesk application is the *CheckAMSession* function, which checks to see whether the user already has a valid CDO Session object with the Exchange server. If this function does not find a valid object, it calls the *NoSession* subroutine to log the user on to CDO.

In the *NoSession* subroutine, the variable *bstrServer* is set to the Exchange Server name, which is pulled from the registry in the Global.asa file. (Again, retrieving the Exchange Server name from the registry allows the portability of the code to any Web server and any Exchange server.) The script then checks to see whether the ASP Session variable named *mailbox* contains a valid mailbox alias name. If this variable

is empty, the script attempts to grab the mailbox name from the URL that was passed to the Web server by the logon screen of the application.

The *BAuthenticateUser* function is called to ensure that the user is logged on to Windows NT correctly before the code attempts the CDO Session *Logon* method. After the CDO object is created, its *Logon* method is called. The variable *bstrProfileInfo* is set so that CDO can dynamically create a profile for the user. The script calls the *Logon* method, and if that method returns no errors, the script tries to retrieve the Inbox for the user. You should follow similar steps in your applications because the *Logon* method can return successfully when called with dynamic profiles, even when the server or mailbox name is not valid. If you do not try to retrieve information from the server directly and assume the Session object is valid because the method returned successfully, you will run into many problems.

If the code cannot retrieve the user's Inbox successfully, the code logs off and redirects the user to the logon page. If the code can retrieve the Inbox, it stores the handle to the security context for this user in the *hImp* session variable.

Accessing Folders in the Helpdesk

You don't have much of a Helpdesk application if your users can't enter information and your technicians can't retrieve information. This Helpdesk application uses a public folder to store and retrieve information about ticket status. CDO provides access to public folders through its InfoStores collection, which contains all the different stores or databases that CDO can access. For example, with the InfoStores collection, you can access not only public folders and server-based mailboxes, but also personal folders stored in .pst files or offline replicated folders stored in .ost files. Of course, to enable CDO to access some of these infostores, you must run your CDO application on the client machine. Figure 12-7 shows the InfoStores collection with some of its child objects.

In the Helpdesk application, the InfoStores collection is used to find the public folder store. The following code from default.asp uses a For Each...Next statement to loop through the InfoStores collection and retrieve each store in the collection. Each store is checked to see whether the store name corresponds to the public folder store. If a corresponding store name is found, the For...Each statement is exited.

```
Set objInfoStoresColl = objOMSession.InfoStores
For Each objInfoStore In objInfoStoresColl
   If objInfoStore.Name = "Public Folders" Then
      Exit For
   End If
Next
```

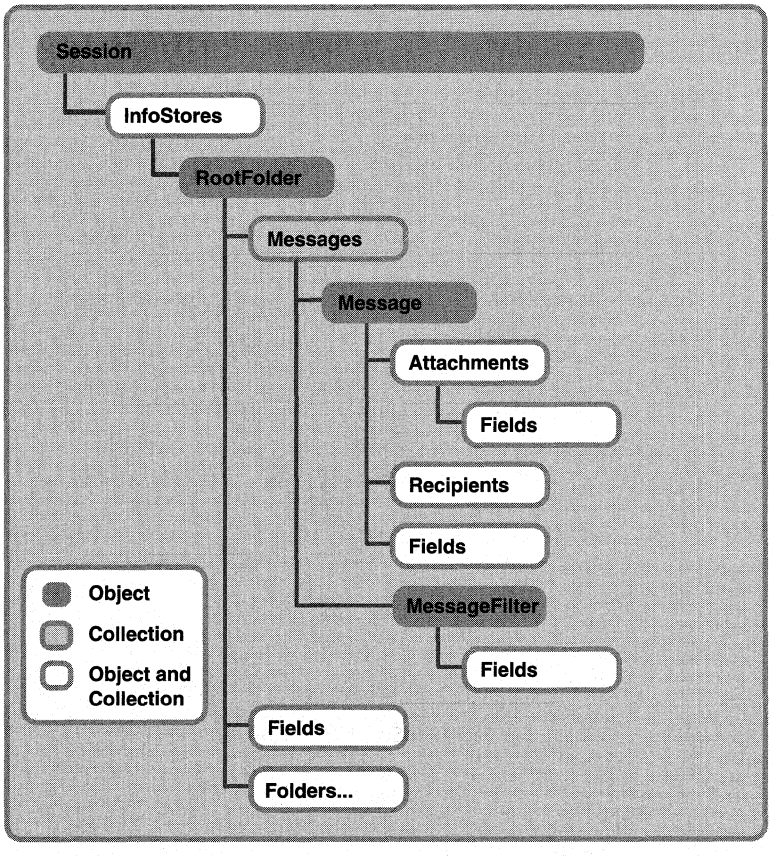


Figure 12-7. *The InfoStores collection in CDO. This collection is used to access data stored in Exchange Server.*

Now that the application has the correct infostore, the correct set of folders needs to be accessed in that infostore. The InfoStore object in CDO has a *RootFolder* property that returns a Folder object representing the root of all the folders. For your mailbox and public folder stores, the *RootFolder* property returns a Folder object named IPM_Subtree. By using that Folder object, you can traverse all folders in your mailbox or top-level public folders. However, there's one caveat with this property: you cannot use the *RootFolder* property to access public folders if your application is running as a Windows NT service. Instead, you need to use the Fields collection on the InfoStores object with the specific property tag *PR_IPM_PUBLIC_FOLDERS_ENTRYID (&H66310102)*. Once you retrieve the EntryID for the root public folder from the Fields collection, you can use the *GetFolder* method

to actually retrieve the root Public Folder. (An EntryID is like a globally unique identifier, or GUID, in that it uniquely identifies the folder in the Exchange Server database.) After retrieving the root folder, you can then use the Folders collection of the root public folder to retrieve the root folder's subfolders. The following code, taken from default.asp, shows the *GetFolder* method in action. It retrieves the folders in the public folder tree, and then recurses the top-level folders as it searches for the Helpdesk folder.

```
'This is the EntryID for the root public folder
bstrPublicRootID = objInfoStore.Fields.Item( &H66310102 ).Value

Set myrootfolder = objOMSession.GetFolder(bstrPublicRootID, _
   objInfoStore.ID)
'Now get the Folders collection below the root
Set myfoldercollect = myrootfolder.Folders
Set recursefolder = myfoldercollect.GetFirst()
'Recurse it until we get the folder we are looking for
While recursefolder.Name <> "Helpdesk"
   Set recursefolder = myfoldercollect.GetNext()
Wend
set objFolder = recursefolder
set Session("HelpdeskFolder") = recursefolder
Session("InfoStoreID") = objInfoStore.ID
```

Once the code finds the Helpdesk folder, it stores the CDO Folder object that corresponds to the Helpdesk folder as well as the unique identifier for the public folder InfoStore object because, from a performance standpoint, recursing the InfoStores and Folders collection in every ASP page in the Helpdesk application to find this information is expensive. Since we now have this information available across the entire session, the other pages use the Session object for the folder whenever access to the Helpdesk folder is required.

Implementing Helpdesk Folder Security

When a user logs on to the Helpdesk application, the options displayed depend on the user's folder permissions. For example, if a user accesses the application and she has only Create Items permission for the Helpdesk folder, the ASP will display only the Submit A Help Ticket and Logoff options, as shown in Figure 12-8. However, if a technician who has Read Items and Create Items permissions for the Helpdesk folder logs on to the application, the ASP will display an additional menu option—View Current Help Tickets, as shown in Figure 12-9.

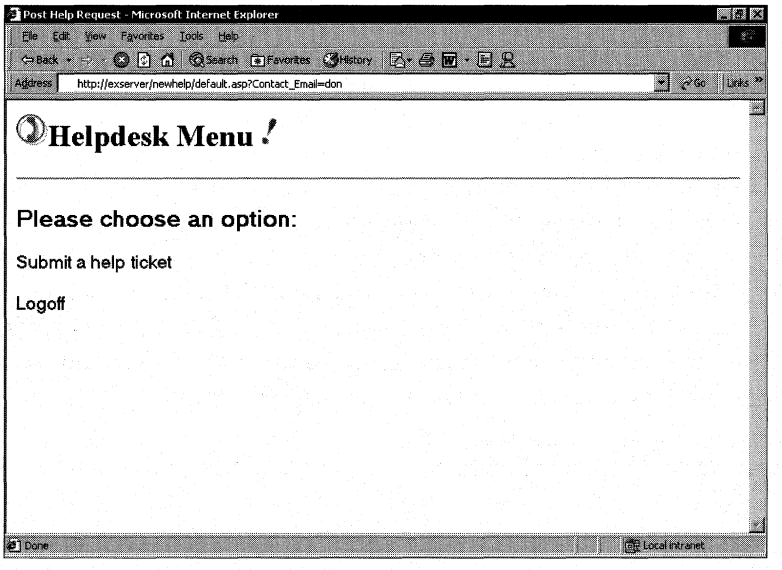


Figure 12-8. *The Helpdesk Menu page, where the user who has logged on has Create Items permissions for the Helpdesk folder.*

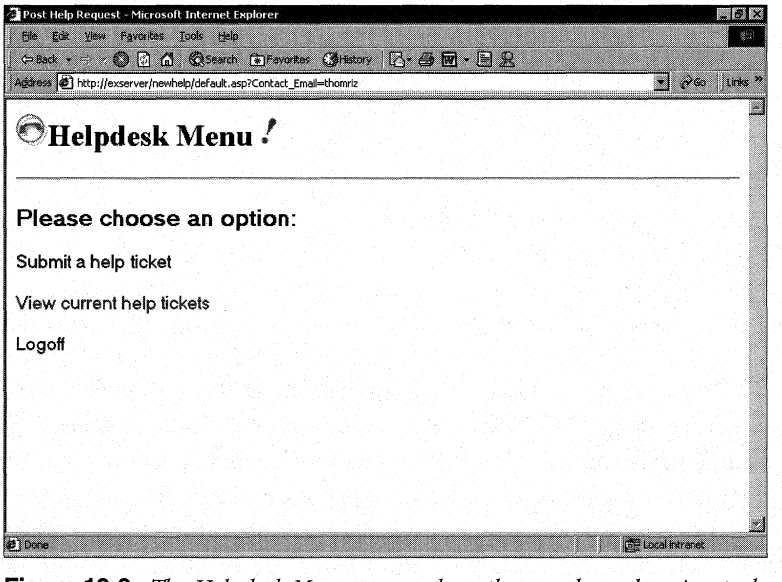


Figure 12-9. *The Helpdesk Menu page, where the user logged on is a technician and has Create Items and Read Items permissions for the Helpdesk folder.*

This functionality is implemented in CDO by using the Fields collection on the Folder object. The Fields collection, a common collection across many of the objects in CDO, allows you to access specific properties stored for an object that CDO does not have an explicit object or property for. With the Fields collection, you can pass a unique identifier that corresponds to the properties you want to retrieve. In the next section of Helpdesk code, which is from the default.asp file, we pass in the unique identifier for the MAPI property *ActMsgPR_ACCESS (&H0FF4003)*, which contains a bitmask of flags corresponding to the user's permissions level for the current Folder object. The code then performs a logical AND on the returned value from the Fields collection by using the known bitmask of the MAPI_ACCESS_READ permission. If the result does not equal zero, the user can read items in the folder, and the ASP displays the View Current Help Tickets link in the helpdesk menu.

```
<h2>Please choose an option:</h2>
<a href="default1.asp">Submit a help ticket</a><P>

<%
'Check permissions on the folder to see whether the user has
'Read access. If the user does, the user must be a technician,
'so display the ability to view help tickets.
nAccess = objFolder.Fields.Item(ActMsgPR_ACCESS)
bCanPost = nAccess And MAPI_ACCESS_READ
If bCanPost <> 0 then
%>
<a href="render.asp">View current help tickets</a>
<% end if %>
<P>
<a href="logoff.asp">Logoff</a>
```

You can pass many types of identifiers to the Fields collection. The Helpdesk application does not use the *CdoPR_CONTAINER_CLASS (&H3613001E)* property, but I point it out to you because it demonstrates the type of information you can access through the Fields collection on CDO objects. It contains the message class of the default type of item contained in the folder. For example, if in Outlook you set the default item type for a Public Folder as contacts, the *CdoPR_CONTAINER_CLASS* property will contain *IPF.Contact*. How do you change the default item type in a folder programmatically through CDO? By using the Fields collection on a folder and this property, as shown in the next bit of sample code, the code changes the default item type of a folder to tasks. The code uses the InfoStores collection to retrieve the mailbox of the user. It then retrieves the root folder of the mailbox by using the *RootFolder* property. From that root folder, it retrieves the contacts folder in the mailbox. The next line of code uses the Fields collection on the folder to set a property that corresponds to the default item type for the folder. Finally, to make the default item-type change permanent, the *Update* method of the Folder object is called, committing the changes to the disk memory.

```
Set oStore = oSession.InfoStores("Mailbox - Thomas Rizzo (Exchange)")
Set oFolder = oStore.RootFolder.Folders("Contacts")
MsgBox "Before Update: " & oFolder.Fields(&H3613001E)
oFolder.Fields(&H3613001E) = "IPF.Tasks"
oFolder.Update
MsgBox "After Update: " & oFolder.Fields(&H3613001E)
```

> **NOTE** For the complete list of the identifiers you can use with the Fields collection in your applications, refer to the cdo.chm on the companion CD. On the Contents tab, look under the Appendixes section for the MAPI Property Tags document.

Retrieving User Directory Information

If the user clicks on the Submit A Help Ticket option on the helpdesk menu, the file default1.asp is called. This ASP file displays a Help Request form, which is shown in Figure 12-10.

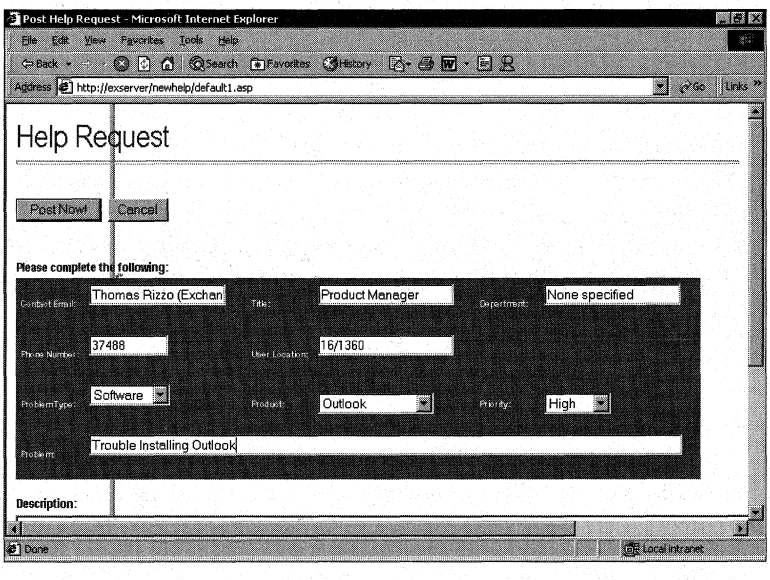


Figure 12-10. *The Help Request form allows users to post information to the Helpdesk application.*

As you can see in Figure 12-10, some information about the user is already entered in the Help Request form. This information is dynamically pulled from the Exchange Server directory to help users save time when filling out requests. The enabling technology for this dynamic lookup involves primarily two CDO objects, the Recipient object and the AddressEntry object. A diagram illustrating the hierarchy of these two objects is shown in Figure 12-11.

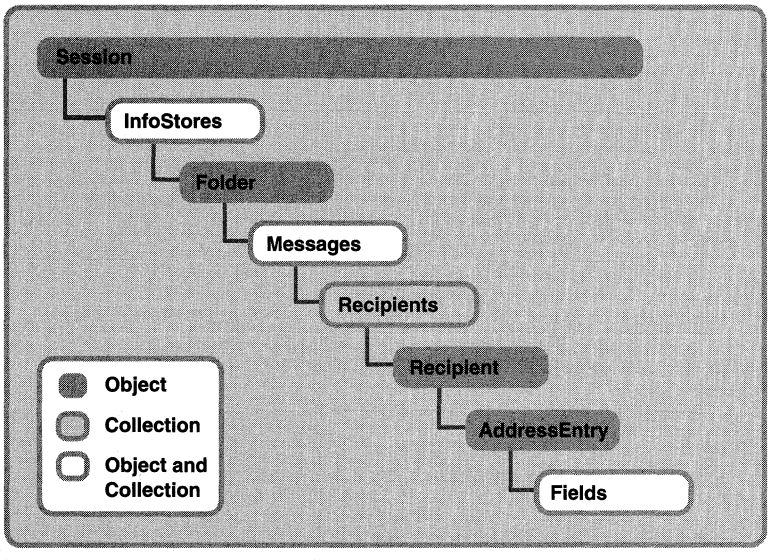


Figure 12-11. *The Recipients collection and Recipient object in CDO.*

The Recipient object represents a recipient of a message. You might be wondering why a Recipient object is used to retrieve directory information from Exchange Server. There are two primary reasons for this. The first is that CDO does not explicitly support querying for directory information without first retrieving the AddressEntry object for the desired user. The second is that adding the name of the user to a message as a Recipient, and then using the *Resolve* method of the Recipient object, is probably the easiest way to retrieve the AddressEntry object for a particular user. After resolving the name, you can call the *AddressEntry* property on the resolved Recipient object to retrieve the corresponding AddressEntry object. Just make sure you destroy the temporary objects you created for the Message and Recipient objects. The AddressEntries collection and AddressEntry object in CDO are shown in Figure 12-12.

The AddressEntry object contains the directory and address information for a user in the Exchange Server system. You can use the built-in properties of this object to access the e-mail address information for a particular user, but to access directory information (department, office location, and phone number), you must use the Fields collection of the AddressEntry object with MAPI property tags. The following code sample, which is taken from default1.asp, shows you how the Helpdesk application implements directory lookup for users:

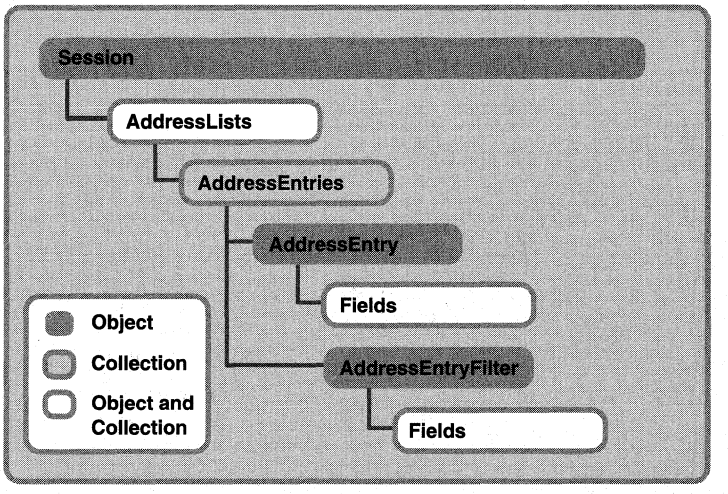


Figure 12-12. *The AddressEntries collection and AddressEntry object in CDO.*

```
Set objmessage = objOMSession.Outbox.Messages.Add
'Create the recipient
Set objonerecip = objmessage.Recipients.Add
'Retrieve the e-mail address from the previous HTML form
objonerecip.Name = Session("mailbox")
'Resolve the name against the Exchange Server directory
objonerecip.Resolve
'Get the address entry so that we can pull out template information
Set myaddentry = objonerecip.AddressEntry
Set myfields = myaddentry.Fields
'The numbers in parentheses are the hard-coded IDs for department,
'title, and so on
Set mydept = myfields.Item(974651422).value
set mytitle = myfields.Item(974585886).value
set myworkphone = myfields.Item(973602846).value
set myoffice = myfields.Item(974716958).value
if mydept = "" then
   mydept = "None specified"
end if
if mytitle = "" then
   mytitle = "None specified"
end if
if myworkphone = "" then
   myworkphone = "None specified"
end if
if myoffice = "" then
   myoffice = "None specified"
end if
set objmessage = Nothing
set objonerecip = Nothing
```

First this code adds a new message to the Outbox of the user by calling the *Add* method on the Outbox folder. Then the code adds a new Recipient object to the message. Since the ASP session contains the display name of the current user, this value is passed in as the *Name* property for the recipient. The code then calls the *Resolve* method on the Recipient object to make CDO check for ambiguous names in the directory. Once the user is resolved, the AddressEntry object for the user is retrieved using the *AddressEntry* property of the Recipient object.

From there, the Fields collection of the AddressEntry object is retrieved. The code then uses some MAPI property identifiers to retrieve specific information from the directory. If the information is unavailable in the directory, the code specifies that the value for the variables be "None specified". To make sure that the temporary Message and Recipient objects are released from memory, the code sets both objects to *Nothing*.

Posting Information in the Helpdesk

Once the user fills in the help ticket information, such as a problem description, the user clicks the Post Now! button on the HTML form. The action of this HTML form sends the user information to another ASP file named posthelp.asp. The posthelp.asp file uses the CDO library to post this information to the Helpdesk public folder by creating a new Message object in the folder. The hierarchy for the Message object and its parent collection, Messages, is shown in Figure 12-13.

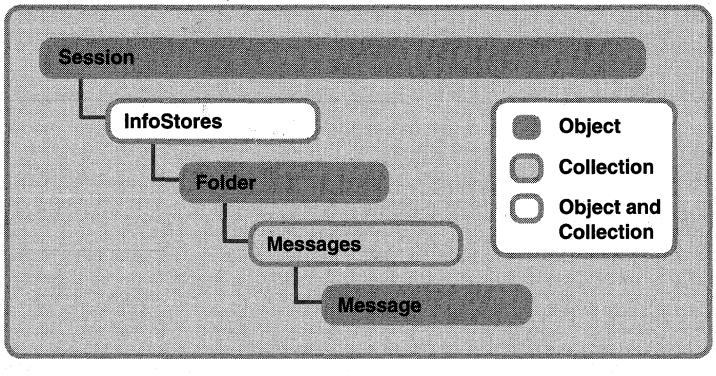


Figure 12-13. *The Messages collection and Message object in CDO.*

The Messages collection is accessed by calling the *Messages* property on a Folder object. The Messages collection consists of Message objects, which you can manipulate to change items in folders or to create new items. To add a new message to a folder, you use the *Add* method of the Messages collection—essentially, it adds a new item to the collection. The type of message created depends on which folder you are calling the *Add* method in. For example, if you call the *Add* method in your Inbox, CDO

will return a new Message object. If you call the *Add* method in your Calendar folder, CDO will return a new AppointmentItem object.

After adding a new item to the collection, you can set the properties for the item and then call the *Send* or *Update* method, depending on the type of item you created. If you used the *Add* method in your Outbox, you would call the *Send* method, because e-mail messages are created in the Outbox. In the Helpdesk application, the script calls the *Update* method because the application does not e-mail new help tickets into the folder but rather posts them into the folder. The following script from posthelp.asp posts the information from the HTML form into the public folder:

```
set objFolder = Session("HelpdeskFolder")
set recursefolder = objFolder
'Add a new message to the public folder
Set oMessage = recursefolder.Messages.Add
'Set the message properties
oMessage.Subject = Request("Subject")
oMessage.Sent = True
oMessage.Unread = True
oMessage.TimeSent = Now
oMessage.TimeReceived = Now
oMessage.Type = "IPM.Task.Help Request"
oMessage.Importance = Request("Priority")
oMessage.Fields.Add "From User", 8, Request.Form("Contact_Email")
oMessage.Fields.Add "Description", 8, Request("Description")
oMessage.Fields.Add "Problem Type", 8, Request("Type")
oMessage.Fields.Add "Product", 8, Request("Product")
oMessage.Fields.Add "Phone", 8, Request("Phone")
oMessage.Fields.Add "User Location", 8, Request("Location")
oMessage.Fields.Add "Flag", 8, "Opened"
'Set the conversation properties
oMessage.ConversationTopic = oMessage.Subject
oMessage.ConversationIndex = objOMSession.CreateConversationIndex
'Post the message
oMessage.Submitted = FALSE
oMessage.Update
Set oMessage = Nothing
```

As you can see in the code, some properties on the new Message object, such as the *Sent*, *Unread*, *TimeSent*, *TimeReceived*, and *Submitted* properties, are explicitly set. These properties need to be set because when you post an item into a public folder as the Helpdesk does, the underlying messaging system does not set these properties for you. They must be set before calling the *Update* method on the Message object for posted items.

The *Sent* property is a Boolean that determines whether the message has been sent through the system. Since we are posting information directly into the folder,

we need to set this property explicitly to *True* before calling the *Update* method. The *Submitted* property is also a Boolean that needs to be set to *True* before we call the *Update* method. The *Submitted* property determines whether the item has been submitted into the messaging subsystem.

The *TimeReceived* and *TimeSent* properties contain dates that tell the user when the message was received by the folder and when the user sent the messages. You should set both of these properties to the current date and time before posting the item into the folder. The easiest way to do this is assign them to the value returned by the *Now* function.

The *Unread* property is a Boolean that represents whether the current user has read the item. This property is not set for you automatically by CDO when posting an item to a public folder. For this reason, you must set this property to *False* before posting your item into the public folder.

Now that we have set the required properties to successfully post a message into a public folder, we can use some of the other properties on the Message object to implement the functionality of our application. Notice in the script that the *Type* property is set to *IPM.Task.Help Request*. The *Type* property corresponds to the message class of the item. By setting this property to a custom message class, Outlook and OWA users can use a custom form that handles that message class to open up the item in the folder.

The *Importance* property in the script is set to the importance established by the user in the Priority drop-down list in the HTML form. This property has three possible values: *CdoLow (0)*, *CdoNormal (1)*, and *CdoHigh (2)*. Setting *Importance* in CDO has the same effect as adding the exclamation point icon or the down arrow icon to an e-mail message in Outlook.

The *Subject* property is set to whatever the user types in the Problem text box of the Helpdesk form. It is rendered as a hyperlink later in the application so that from their Web browser, technicians can click on a specific problem and drill down to the specifics about the help ticket and the user who submitted it.

The Fields collection on the Message object is used in the script to add custom fields to the item programmatically—a powerful technique, because any items you create in CDO can use it. These custom properties, or fields, are then accessible from Outlook. Since the Exchange Server database is a semistructured database, you can even change or add new properties to the items in a folder without worrying about breaking a schema. This means that in your application, every item and its properties can hold different data. Your application can dynamically add new properties to items depending on the input of the user.

The way you add new custom properties to an item is by using the *Add* method of the Fields collection. This is the syntax for the *Add* method:

```
Set objField = objFieldsColl.Add (Name, Class [, Value] [, PropsetID])
```

SEMISTRUCTURED DATABASES

Exchange Server is a semistructured database, as compared to Microsoft SQL Server, which is completely structured. With SQL Server, you need to define your schema before you can start adding data, and the schema is usually fixed. With Exchange Server, every message can be different. For example, one message might have 0 attachments and another might have 15 attachments. Exchange Server is designed to efficiently handle this varying data.

Passing the last two parameters to this method and setting the value returned by this method are optional. You can see in the preceding script that the code does not set an object variable to the return value of this function.

The first parameter the *Add* method takes is the name of the custom property. This name is limited to 120 characters, and if you attempt to exceed this limit, CDO will return an error. The second parameter is the class, or data type, that you want to store in the property. The *class* parameter should pass one of the following values: *vbArray (8192)*, *vbBlob (65)*, *vbBoolean (11)*, *vbCurrency (6)*, *vbDataObject (13)*, *vbDate (7)*, *vbDouble (5)*, *vbEmpty (0)*, *vbInteger (2)*, *vbLong (3)*, *vbNull (1)*, *vbSingle (4)*, *vbString (8)*, or *vbVariant (12)*. The most common data type you will use in your applications is *vbString (8)*. As you can see in the Helpdesk script, all custom properties on the posted item use the string data type.

The third parameter, which is optional, is the value for the property. Usually you want to pass this parameter to the method so that you do not have to write extra code to initialize the custom property with the value.

The fourth and final parameter is the *PropsetID*. The *PropsetID* parameter is a GUID that uniquely identifies the MAPI property set to which the custom field should belong. In your applications, you will almost always use the default property set, which is assumed if you omit the *PropsetID* parameter. Only if you are developing a custom MAPI application that uses its own property sets will you ever need to set *PropsetID*.

In addition to adding new custom properties, the script sets the *ConversationTopic* and *ConversationIndex* properties for the help ticket before posting it into the folder. These properties are used by both CDO and Outlook to allow you to create threaded views of information in your folders. Since your users might want to create these types of views in your applications, you should set these properties in your code.

The *ConversationTopic* property is a string that describes the subject of the conversation. All the items in the same conversation have the same property value, and since *ConversationTopic* corresponds to the overall subject of the conversation, the most logical value for it is the *Subject* property of your message.

The *ConversationIndex* property is a hexadecimal string that represents the relationship between items in the same conversation. This property is used by CDO and Outlook to determine which items are replies to other items and how to thread these items in a view. Since you do not want multiple messages with the same index, CDO provides a method for you to generate unique conversation index values—the *CreateConversationIndex* method on the CDO Session object. As you can see in the Helpdesk code, all you need to do is call *CreateConversationIndex* and assign its value to the *ConversationIndex* property for your item.

After setting all these properties, the script calls the *Update* method on the Message object, and a new help ticket is created in the folder. If you do not call the *Update* method, CDO will not commit your changes to the public folder.

Rendering the List of Helpdesk Tickets

When creating a Web application, one of the hardest aspects to design is the user interface. You have to worry about using HTML tables to line up content, and you have to make sure that these tables have the correct format and spacing to appear properly in a browser. The beauty of CDO is that you do not have to worry about the user interface. The CDO library has a companion library, named the CDO Rendering library, that provides objects that automatically convert Exchange Server information to an HTML format in a preset layout. Figure 12-14 shows the relationships among the objects and collections of the CDO Rendering library.

The CDO Rendering library can not only render simple types of information such as your Inbox, but it can also leverage any custom table views you create in Outlook. For example, you could use the CDO Rendering library to render your Inbox as HTML, grouped by who sent you the message. The CDO Rendering library provides this functionality with a minimal amount of coding, as you will see. Plus, the formats that the library uses to render information to HTML are customizable, so if you want to change the color or the font of items that meet particular criteria or contain particular properties, you can easily do this by using the Rendering objects. Figure 12-15 shows an HTML view of help tickets in a folder, created using the CDO Rendering library.

Similar to the Session object in the CDO library, the RenderingApplication object of the CDO Rendering library is the parent object from which all other objects in the library are derived. To create a RenderingApplication object, you need to use the *CreateObject* method and pass the ProgID of the CDO Rendering library, which is AMHTML.Application. In the Helpdesk application, the RenderingApplication object is created in the Global.asa file and given application scope in ASP so that all sessions in the Helpdesk application can create individual objects from the global RenderingApplication object. This is a good practice to use in your CDO ASP applications.

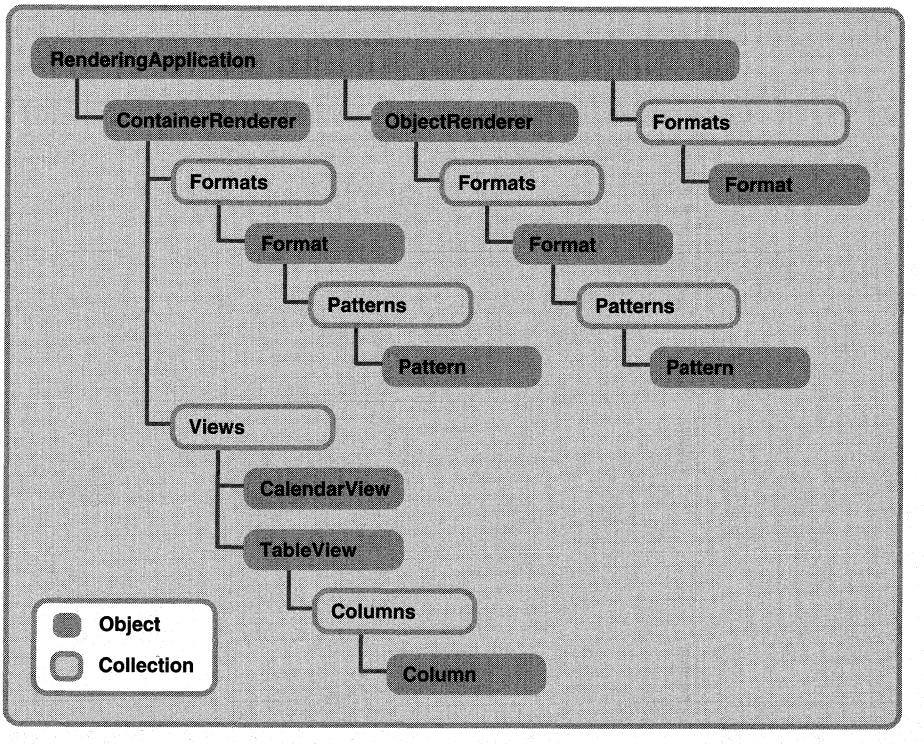


Figure 12-14. *CDO Rendering library.*

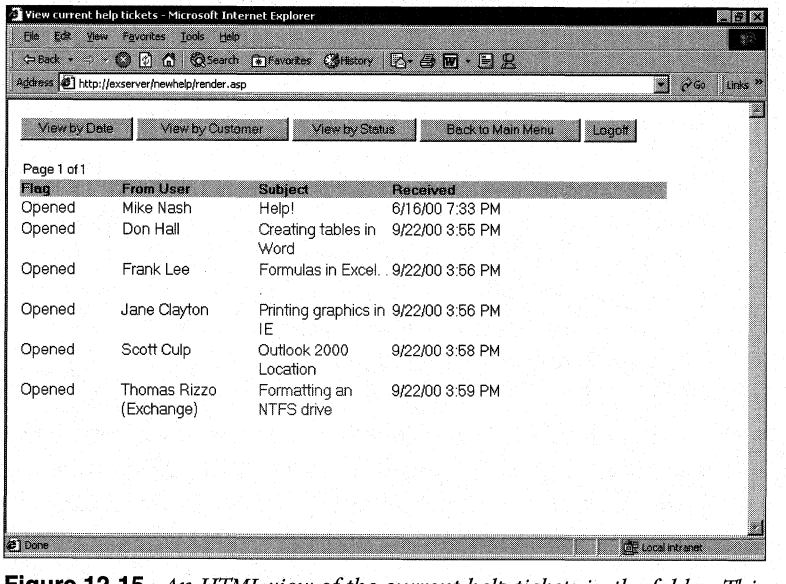


Figure 12-15. *An HTML view of the current help tickets in the folder. This was created using the CDO Rendering library.*

The RenderingApplication object contains a number of properties that you can set, such as the code page or virtual root, that can be used throughout all the rendering objects created from RenderingApplication. When you first create your RenderingApplication object, most of the properties on the object will be filled in with default values. Most of these default values do not need to be changed unless you are developing completely customized applications that cannot use the defaults.

The most important property and the two most important methods of the RenderingApplication object to learn about are the *ConfigParameter* property, the *CreateRenderer* method, and the *LoadConfiguration* method. This property and these methods will be used in almost all of your ASP CDO applications. The *LoadConfiguration* method and the *ConfigParameter* property work together to tell the CDO Rendering library where to pull configuration information from—such as the registry or the Exchange Server directory. This location information is used to retrieve specific configuration data from the selected data source. In the Global.asa file for the Helpdesk application, the name of the Exchange server that CDO communicates with is retrieved by using the *LoadConfiguration* method and the *ConfigParameter* property. The *ConfigParameter* property can retrieve other types of configuration parameters, such as whether anonymous access is enabled on the Exchange server, the organization and site names for the Exchange Server directory, and whether the http protocol is even enabled on the Exchange server.

The *CreateRenderer* method creates a new rendering object, which is attached to the current RenderingApplication object. This method takes an integer argument that specifies the type of rendering object to create. The values for this integer parameter can be either *CdoClassContainerRenderer (3)* or *CdoClassObjectRenderer (2)*. In your applications, use the object renderer to render only specific properties on items, such as the subject or the text of the item. You should use the container renderer to render rows of information, such as all the messages in a folder.

In the render.asp file for the Helpdesk application, the following script first gets the global RenderingApplication object and then calls the *CreateRenderer* method to create a container renderer. The application uses a container renderer to display all the items in the Helpdesk folder rather than specific properties on a specific item.

```
'Create Rendering Application
set objRenderApp = Application("RenderApplication")
'Create Container Renderer id=3
Set objrenderer = objRenderApp.CreateRenderer(3)
```

The application then sets the *DataSource* property for the container renderer to the Messages collection in the Helpdesk folder. With any type of rendering object, you need to set the *DataSource* property because it tells the rendering object what data to actually convert to HTML. The container renderer can accept an AddressEntries, a Folders, a Messages, or a Recipients collection as its data source.

```
Set oMsgCol = objFolder.Messages
'Set the data source for the Rendering object to the
'helpdesk public folder
objrenderer.DataSource = oMsgCol
```

The next step in the code figures out whether the application passed along a custom Outlook view name with the query string. If it did, the script uses that custom view name as the default view for the Rendering object. This is an important point. You can use custom Outlook views as views in your Rendering objects. This means that you do not have to create views on the fly in your CDO applications, but can instead create views in Outlook and then leverage those views in your application. The Rendering object will support views that contain sorting, grouping, and filtering—support that will save you many hours of coding custom views.

If the application did not pass a view name with the query string, the default view named Helpdesk is used to render the tickets in the folder, as shown in the following code:

```
'Get the requested view from the query string
requestedview = Request.QueryString("view")
'Get the Folder Views collection
set objviews = objrenderer.views
'If there is no selected view, set it to Helpdesk view
'created in Outlook
if requestedview = "" then
   requestedview = "Helpdesk"
end if
'Search the Views collection until you find the view
i=1
While objviews.item(i).Name <> requestedview
   i=i+1
Wend
set objview = objviews.Item(i)
```

The next step in setting up the container renderer is to enable a hyperlink on a field in the view so that technicians can click on the hyperlink to retrieve the information contained in the ticket. If you do not include this step in your CDO applications, the rendering objects will return HTML without any clickable links. To create hyperlinks, we must first select the column in the view for which we want to create the hyperlink. In the Helpdesk application, the third column in the view is always used to create the hyperlink. The way you access the third column is by using the Columns collection and the Column object of the TableView object that is represented by the custom Outlook view. The Column object has a property called *RenderUsing* that allows you to specify the HTML code to use when rendering that specific column. You specify not only the HTML but also substitution tokens within percent signs in the *RenderUsing* property, which CDO will replace with actual values

when rendering the information to the Web browser. The two most common substitution tokens you will use are %obj% and %value%.

Use the %obj% token when you want to place the unique identifier for the item as a string in your HTML. This token is used in the Helpdesk application so that the hyperlink on the third column passes the unique identifier for the ticket to the next ASP file, framemsg.asp. Use the %value% token when you want to place the actual value of the property into the HTML returned by the rendering object. The Helpdesk application uses this token to display the actual third-column property value in the view. For the Helpdesk view, this property value is the subject of the message. The following code shows you how to use both these substitution tokens as well as the *RenderUsing* property and Column object:

```
set objcolumns = objview.Columns
set objcolumn = objcolumns.Item(3)
'Change the column renderer so that it renders the subject field
'as an ahref with the entry ID of the message
objcolumn.RenderUsing = _
   "<a href='framemsg.asp?entryid=%obj%'>%value%</a>"
```

The final step in enabling the container renderer is to set the *CurrentView* property as the custom view just modified by the application. To do this, the application sets the *CurrentView* property equal to the TableView object we just modified, as shown in the following code:

```
'Set the current view equal to the view just selected
objrenderer.CurrentView = objview
```

To actually render the information to HTML and return it to the browser, you must call the *Render* method on your ContainerRenderer object. The *Render* method takes four parameters. The first parameter is a Long data type that determines the style that the data should be rendered in. This parameter has two possible values—*CdoFolderContents (1)* and *CdoFolderHierarchy (2)*:

- *CdoFolderContents* is used to render the actual contents of the data source and not the child objects.
- *CdoFolderHierarchy* is used to render child folders for a Folders collection. If you wanted to build an HTML page that displays a folder hierarchy for users to scroll through, you would use the *CdoFolderHierarchy* style.

The second parameter, which is optional, also is a Long data type and specifies the page number at which rendering should begin. In the CDO Rendering library, you can have CDO automatically break up the content of a data source into data pages so that when the HTML is rendered by the library, the length of the resulting HTML table is not massive. By default, CDO will break the content at every 25 rows in the HTML table. CDO does this to enhance the performance of your application as well

as make it easier for your users to read the information. You can change the default number of rows rendered by setting the *RowsPerPage* property on the Container-Renderer object.

The third parameter is for internal use only by the CDO Rendering library. You should always pass *0* as the value to it. The final parameter is the Active Server Response object to which you want to send the HTML output from the *Render* method. This parameter should always be *Response* if you want to render the information to the browser.

When you have large amounts of information to render, you should be aware that CDO does not automatically generate a way to navigate rows on multiple pages nor will it tell you that there are multiple pages of information you need to render. Therefore, in your application, you must provide navigation elements if the number of rows in the table is larger than the value of the *RowsPerPage* property. You also must remember which page the user is currently rendering as well as the total number of pages. If the number of help tickets exceeds the *RowsPerPage* property, the render.asp file for the Helpdesk application will display graphical navigation arrows as well as text indicating the current page of information. When a user clicks on the Next Page or the Previous Page arrows, the current page variable is either incremented or decremented, and this value is sent with the query string. The ASP script retrieves the value and renders the correct content on the page. If there are no previous or next pages, the graphical navigation elements are not displayed. The following code from render.asp handles viewing help tickets on multiple pages:

```
'Calculate total number of pages
intMessageCount = oMsgCol.Count
numrows = objrenderer.RowsPerPage
intPages = (intMessageCount - 1) \ numrows
intPages = intPages + 1
intCurPage = CInt(Request.QueryString("curpage"))
if intCurPage > intPages or intCurPage < 1 then
   'Initialize it
   intCurPage = 1
elseif intMessageCount < 1 then
   intCurPage = 1
end if

<% if intCurPage <> 1 then %>
<a href=
"render.asp?view=<%=requestedview%>&curpage=<%=intCurPage-1%>">
<img src = "left.gif" Align="Middle" border=0 Alt="Previous Page"></a>
<% end if %>
  Page <%=intCurPage%> of <%=intPages%> 
<% if intCurPage <> intPages then %>
<a href="render.asp?view=<% response.write requestedview %>
```

(continued)

```
&curpage=<%=intCurPage+1%>">
<img src = "right.gif" Align="Middle" border=0 Alt="Next Page"></a>
<% end if %>
<table border="0" width="65%">
   <tr>
      <td><% objRenderer.Render 1,intCurPage,0,Response %></td>
   </tr>
</table>
```

Notice how you can easily calculate the total number of pages you need to render to completely show all the information in your application. Take the number of messages minus 1, integer divide that number by the number of rows per page set for the ContainerRenderer object, and then add 1 to that value. Therefore, if the number of messages is less than or equal to the number of rows per page, the integer division will return 0. This value will be incremented by 1, indicating there is one page of information.

Rendering the Actual Help Ticket

When the technician clicks on one of the hyperlinks in the rendered list of help tickets, the application calls another ASP file, message.asp, to render the actual ticket for the technician, as shown in Figure 12-16. The technician can then view a number of different items for the ticket, resolve the ticket, or schedule a meeting with the customer who submitted the ticket.

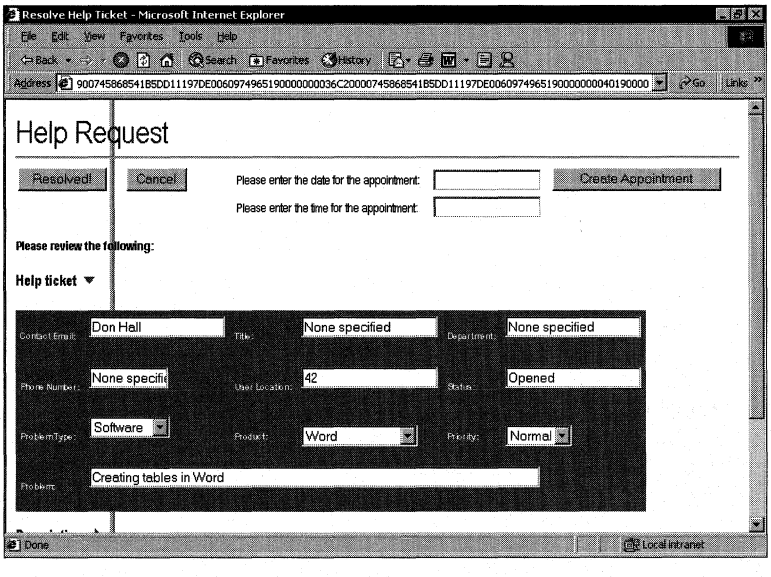


Figure 12-16. *The DHTML version of a help ticket rendered when a technician clicks on a hyperlink from a list of tickets.*

The Helpdesk application uses DHTML to simplify navigating information contained in a ticket. It also uses ADO and queries an Access database to pull out the relevant information about the user's machine. This portion of the application shows you how you can combine CDO and ADO to make rich Web applications that access information from two types of data sources: Exchange Server and an ODBC database.

In the code for the help ticket page, the EntryID of the help ticket link that the user clicked on is retrieved from the Request.QueryString collection. The script then calls the *GetMessage* method of the CDO Session object to retrieve the message from the Exchange Server database. The *GetMessage* method is an easy and fast way to retrieve information from Exchange Server if you know the unique EntryID for the desired message. If you do not know the EntryID, you will need to search the folder where the message is stored or use a MessageFilter object to filter out only your message. The MessageFilter object is discussed in the Calendar of Events application later in this chapter.

The script then sets the *Unread* property of the message to False and calls the *Update* method of the Message object to save that property back to the database. Since CDO does not automatically update the *Unread* property for you, you must set it in code. After the code sets *Unread*, when the technician goes back to the table of tickets in the folder, the code displays all messages read by the technician as nonbold tickets. This functionality of not bolding read messages is automatically provided to you by the Rendering objects and ultimately makes your application easier to use. Also, the Exchange Server database maintains a per-user *Unread* property so that each technician will receive different read and unread messages in the folder according to what each has actually read. The VBScript code for this functionality is shown here:

```
'Get the EntryID for the message from the query string
set objMessageID = Request.QueryString("entryid")
'Get the message by its ID
set oMessage = objOMSession.GetMessage(objMessageID, _
   Session("InfoStoreID"))
'Set the message as read
oMessage.Unread = FALSE
'Update the message in the folder
oMessage.Update
```

The application then pulls out the status of the ticket, either *Opened* or *Done*, and also the name of the user who submitted the ticket. An ADO Connection object is created, and a connection to the Access database is established by the application. In ADO, you can specify values for the *Open* method on the Connection object to determine which OLE DB data source you want to open. In this case, the DSN name *Helpdesk*, which we created earlier on the machine, is passed as the parameter for the *Open* method. After establishing a connection, three queries are executed against

three database tables to figure out the machine configuration for the current user. This is accomplished by using the name of the user retrieved from the help ticket. These queries use the *Execute* method of the ADO Connection object. This information is then used later in the form. The following code shows the ADO connection and queries. (For more information about ADO and its object library, you should visit the Microsoft Web site at *http://www.microsoft.com/data*.)

```
'Start the ADO connection
on error resume next
Set Conn = Server.CreateObject("ADODB.Connection")
Conn.Open "Helpdesk"
Set RS = Conn.Execute( _
   "Select SystemChipType, SystemChipSpeed, SystemChipCount, " & _
   "SystemOS, SystemRAM FROM tblMachine " & _
   "WHERE Userid like '" & objuser & "';")
Set RSIP = Conn.Execute( _
   "Select CompName, IPAddress FROM tblNetwork " & _
   "WHERE Userid like '" & objuser & "';")
Set RSSoft = Conn.Execute( _
   "Select SoftwareName, SoftwareVersion FROM tblSoftware " & _
   "WHERE Userid like '" & objuser &"';")
```

The script retrieves the user information from the Exchange Server directory and stores it in variables. (You saw this code earlier when submitting a help ticket.) Then the body of the actual ticket is displayed using DHTML. The DHTML code includes some JavaScript to allow the user to dynamically expand or collapse the different sections of the help ticket, such as system information or the description of the problem.

Creating the Calendar Information

The one interesting section in the body of the HTML is the drop-down section of calendaring information, shown in Figure 12-17. This drop-down section allows technicians to view the free/busy information for the user and for themselves. By providing the free/busy information at the bottom of the help ticket page, a technician can quickly find time slots that are open for both the technician and the user.

> **NOTE** To obtain up-to-the-minute calendar information for a user on the Help Request page, it might be necessary to adjust the calendar settings in Outlook. By default, Outlook updates the calendar free/busy information on the server every 15 minutes. You can decrease this time in Outlook by choosing Options from the Tools menu, clicking Calendar Options, and then clicking Free/Busy Options.

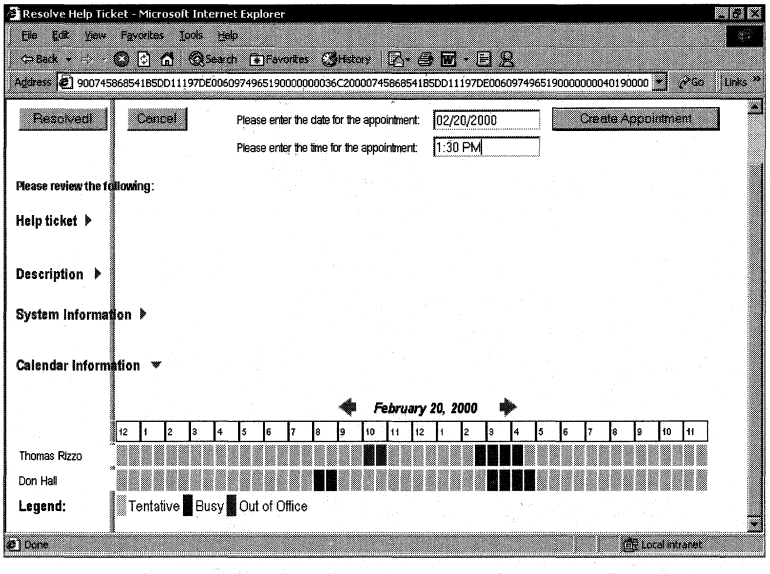


Figure 12-17. *The help ticket showing the calendar information.*

This calendar information is created by using the calendaring functionality of the CDO library. The code for the drop-down section is shown here:

```
<%
bcGrayM = "#c0c0c0" 'gray
bcGrayD = "#909090"
buildmonth = Request.QueryString("m")
buildday = Request.QueryString("d")
buildyear = Request.QueryString("y")
builddate = buildmonth & "/" & buildday & "/" & buildyear
dtCurrentDay =  DateValue(builddate)
arrBGColors= _
   Array(bcGrayM, "#99ccff", "#0000ff", "#940080", "#ff0000")
Dim MeetingPlanner(2)
%>
<B><FONT STYLE="ARIAL NARROW" SIZE=3><SPAN style="cursor:hand"
CLASS=ex TITLE="Calendar Information"
ID="CalInfo" onclick="checkExpand()" myArrow=4>Calendar Information
</SPAN></FONT></B>
<IMG WIDTH=15 HEIGHT=13 SRC="addarrow.gif" ID="imgArrow4"><P>

<TABLE ID = "CalInfoChild" style="display:none" BORDER = 0>
<TR>
```

(continued)

```
<td nowrap=nowrap width="355" align="right">
<A href="message.asp?entryid=<%response.write objMessageID%>
&m=<%=Month(dtCurrentDay-1)%>&d=<%=Day(dtCurrentDay-1)%>
&y=<%=Year(dtCurrentDay-1)%>" target="main">
<img src="left.gif" border=0 alt="Previous Day"></a></td>
<td nowrap=nowrap width="10"></TD>
<td nowrap=nowrap border=0><i><b>
<%=MonthName(Month(dtCurrentDay)) & " " & Day(dtCurrentDay) & _
", " & Year(dtCurrentDay)%></b></i></TD>
<td nowrap=nowrap border=0 width="10"></td>
<td nowrap=nowrap width="20" align="right">
<A href="message.asp?entryid=<%response.write objMessageID%>
&m=<%=Month(dtCurrentDay+1)%>&d=<%=Day(dtCurrentDay+1)%>
&y=<%=Year(dtCurrentDay+1)%>" target="main">
<img src="right.gif" border=0 alt="Next Day"></a></td>
</Table>

<table ID = "CalInfoChild2" style="display:none" cols=50
bordercolor=#FFFFFF border=1 cellspacing=0 cellpadding=0>
<tr>
<td nowrap=nowrap width="102">
<td colspan=2> </td>
<% For idx = 0 to 23 %>
<td bordercolor=#000000 colspan=2 align=left width="24">
<font size=-2%> <%= CStr((((idx + 11) Mod 12) + 1))%>
</font></td>
<% Next %>
</tr>
</TABLE>
<%
'Loop from 3 to 4
for x = 3 to 4
   if x = 3 then
      set objRecip = objOMSession.CurrentUser
   else
      set objRecip = objonerecip
   end if
   'Initialize the string
   aFB = ""
   aFB = objRecip.GetFreeBusy( dtCurrentDay, dtCurrentDay+1, _
      30, true)
   szFreeBusy = aFB
%>
   <table ID = "CalInfoChild<%=x%>" style="display:none" Border=0>
   <tr bordercolor=#000000>
   <td nowrap=nowrap bgcolor="#ffffff" width="100"><font size=-1>
   <%= objRecip.Name %><br></font></td>
<%
```

```
   if Len(szFreeBusy) = 0 then
%>
      <TD><font size=2>Free/Busy Information is not available
      </font></TD>
<%
   else
      For idy = 1 to 48
         sCell= Mid( szFreeBusy, idy, 1)
%>
         <td bgcolor=<%= arrBGColors(CInt(sCell)) %> width="9">
          </td>
<%
      Next
   end if
Next
%>
</table>
<TABLE ID="CalInfoChild5" style="display:none" border=0>
<TR>
<td nowrap=nowrap width="100"><b>Legend:</b></TD>
<TD bgcolor=<%= arrBGColors(1) %>>  </TD>
<TD> Tentative </TD>
<TD bgcolor=<%= arrBGColors(2) %>>  </TD>
<TD> Busy </TD>
<TD bgcolor=<%= arrBGColors(3) %>>  </TD>
<TD> Out of Office </TD>
</TABLE>
```

First the code tries to retrieve, from the query string, the day, the month, and the year values passed by the application. When the user first clicks on a hyperlink from the list of help tickets, these date values are filled with the current day, month, and year from the Web server machine. The code then builds a date from the values, such as 10/12/2000. This is done so that the string containing the date value can be used as a parameter to the *DateValue* function in VBScript. This function will return a Variant of the subtype Date so that we can use it in the rest of the code.

The application then builds an array of information. This array, named *arrBGColors*, comprises background colors used to render the free/busy information to the Web browser. These colors correspond to the Outlook colors for rendering calendar information, such as blue for busy slots, light blue for tentative slots, and purple for out-of-office slots in the calendar. You'll understand why we place these colors into an array when we look at the *GetFreeBusy* method in CDO, and see how the values are returned from this method and parsed by the application.

The next step in the code is to render the navigation elements to move to the previous or next day in the calendar and also to print out the current day the technician is viewing. The navigation elements to move to the next and previous days are

implemented using hyperlinks, which pass the next or previous date, broken up into day, month, and year, across the query string to message.asp. To retrieve the day, month, and year from the *dtCurrentDay* variable, which has the form 12/31/2000, the VBScript functions *Day*, *Month*, and *Year* are used.

The code then uses a For...Next loop from 0 through 23 to draw a table that creates the time values across the top of the free/busy information. Starting at 12 AM, this code draws table elements until 11:30 PM the same day. Adding 11 to the current index of the loop, using the Mod operator to return the remainder when divided by 12, and then adding 1 to that value produces the correct identifiers for the time slots. The code uses the *CStr* function to convert the number returned by the formula to a string value.

The next portion of the code uses the calendaring features of CDO. Figure 12-18 shows some of the calendar-related objects in the CDO library. The CDO library provides extensive support for building calendaring applications: the Helpdesk application shows how to use the *GetFreeBusy* method and the meeting request functionality of CDO, and the Calendar of Events application, which we'll examine later in this chapter, shows appointment filtering and rendering of calendars using the Rendering library.

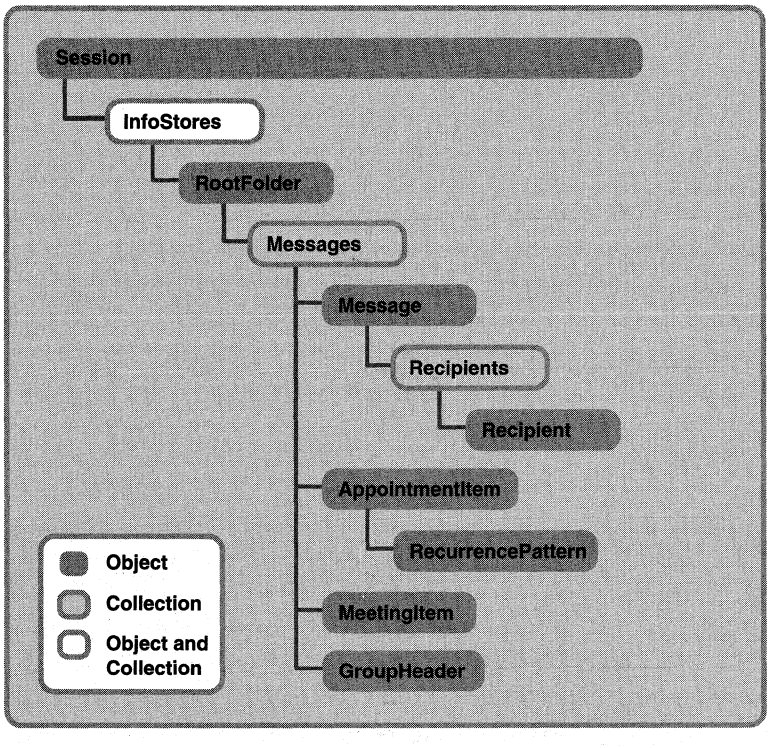


Figure 12-18. *The calendar-related objects of the CDO library.*

A For...Next loop is used to loop through the code twice. The loop starts at the number 3 and continues through the number 4. The loop does not start at the number 1 because the index of the loop is used to create unique <DIV> elements in the HTML code so that the JavaScript *checkExpand* function can find the tables using them. Since this code is fairly generic, you can use it in other applications that render tens or hundreds of blocks of free/busy information for users rather than blocks for only two users. All you need to do is change the ending point of the loop and pass in the targeted users' free/busy information as an array of names.

The code uses the *GetFreeBusy* method of the CDO Recipient object to return the free/busy calendaring information for the current user, which is retrieved using the *CurrentUser* property on the CDO Session object, and the name of the person who submitted the help ticket, which is stored in the Recipient object named objonerecip. The *GetFreeBusy* method in CDO is exactly the same as the *GetFreeBusy* method in the Outlook object library. You need to pass these four parameters to this method:

- The start time of the first slot you want to retrieve.
- The end time of the last slot you want to retrieve.
- The interval, in minutes, of each of the slots.
- A Boolean specifying whether you want to have CDO return full formatting for the slots. Full formatting will inform you of the exact nature of busy slots: *CdoFree (0)*, *CdoTentative (1)*, *CdoBusy (2)*, or *CdoOutOfOffice (3)*. If you do not enable full formatting, CDO will return *0* for free and *1* for busy.

For *GetFreeBusy*, CDO will return the most committed value in the time slots. This means that if a user has overlapping appointments in the slot and one is tentative and the other is out-of-office, CDO will return the out-of-office value. CDO will not return "free" for time slots unless the entire time slot is free. Be careful when working with free/busy information far beyond the current date. By default, free/busy information is published only two months past the current date. If you query beyond the published information for free/busy, CDO will return free slots even though the user might have appointments during those slots. You can ask your Outlook users to change the number of months to a maximum of 12 months past the current date for free/busy information if you need longer term information. They can change this number of months in Outlook by selecting Options from the Tools menu, clicking Calendar Options, and clicking Free/Busy Options.

The return value for this method returns a string of numbers that correspond to the free/busy status of the user for all the time slots you specified. Your code must then parse the string and make a graphical representation of this information in your application.

The Helpdesk application parses this string by first looking at the length to make sure it's not zero, which would mean there is no free/busy information for the user. It then uses a For...Next loop and the VBScript *Mid* function to retrieve each character from the string and display the free/busy status for the user in an HTML table. Notice in the code that the value from the retrieved free/busy status interval is used as an index for the *arrBGColors* array of colors we created earlier: *CdoFree (0)* will display light gray, *CdoTentative (1)* will display light blue, *CdoBusy (2)* will display blue, and *CdoOutOfOffice (3)* will display purple. The For...Next loop ranges from 1 through 48 because we specified 30 minutes as the interval for the time slots in the call to the *GetFreeBusy* method. If you calculate the number of 30-minute intervals from 12 AM to 11:30 PM, the result turns out to be 48.

Creating a Meeting with the User

Once a technician finds time slots in the Calendar Information section of a help ticket that are open for both the technician and the user, the technician can schedule a meeting with the user to troubleshoot on site. The technician can schedule a meeting by first typing the date and the time for the appointment in the text boxes provided at the top of the help ticket, as shown in Figure 12-19.

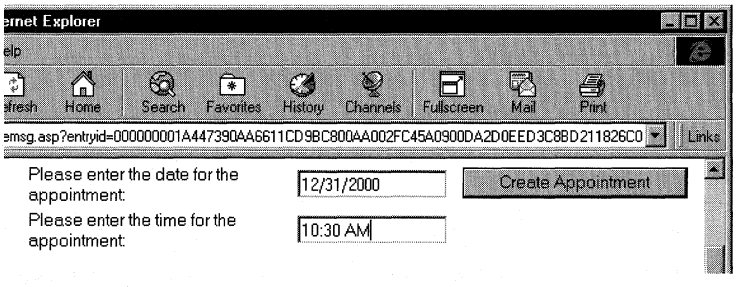


Figure 12-19. *The help ticket with the date and time for an appointment filled in. Technicians can automatically schedule meetings with users to troubleshoot their machines on site.*

When a technician types in a date and a time and then clicks Create Appointment, createcal.asp is called by the application to create the actual meeting request, as shown in this code:

```
<%
'Convert the passed-in date to a vbDate
set querydate = Request.Form("Date")
apptdate = CDate(querydate)
set querytime = Request.Form("Time")
appttime = TimeValue(querytime)
compdatime = apptdate & " " & appttime
compdatime = CDate(compdatime)
```

```
'Add a new message to the user's calendar
Set calfolder = objOMSession.GetDefaultFolder(0)
set CalRequest = calfolder.messages.add

'Set the message properties
CalRequest.Subject = oMessage.Subject
CalRequest.StartTime = compdatime
CalRequest.EndTime = DateAdd("n", 90, CalRequest.StartTime)
CalRequest.Location = myoffice
CalRequest.ResponseRequested = TRUE
set meetingrecip = CalRequest.Recipients.Add
meetingrecip.Name = objUser
meetingrecip.Resolve
CalRequest.ReminderSet = True
CalRequest.ReminderMinutesBeforeStart = 30
CalRequest.MeetingStatus = 1
CalRequest.Send
if err.number = 0 then
   response.write "<SCRIPT LANGUAGE = 'JavaScript'>"
   response.write
      "window.alert('Meeting successfully created!');"
   response.write "</SCRIPT>"
else
   response.write "<SCRIPT LANGUAGE = 'JavaScript'>"
   response.write "window.alert('Error: " & err.number & " " &
      err.description &"');"
   response.write "</SCRIPT>"
end if
%>
```

First the code converts the passed-in date to a Date data type and the passed-in time to a Date. Then it combines the two and converts the results to a Date so that the date can be passed to the CDO property *StartTime* to indicate the start time of the appointment. The code uses the *GetDefaultFolder* method of the CDO Session object to retrieve the calendar folder of the technician. The *GetDefaultFolder* method in CDO is similar to the *GetDefaultFolder* method in the Outlook object library, but be careful when using them because the values for the constants that represent the folders are different in the two libraries.

By using the *Add* method on the Messages collection in the calendar folder, CDO automatically adds a new AppointmentItem object to the collection. The code then sets the properties for this new AppointmentItem object to turn it into a MeetingItem object, which will be sent to the user.

To create a meeting request in CDO, you need to set some specific properties on the AppointmentItem object to turn it into a MeetingItem object. First you need to add some recipients to the inherited Message object by creating a Recipient object

in the Recipients collection. In the Helpdesk, the name of the person who submitted the help ticket is added and resolved against the address book as a recipient for the MeetingItem object.

Then you need to set the *MeetingStatus* property to *CdoMeeting (1)* for the AppointmentItem object you created. By setting this property, the current user, who is the technician, is set as the meeting organizer in the *Organizer* property. The *MeetingStatus* property can take other values as well:

- *CdoNonMeeting (0)* is the default and tells CDO that the appointment being organized is a regular appointment and does not represent a meeting.
- *CdoMeetingCanceled (5)* indicates that the meeting organizer has canceled the meeting.

Although the capability to cancel a meeting is not used in the Helpdesk application, here is some information about it. When canceling a meeting, call the *Send* method again on the object to send the cancellation to all attendees. Also, make sure you set the object that holds the meeting to *Nothing*. If you are the organizer of the canceled meeting and Outlook is the main calendar store for your users, you also need to call the *Delete* method on the Message object that is the parent of the MeetingItem object you just canceled. If you do not do this, you might get unexpected results when working with the folder and its contents in the future.

You should also set the *ResponseRequested* property. This property takes a Boolean that tells CDO whether the meeting organizer wants responses to the meeting request. You should set this property to *False* only if you want to send out an FYI meeting request, which adds the item in a user's calendar, but does not need to track whether the user has accepted or rejected the item. For example, you would set the property to *False* if you sent out meeting requests for all the holidays in a year but did not actually care which holidays your users decided to take off.

The preceding script sets some general properties for the appointment item such as the *Subject*, which is the problem contained in the ticket. It also sets the *StartTime* and *EndTime* properties of the appointment so that the *EndTime* is 90 minutes after the *StartTime*. The *Location* property is set to the office location for the user, which is pulled from the Exchange Server directory. The *ReminderSet* and *ReminderMinutesBeforeStart* properties are also set to *True* and 30 minutes before the appointment starts to make sure both parties are aware that they need to meet.

The final step when you create any meeting request is to call the *Send* method on the MeetingItem object, which sends the request to the recipients you invited to the meeting. If any of the properties you set are incorrect or empty, CDO will return

an error after calling *Send*. For this reason, the code checks the Err object in VBScript. Depending on whether or not an error occurred, the JavaScript client code will display a success message or an error message. Figure 12-20 shows the success message. If a property is empty, such as the *Office* property for an Exchange user, the error message "Error 448: Named argument not found" is displayed.

Figure 12-20. *A JavaScript alert box indicating that the technician successfully created a meeting with the user in the help ticket.*

After the meeting request is sent, the user can use any client to accept or decline the meeting request. You can even send meeting requests to users over the Internet, and as long as they are using Outlook or OWA, they can view and accept or decline meeting requests. You will not, however, be able to view the free/busy information of users on different systems. Figure 12-21 shows the scheduled meeting in the user's Outlook calendar.

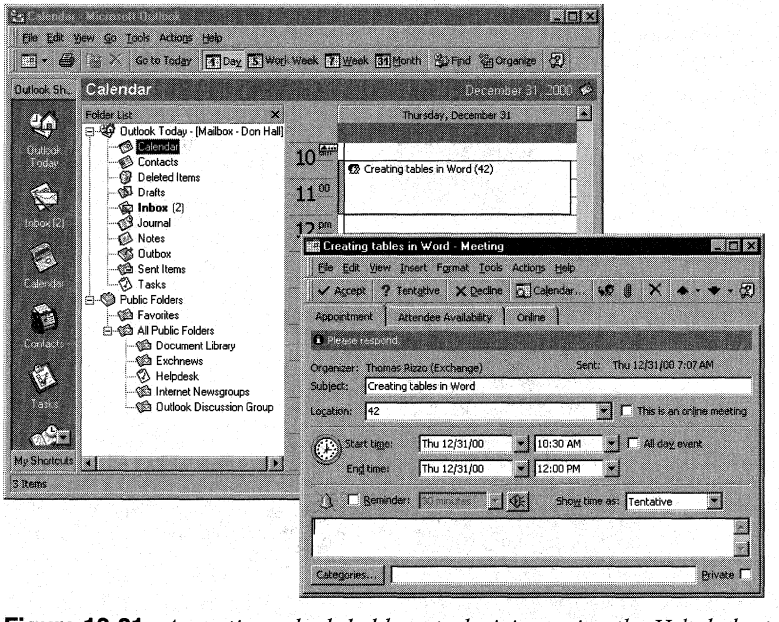


Figure 12-21. *A meeting scheduled by a technician using the Helpdesk application.*

Resolving the Help Ticket

The technician can mark a help ticket as resolved. Resolving a help ticket consists of setting the status of the ticket as *Done* and sending an e-mail message to the user who submitted the ticket explaining that the issue has been resolved. The code that resolves the help ticket is resolved.asp. This code shows you how to update fields on a message and also how to send e-mail messages using the CDO library:

```
<%
if InStr(Request.Form("Action"), "Resol") then
   set objFolder = Session("HelpdeskFolder")
   set recursefolder = objFolder

   set objMessageID = Request.QueryString("entryid")
   set oMessage = objOMSession.GetMessage(objMessageID, _
      Session("InfoStoreID"))
   'Retrieve the message flag and set it
   set msgstatus = oMessage.Fields("Flag")
   msgstatusid = msgstatus.ID
   oMessage.Fields(msgstatusid) = "Done"
   oMessage.Update

   'Send a resolved message to the user stating that the
   'problem was resolved
   Set objNewMsg = objOMSession.outbox.messages.add
   objNewMsg.Text = "Your problem: " & oMessage.Subject & _
      " was resolved on " & Now & chr(10) & chr(10) & _
      "Please see the helpdesk FAQ at " & _
      "http://exserver/faq/ for commonly asked questions."
   objNewMsg.Subject = "Resolved: " & oMessage.Subject
   Set objonerecip = objNewMsg.recipients.add
   objonerecip.Name = oMessage.Fields("From User")
   objonerecip.Resolve
   'Send the message without showing a dialog box
   objNewMsg.Send showDialog=False
end if
%>
```

The script retrieves the help ticket from the folder. Then it uses the *Add* method on the Messages collection to add a new message to the Outbox of the technician. The script sets the message text by using the *Text* property of the Message object. The *Text* property can contain only plain text. It does not support formatted text.

The code then sets the *Subject* of the message and adds recipients to the Recipients collection. The recipient for this message is the user who submitted the ticket. The code uses the *Resolve* method of the Message object to make sure that there are no

ambiguous recipients on the message. Finally, it calls the *Send* method to send the message to the user. Figure 12-22 shows a sample of the e-mail received by the user.

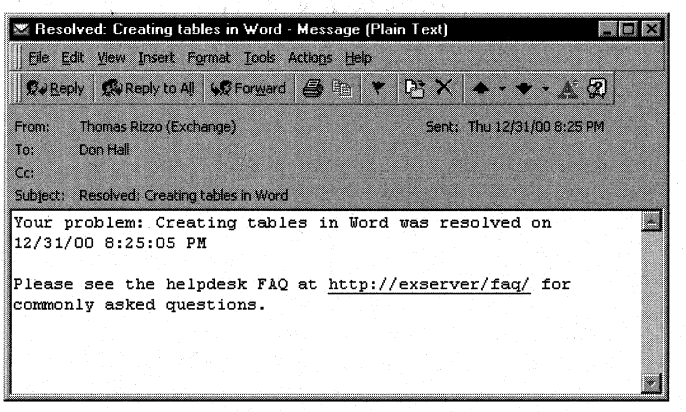


Figure 12-22. *An e-mail message, sent to the user by the technician, indicating the problem has been solved.*

CALENDAR OF EVENTS APPLICATION

While the Helpdesk application showed you a little bit about the calendaring functionality in the CDO library, the Calendar of Events application demonstrates the full power of CDO calendaring. The idea behind the Calendar of Events application is to allow a corporation to publish, either internally or externally, a calendar of corporate events. Since the application is based on Exchange Server and Outlook, it is rich enough to support permissions for the creator or modifier of the calendar content and easy enough for users to add new content by employing familiar tools.

The Calendar of Events application shows how to create publicly accessible calendars, use message filters in the CDO library, and use more objects in the CDO Rendering library. You will also see some of the current limitations in the CDO library and learn about ways to work around these limitations, particularly the limitations of not supporting public folder calendars and not being able to filter by category when using appointment items.

The monthly results page for the application is shown in Figure 12-23. As you can see, the application is Web-based, but users can take advantage of Outlook's calendaring features to create the contents for the Web-based calendar. The application dynamically connects and retrieves the information that Outlook users create in the Exchange Server database and exposes it to Web clients. Outlook users can also use their Outlook client to retrieve calendar information, as shown in Figure 12-24.

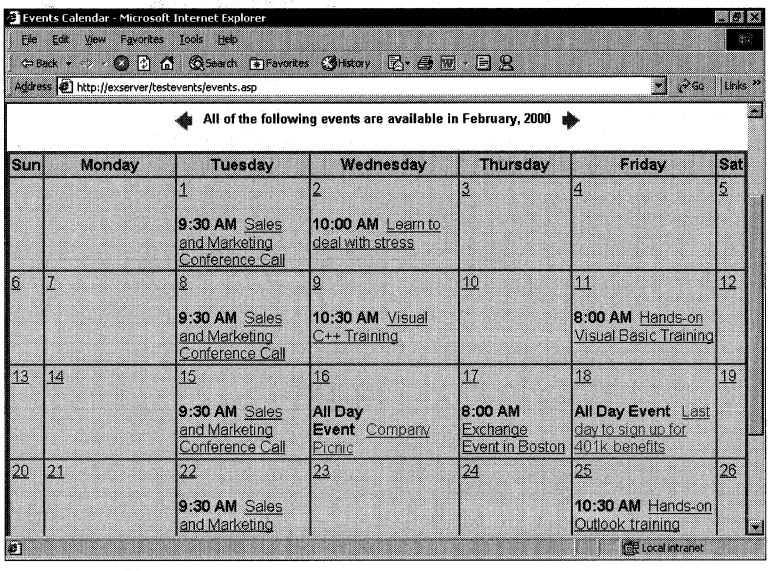


Figure 12-23. *The monthly view of the Calendar of Events in HTML.*

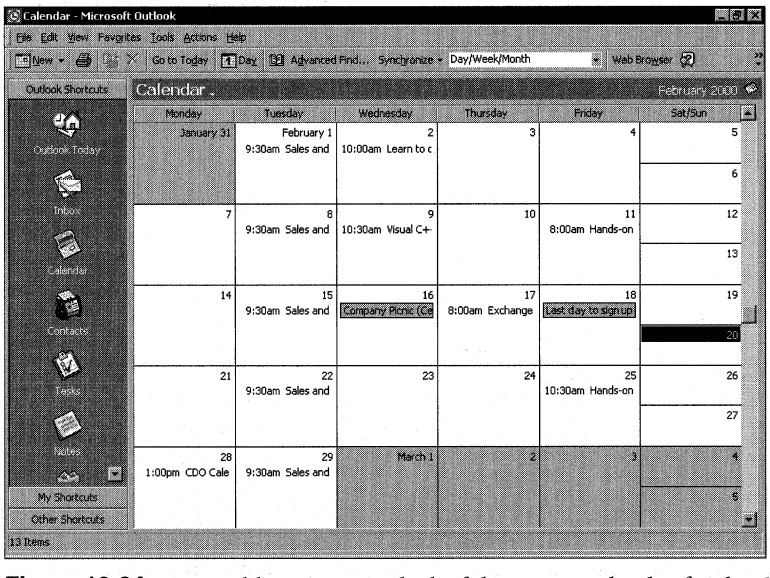


Figure 12-24. *A monthly view in Outlook of the source calendar for the Calendar of Events application.*

Setting Up the Calendar of Events Application

Before you can install the Calendar of Events application, you must have a Windows NT 4.0 Server or a Windows 2000 Server and a client with certain software installed. Table 12-3 outlines installation requirements.

Required Software	*Installation Notes*
Exchange Server 5.5 Service Pack 1 or later with Outlook Web Access	Service Pack 3 is recommended.
IIS 3.0 or higher with Active Server Pages	IIS 4.0 is recommended.
CDO library (cdo.dll) and CDO Rendering library (cdohtml.dll)	Exchange Server 5.5 Service Pack 1 installs CDO library 1.21 and CDO Rendering library 1.21. Outlook 2000 installs CDO library 1.21.
For the client:	
A Web browser or Outlook	You can run the client software on the same machine or on a separate machine.

Table 12-3. *Installation requirements for the Calendar of Events application.*

To install the application, copy the Calendar Of Events folder from the companion CD to the location on your Web server from which you want to run the application.

Start the IIS administration program. Create a virtual directory that points to the location where you copied the Calendar Of Events files, and name the virtual directory *events*. Enable Execute permissions on the virtual directory. This step allows you to use the following URL to access your Events Calendar: *http://yourservername/events*.

Launch the Exchange Administrator program. Select Options from the Tools menu and click the Permissions tab. Make sure that the Show Permissions Page For All Objects check box is checked and the Display Rights For Roles On Permissions Page check box is checked. Create a new mailbox that will contain the Events Calendar by selecting New Mailbox from the File menu. In the Properties dialog box, fill in the information for the mailbox, such as *Events Calendar* for Display and *events* for Alias. Set the Primary Windows NT Account to the IIS anonymous user account (IUSR_*servername*). Click on the Permissions tab, and add yourself as a user account with permissions to this mailbox.

> **NOTE** Your anonymous IIS user account should be a domain account or at least assigned as the owner of a mailbox in Exchange Server. Normally, the account IIS uses to log users on to your Web pages anonymously is named IUSR_*servername*. The Calendar of Events application uses this account when starting an ASP session to automatically log on to the Exchange server without prompting for credentials. You will see how this is used when we step through the application. If the IIS anonymous user account is not a domain account or cannot be assigned as an owner of a mailbox in Exchange Server, change the account so that it can be assigned Owner permissions.

If you use a different alias name for the Events Calendar mailbox, you will need to modify the Global.asa file. Open the Global.asa file in the Calendar Of Events folder on your IIS server. Find the line

```
Application("MailboxName") = "events"
```

and change the name to the alias name of the Events Calendar mailbox you created.

The application includes a file named cats.inc. Since the application allows you to filter events based on Outlook categories, you might want to change cats.inc to reflect the categories that are important to your application. If you do change the categories, you will need to specify the total number of categories you want to filter on. The following code is the sample file from the companion CD:

```
<% NumberofCats = 5
Dim strCategories(5)
strCategories(1) = "Business"
strCategories(2) = "Competitive"
strCategories(3) = "Presentations"
strCategories(4) = "Hands-On Training"
strCategories(5) = "Social"
%>
```

To change the values for your categories, modify the *NumberofCats* integer to be the total number of categories. Then change the Dim strCategories(5) statement to reflect the number of categories, thus enabling VBScript to create an array of the category names. Now type the name of the category as a string argument in one of the cells of the array.

If you specified a name for the virtual root that is different from */events*, find the file named virtroot.inc, open it, and change the virtual root you created for the application.

Create a profile for Microsoft Outlook that connects to the Events Calendar mailbox you created. You can create a new profile by opening the Mail control panel applet and clicking Show Profiles on the Services tab. On the General tab of the displayed Mail dialog box, you can add a new profile. You can also create a new profile by clicking the New button on the Choose Profile dialog box when you start Outlook. If the Choose Profile dialog box does not automatically display when you start Outlook, choose Options from the Tools menu in Outlook and click on the Mail Services tab. In the Startup Settings area, select the Prompt For A Profile To Be Used option.

With Outlook opened to the Events Calendar mailbox, right-click on the Calendar folder, select Properties, and click on the Permissions tab. Set permissions for the users in your organization who need to create and edit appointments in the calendar. You do not have to set the Default permissions on the folder, so you can restrict the access permissions for each user in the organization and enable permissions for creating and deleting items without giving permissions to everyone with access to

the calendar. This step will allow Outlook users with the proper permissions to view and possibly edit the calendar for the Events Calendar mailbox. To open the Calendar folder for the Events Calendar mailbox, other users would choose Open from the File menu in Outlook and then select Other User's Folder. From the displayed Open Other User's Folder dialog box, users can select and open the Calendar folder. You're finished. You can now add events to the Outlook calendar for the Events Calendar mailbox, and then test viewing those events from the URL *http://*yourservername*/events*.

CDO Sessions

The Calendar of Events application uses the Global.asa file of the Helpdesk application with a few changes. The file is modified primarily because CDO does not support accessing calendars in public folders or delegated calendars at the time this book went to press. Just a quick note, though—CDO for Exchange 2000 does support this capability, but not with an Exchange 5.5 Server. It supports it only on an Exchange 2000 Server. You can access calendars only when you are the primary Windows NT account owner of the mailbox. With ASP, you can get around this limitation by assigning the anonymous IIS user account as the primary owner of a mailbox, thereby making all users who browse your Web page automatically log on to CDO using this mailbox as their default.

Because IIS uses the security context of this anonymous user account to browse Web pages anonymously, you do not have to prompt the user for security credentials to enable them to log on to the mailbox, as we had to in the Helpdesk application. Instead, all you need to do is add a CDO logon to the *Session_OnStart* subroutine in your Global.asa. This logon method will force every new Web user to log on to the Exchange server using the mailbox you created as well as the security credentials of the anonymous user account in IIS. The Global.asa code for the Calendar of Events application is shown here:

```
<SCRIPT LANGUAGE="VBScript" RUNAT="Server">

Sub Application_OnStart
   On Error Resume Next
   Set objRenderApp = Server.CreateObject("AMHTML.Application")
   If Err = 0 Then
      Set Application("RenderApplication") = objRenderApp
   Else
      Application("startupFatal") = Err.Number
      Application("startupFatalDescription") = _
         "Failed to create application object<br>" & _
         Err.Description
   End If

   Application("hImp") = Empty
```

(continued)

```
    'Load the configuration information from the registry
    objRenderApp.LoadConfiguration 1, _
       "HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\" & _
       "MSExchangeWeb\Parameters"
    Application("ServerName") = objRenderApp.ConfigParameter("Server")
    Application("MailboxName") = "testacct"
    Err.Clear
End Sub

Sub Application_OnEnd
    Set Application("RenderApplication") = Nothing
End Sub

Sub Session_OnStart
'On Error Resume Next
    Set objRenderApp = Application("RenderApplication")
    hOldImp = objRenderApp.ImpID
    Set Session("AMSession") = Nothing
    set objOMSession = Server.CreateObject("MAPI.Session")
    bstrProfileInfo = Application("ServerName") + vbLF + _
       Application("MailboxName")
    objOMSession.Logon "", "", False, True, 0, True, bstrProfileInfo
    set Session("AMSession") = objOMSession
    'This is a handle to the security context.
    'It will be set to the correct value when the CDO session is created.
    Session("hImp") = objRenderApp.ImpID
End Sub
'While calling the Session_OnEnd event, IIS doesn't call us in
'the right security context.
'Workaround: current security context is stored in Session
'(look at logon.asp) and then gets restored in Session_OnEnd
'event handler.
'
'All CDO and CDOHTML objects stored in the Session object
'need to be explicitly set to Nothing between the two
'objRenderApp.Impersonate calls below.
Sub Session_OnEnd
    On Error Resume Next
    set objRenderApp = Application("RenderApplication")
    hImp = Session("hImp")
    If Not IsEmpty(hImp) Then
       objRenderApp.Impersonate(hImp)
    End If
    'Do our cleanup. Set all CDO and CDOHTML objects inside
    'the session to Nothing.
    'The CDO session is a little special because we need to do
    'the Logoff on it.
    Set objOMSession = Session("AMSession")
    If Not objOMSession Is Nothing Then
```

```
        Set Session("AMSession") = Nothing
        objOMSession.Logoff
        Set objOMSession = Nothing
    End If
End Sub
</SCRIPT>
```

Since all users of the application will be accessing the same mailbox, you might be wondering why the code for logging on to the Exchange server is in the *Session_OnStart* subroutine and not in the *Application_OnStart* subroutine. The main reason for creating a new session for each user to the same mailbox is to improve the performance of the application. If the application did not do this, all users would use the same CDO session to connect to the Exchange server.

Prompting the User for Input

After a CDO session has been created for the user but before the user can start viewing the calendar, the application must ask which appointment types and month the user wants to view in the calendar. To do this, the application presents a search page with options to select the month, year, and event categories, as shown in Figure 12-25.

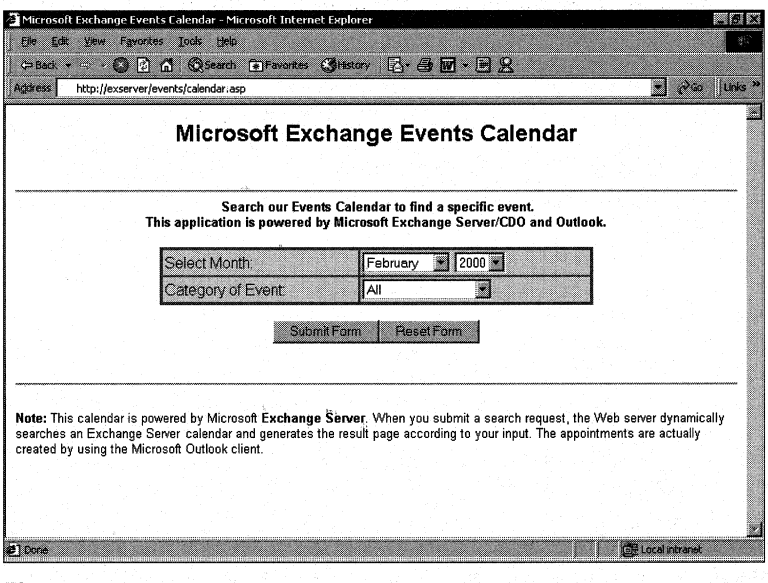


Figure 12-25. *From the search page of the Events Calendar, the user can select the month, year, and event categories to search for.*

The next section of code is for the search page in Figure 12-25. Notice how the code figures out the current month on the Web server machine and uses it as the default value in the Select Month drop-down list box. Using the current year as the point of reference, the code dynamically generates the previous year and next year

in the Year drop-down list box. This page also uses a hidden control on the HTML form that will indicate to the next ASP page, Events.asp, that the user originated from the current page.

The HTML page does not have to check whether a valid ASP session exists for the current user because the page does not use any CDO code. The CDO logon code is handled in the *Session_OnStart* procedure, so when the user's session has timed out, the user is automatically logged on again when he refreshes the screen or moves to a different page. Here is the code for the search page:

```
<!--#include file="cats.inc"-->
<Title>Microsoft Exchange Events Calendar</Title>
<center>
<p><b><FONT FACE="Arial, Helvetica" SIZE=5>
Microsoft Exchange Events Calendar</font></b></P>
<BR>
<HR>
<FONT FACE="Arial, Helvetica" SIZE=2>
<b>Search our Events Calendar to find a specific event. <BR>
This application is powered
by Microsoft Exchange Server/CDO and Outlook.</b>
<FORM METHOD=POST ACTION="events.asp">
<TABLE BORDER=2 Width=60% Bordercolor="008000" cellpadding="2"
cellspacing="0" borderdarkercolor="008000" bgcolor="#FFCC00"
borderlightcolor="008000">

<%
'**********************************
'Figure out the month
'**********************************
%>
   <TR>
   <TD>Select Month:</TD>
   <TD>
   <SELECT NAME="month" SIZE=1>
   <%
   Dim MonthArray(12)
   MonthArray(1)="January"
   MonthArray(2)="February"
   MonthArray(3)="March"
   MonthArray(4)="April"
   MonthArray(5)="May"
   MonthArray(6)="June"
   MonthArray(7)="July"
   MonthArray(8)="August"
   MonthArray(9)="September"
   MonthArray(10)="October"
   MonthArray(11)="November"
   MonthArray(12)="December" %>
```

```
   <% for i = 1 to 12 %>
      <% if month(now) = i then %>
         <OPTION Selected Value = <%=i%>> <%= MonthArray(i) %>
      <% Else %>
         <OPTION Value = <%=i%>> <%= MonthArray(i) %>
      <% End if %>
   <% Next %>
   </SELECT>

<%
'**********************************
'Figure out the year
'**********************************
%>
   <SELECT NAME="year" SIZE=1>

<% 'Figure out the current year, and go back and ahead 1 year %>
<% yearprevious = dateadd("yyyy",-1,date) %>
   <OPTION> <% response.write year(yearprevious) %>
   <OPTION SELECTED> <% response.write year(date) %>
<% yearnext = dateadd("yyyy",1,date) %>
<OPTION> <% response.write year(yearnext) %>
   </SELECT>
   </TD></TR>

<%
'**********************************
'Figure out the categories
'**********************************
%>

   <TR>
   <TD>Category of Event:</TD>
   <TD><SELECT NAME="Type" SIZE=1>
      <OPTION SELECTED>All
      <% for c = 1 to NumberofCats
         response.write "<OPTION>" & strCategories(c)
      next
      %>
   </SELECT>
   </TD></TR>
</TABLE>

<%
'**********************************
'Create a hidden field so that we know the
'request came from calendar.asp
'**********************************
%>
```

(continued)

```
<INPUT TYPE=HIDDEN NAME="fromcalendar" VALUE="fromcalendar"><BR>
<INPUT TYPE=SUBMIT VALUE="Submit Form"><INPUT TYPE=RESET
VALUE="Reset Form">
</FORM>
<br>
<HR>
</center>
</font>
<BR>
<FONT FACE="Arial, Helvetica" SIZE=2>
<P><b>Note:</b> This calendar is powered by Microsoft
<b>Exchange Server</b>.
When you submit a search request, the Web server dynamically
searches an Exchange Server calendar and generates the
result page according to your input. The appointments are
actually created by using the Microsoft Outlook client.
</font>
</BODY>
</HTML>
```

Displaying Views of the Calendar

When the user clicks the Submit Form button on the HTML form, the application passes the entered information to the next ASP page in the application, Events.asp. This page creates a monthly view of the information stored in the Events Calendar, as shown in Figure 12-26.

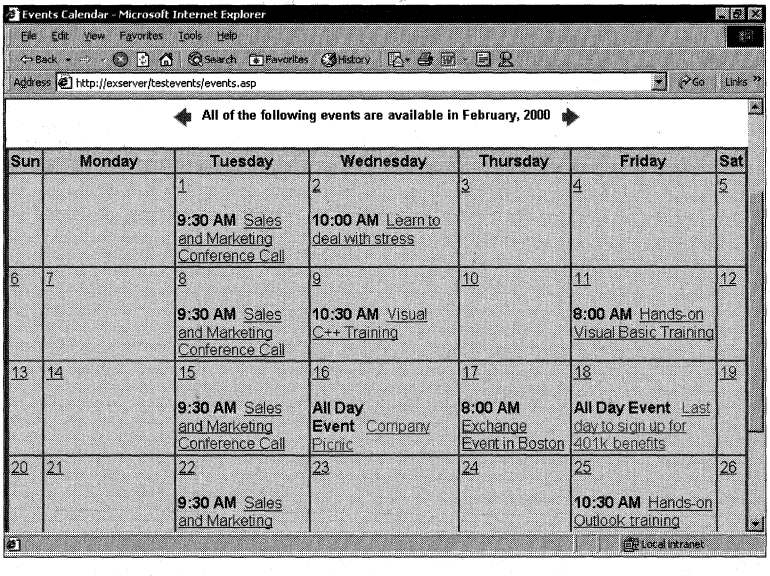


Figure 12-26. *The code for this page is Events.asp. The code creates a monthly view of the appointments in the calendar by using HTML.*

Since the CDO Rendering library does not natively support monthly calendar views, the page in Figure 12-26 creates a monthly view using only HTML tables and data from the Events Calendar. However, the CDO Rendering library does support daily and weekly views on calendars. Therefore, when the user selects to view all events in the calendar, the application renders the calendar day numbers as hyperlinks from which the user can drill down into either daily views of the calendar day or weekly views of the events for the entire calendar week. Weekly views are available only when the user clicks on the hyperlink for Sunday. Daily views are available on all other calendar days. Both view types are generated by the CDO Rendering library.

This page also stores values for the month, day, and year in ASP-session scope variables so that the application can remember the values in other pages. Storing these values also enables the application to create filters on the appointments contained in the calendar folder so that only the appointments for the specified month appear in the calendar. Let's take a look at the application and associated CDO objects in more detail.

Filtering Events from the Calendar

So that only certain events appear in the calendar, the application uses the MessageFilter object in the CDO library. The MessageFilter object is available in any Messages collection and allows you to specify the criteria that messages must meet before they are added to the collection. When you instantiate a new Messages collection, by default a MessageFilter object is created without filters on the content.

The MessageFilter object allows you to filter on built-in and custom properties for message objects in the Messages collection. There is one caveat, however, with the MessageFilter object: if the collection contains AppointmentItem objects (and a calendar folder does), the MessageFilter object offers only a limited subset of its functionality. This subset is the ability to filter only on the start and end times for the items in the collection. For this reason, in the source code for the Events.asp file, you'll notice a MessageFilter object that uses the month selected by the user as the input for the filter's start and end times. You will also notice that custom VBScript code searches through the filtered set of appointments to figure out which appointments actually have the category the user selected. This functionality is implemented as custom VBScript because the MessageFilter object lacks this functionality for appointments. The following code creates and sets the MessageFilter object for the events calendar:

```
<%
'*****************************************************************
'Filter all appointments except the requested month's appointments
'*****************************************************************
'Get the Calendar folder
Set Session("objFolder") = objOMSession.GetDefaultFolder(0)
set objFolder = Session("objFolder")
```

(continued)

```
set objAppointment = objFolder.messages.getfirst()
set objAppointments = objFolder.Messages
set objMsgFilt = objAppointments.Filter

'Calculate the start and end dates based on the month the
'user selected
StartDate = EventMonth & "/" & "1" & "/" & EventYear
EndDate = EventMonth & "/" & "1" & "/" & EventYear
EndDate = DateAdd("m",1,EndDate)
objMsgFilt.Fields(ActMsgPR_START_DATE) = EndDate
objMsgFilt.Fields(ActMsgPR_END_DATE) = StartDate

set Session("objAppointments") = objAppointments
Session("LastDayofMonth") = iLastDay
%>
```

As the preceding code illustrates, the first step in creating a filter is retrieving the Messages collection you want to apply the filter to. Since the MessageFilter object is a child of the Messages collection, you need to retrieve it by using the *Filter* property on the Messages collection. If the collection you were filtering did not contain appointments, you could create your filter by setting the properties on the MessageFilter object.

Because we are retrieving the calendar folder for the events calendar mailbox, we need to specify properties for the start and end times by using the Fields collection of the MessageFilter object. The specific identifiers for these two properties are *&H00600040* for the start date and *&H00610040* for the end date. (These identifiers are defined in the file Amprops.inc, which is included with the Calendar of Events files on the companion CD.) To create the filter, all you need to do is set these identifiers, in the Fields collection, to your values.

Be careful when setting these properties—they don't work in the way you think they should. For example, you would think that when you specified a value for the start date, you would enter in the first day for the filter, which would make CDO return every appointment starting from the day you entered and moving forward in time. However, the way the code is implemented in the library, the MessageFilter object actually returns any appointments that start on that day or occurred *before* that date. For the end date, the filter returns any appointments that end on the date or occur *after* that date. Therefore, in the Calendar of Events application, the first day of the month selected by the user is specified as the start date value for the filter, and the first day of the next month after the month selected by the user is specified for the end date value. These values return all appointments in the specified month.

Now that we have all the appointments in the month, we need to manually filter them by the category the user specified. For example, if the user specified only hands-on training events, we must provide a subroutine to filter and print only hands-on training events. The next snippet of code does this for you. It uses the For Each...

Next statement in VBScript to scroll through the filtered Messages collection we created. While the code loops through the collection, it checks to see whether the current appointment starts on the current day. If the appointment does start on the current day, the code checks to see whether the user selected a specific category. If the user did select a category, the code loops through the categories on the AppointmentItem object, checking to see whether the object contains the specified category. If the category is found, the application prints out the appointment. If the category is not found, the code moves to the next appointment in the collection.

You might have noticed a variable in the code named *AlreadyPrinted*. I added this variable to help you enable the application to support users who specify multiple categories to search on. Imagine that you have an event that is marked for the Business and Hands-On Training categories. If you allow users to specify both search categories such that any event categorized as either Hands-On Training or Business is identified, you will run into problems with duplicate printing of events because the values for appointment categories added in Outlook are not guaranteed to be in a particular order. The Categories field for one appointment could have the values *Business, Hands-On Training, Competitive* while another could contain the same values but in a different order: *Hands-On Training, Competitive, Business*. When this is the case, both events print.

The code uses a For Each...Next loop to scroll through the categories collection. After it finds the targeted category value and prints the item on the calendar, the code changes the *AlreadyPrinted* variable to True. Therefore, if this item meets other subsequent categories the user selected, it won't be duplicated on the calendar. Why does the code use a variable rather than contain an Exit For statement? I used a variable because it gives you more flexibility if you want to change the code to perform other functionality. However, an Exit For statement would work in this case just as well. Here's the code that filters the appointment categories:

```
<%
'*****************************************************
'Check to see whether event should be written
'*****************************************************
%>
<%

AlreadyPrinted = FALSE
   for each objappointment in objAppointments
      StartTime = objappointment.StartTime
      'Check the day of the message
      oDay = DAY(StartTime)
      'Figure out friendly start time
      if Hour(StartTime) = 12 then  '12:00 PM
         if Minute(StartTime) = 0 then '0 minutes
```

(continued)

```
            dStartTime = "12:00 PM"
         else
            dStartTime = "12:" & Minute(StartTime) & " PM"
         end if
      elseif Hour(StartTime) > 12 then
         if Hour(StartTime) > 11 then  'PM
            if Minute(StartTime) = 0 then '0 minutes
               dStartTime = (Hour(StartTime)-12) & ":00 PM"
            else
               dStartTime = (Hour(StartTime)-12) & ":" & _
                  Minute(StartTime) & " PM"
            end if
         end if
      else
         if Hour(StartTime) = 0 then '12 AM
            if Minute(StartTime) = 0 then '0 minutes
               dStartTime = "12:00 AM"
            else
               dStartTime = "12:" & Minute(StartTime) & _
                  " AM"
            end if
         else
            if Minute(StartTime) = 0 then '0 minutes
               dStartTime = Hour(StartTime) & ":00 AM"
            else
               dStartTime = Hour(StartTime) & ":" & _
                  Minute(StartTime) & " AM"
            end if
         end if
      end if 'Friendly start time

      if oDay = (i-iDayMarker) then
         'Check the categories if AllBit = 0
         if AllBit = 1 then
            'Check to see if all-day event
            if objappointment.AllDayEvent = True then
               response.write "<B>All Day Event" & _
               "   </B><A HREF='details.asp?id=" & _
               objappointment.id & _
               "' style='color: rgb(255,0,0)'>" & _
               objappointment.Subject & "</a><BR>"
            else
               response.write "<B>" & dStartTime & _
               "</B>  <A HREF='details.asp?id=" & _
               objappointment.id & _
               "' style='color: rgb(0,0,255)'>" & _
               objappointment.Subject & "</a><BR>"
            end if
```

```
            else
               'Check categories!

               if IsEmpty(objappointment.Categories) then
               'No categories
               else
                  for each category in objappointment.Categories
                     if InStr(category,EventType) then
                        if Not(AlreadyPrinted) then
                           if objAppointment.AllDayEvent = True then
                              response.write _
                              "<B>All Day Event" & _
                              "   " & _
                              dStartTime & "</B>  " & _
                              <A HREF='details.asp?id=" & _
                              objappointment.id & "' style" & _
                              "='color: rgb(255,0,0)'>" & _
                              objappointment.Subject & _
                              "</a><BR>"
                           else
                              response.write "<B>" & _
                              dStartTime & "</B>  <A " & _
                              HREF='details.asp?id=" & _
                              objappointment.id & "' style" & _
                              "='color: rgb(0,0,255)'>" & _
                              objappointment.Subject & _
                              "</a><BR>"
                           end if
                           AlreadyPrinted = TRUE
                        end if 'Not Already Printed
                     end if 'Categories Match
                  next
               end if 'Check categories
            end if 'All Bit
         end if 'oDay
      AlreadyPrinted = FALSE 'Reset Already Printed
      Next

set objappointment = nothing
n=1
%>
```

Displaying a Weekly View

When the user is not filtering by category and clicks on the hyperlink for any Sunday in the calendar, a weekly view appears showing events for the current week, as illustrated in Figure 12-27.

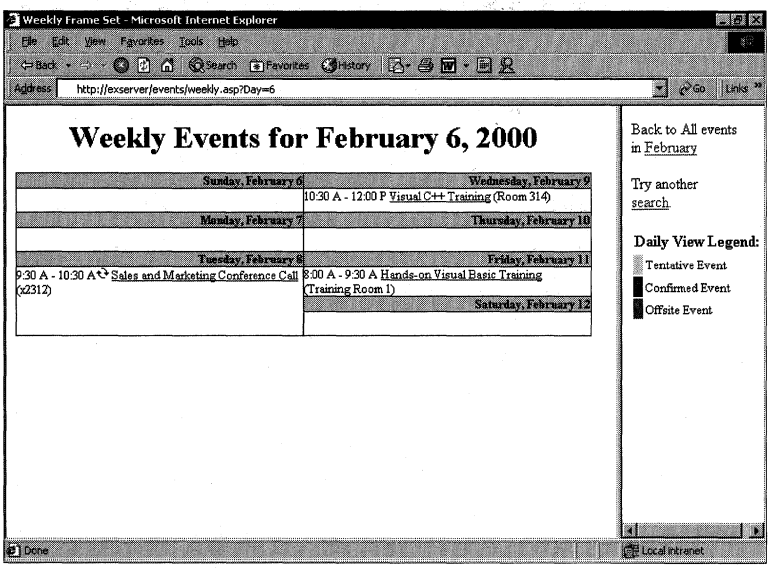


Figure 12-27. *The weekly view in the Events Calendar. This view is reached by clicking on the hyperlink for any Sunday in the calendar.*

The weekly view is implemented in the events calendar by using the CDO Rendering library. While you could create your own weekly view, it is much easier to leverage the CDO Rendering library and customize the way it renders the view using the library's objects. The CDO Rendering library offers rich object support for customizing what is rendered by the library.

As illustrated in the Helpdesk application, the way to get started with the CDO Rendering library in an application is to create either a container or an object renderer by using the *CreateRenderer* method on the RenderingApplication object. The Calendar of Events application creates a container renderer because the items rendered by the application are contained in a calendar folder. However, unlike the Helpdesk application, which used TableView objects to render its data, the Calendar of Events application uses CalendarView objects. The Calendar of Events application also customizes the patterns and formats of the CalendarView object to specify the graphics to be used when rendering information. The placement of the CalendarView object in the CDO Rendering library is shown in Figure 12-28. Most of the properties of the CalendarView object are filled in by default when you instantiate a CalendarView object, so you don't have to set these properties unless you want to customize the way CDO renders the information into HTML.

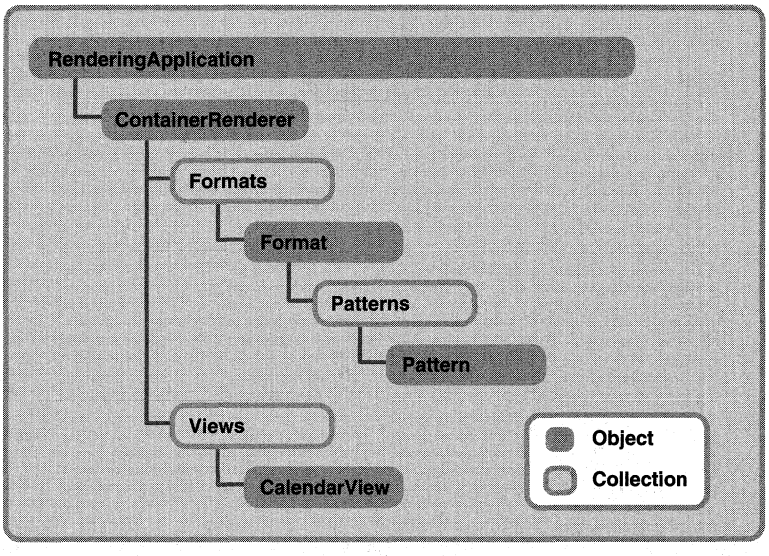


Figure 12-28. *The CDO CalendarView object is a child object of the Views collection in the CDO Rendering library.*

As with the Helpdesk application, we need to set a data source to be rendered. In this case, the data source is the filtered set of appointments we created for the calendar. To instantiate a CalendarView object, we retrieve from our data source a view from the Views collection. Since the daily view always has an index of 1 in the Views collection, the code grabs the daily view for the calendar using this index, instead of scrolling through all the views in the collection. As you will see later, the daily view is morphed into a weekly view by using the *Mode* property for the CalendarView object.

Once we have a CalendarView object, we can manipulate the Format and Pattern objects of the ContainerRenderer object to add custom HTML rendering. The Format object controls how a particular property is rendered by the CDO Rendering library. For example, you can pass to the Formats collection either the ID of a built-in property or the name of a custom property to create a custom HTML format for that property when the property is rendered by the library. This code shows an example from the weeklyview.asp file:

```
'Sensitivity
Set objFormat = oCalContRenderer.Formats.Add( _
   ActMsgPR_SENSITIVITY, Null)
```

After adding the format, you can retrieve the Patterns collection on the Format object to specify how a particular value for a property should be formatted. In the previous example, if the sensitivity of the appointment in the calendar is set to *Private*, an image of a key is placed before the text of the appointment. You can make the patterns more complex because the values for the patterns will accept any legal HTML tags.

You can specify a default pattern for the Pattern objects if a particular property does not contain one of your values. To the Patterns collection, just add a Pattern object that takes the asterisk character (*) as its value. You then specify the HTML tags CDO should use to render the unspecified value types. The code for both specified and unspecified values is shown here:

```
Set objPatterns = objFormat.Patterns
bstrHTML = bstrImgSrc + _
   "/images/private.gif WIDTH=13 HEIGHT=13 BORDER=0>"
objPatterns.Add 1, bstrHTML    ' personal
objPatterns.Add 2, bstrHTML    ' private
objPatterns.Add 3, bstrHTML    ' confidential
objPatterns.Add "*", ""        ' normal
```

The following code shows you the other Format and Pattern object settings for the weekly view in the Calendar of Events application:

```
'Recurring
Set objFormat = oCalContRenderer.Formats.Add( _
   AmPidTag_IsRecurring, Null)
Set objPatterns = objFormat.Patterns
objPatterns.Add 0, ""
bstrHTML = bstrImgSrc + _
   "/images/recur.gif WIDTH=13 HEIGHT=13 BORDER=0>"
objPatterns.Add "*", bstrHTML

'Meeting status
Set objFormat = oCalContRenderer.Formats.Add( _
   AmPidTag_ApptStateFlags, Null)
Set objPatterns = objFormat.Patterns
objPatterns.Add 0, ""
bstrHTML = bstrImgSrc + _
   "/images/meeting.gif WIDTH=12 HEIGHT=13 BORDER=0>"
objPatterns.Add "*", bstrHTML

'Location
Set objFormat = oCalContRenderer.Formats.Add( )
   AmPidTag_Location, Null)
Set objPatterns = objFormat.Patterns
objPatterns.Add "",  ""
objPatterns.Add "*", "(%value%)"
```

After you set your Format and Pattern objects, you can customize the way CDO renders the HTML tables it creates. The properties you need to manipulate the ContainerRenderer object are *TablePrefix*, *TableSuffix*, *RowPrefix*, *RowSuffix*, *CellPattern*, and *LinkPattern*. The following code is taken from weeklyview.asp, which sets these properties:

```
oCalContRenderer.TablePrefix = _
   "<table columns=%columns% border=0 cellpadding=0 cellspacing=1 " _
   & "WIDTH=100% HEIGHT=10% bgcolor='#000000'>" & Chr(10)
oCalContRenderer.TableSuffix = "</table>" & Chr(10)
oCalContRenderer.RowPrefix = "<tr>" & Chr(10)
oCalContRenderer.RowSuffix = "</tr>" & Chr(10)
oCalContRenderer.CellPattern = "<font size=2>%value%</font>"
oCalContRenderer.LinkPattern = "<a href='details.asp?id=%obj%' " _
   & "target='_top'>%value%</a>"
```

The *TablePrefix* property allows you to customize the HTML table that CDO renders before CDO creates a separate table for the actual item content in the data source. By customizing *TablePrefix*, you can add custom HTML tags before CDO renders any content to the browser. This property works in conjunction with the *TableSuffix* property, which specifies what to render at the end of the HTML table created by the *TablePrefix* property.

The *RowPrefix* property allows you to customize the HTML that appears at the beginning of each HTML table row. You can use this property to change the way the row is rendered—for example, modifying the height, width, or alignment of the items in the row. *RowSuffix* specifies the HTML that should appear after the row and is used in conjunction with *RowPrefix*.

The *CellPattern* property specifies the HTML for every cell in each table row that you render. In the code for the Calendar of Events application, the *CellPattern* property is set to a font size of 2 and is set to display the value contained in the appointment. This property does not affect any hyperlinked values in your cell, and CDO always generates a link for exactly one cell in each row. So use *CellPattern* in conjunction with *LinkPattern* to create fully functional table rows, because the *LinkPattern* property affects only the hyperlink cell in your table. As you can see in the code, the application sets the *LinkPattern* property for the hyperlinked cells so that the hyperlink points at the details.asp file, and it passes the EntryID that corresponds to the current appointment clicked on by the user to this ASP using the %obj% token. It also dynamically prints out the subject of the appointment by using the %value% token.

The final section of the code sets some options on the ContainerRenderer object, such as the start and end times for the business day, and the time zone for the appointment dates and times. This section also morphs the daily view into a weekly view by setting the *Mode* property for the CalendarView object to be *CdoModeCalendarWeekly (1)* rather than *CdoModeCalendarDaily (0)*. The code then

calls the *RenderAppointments* method on the CalendarView object, which takes as its arguments the starting date for rendering information and the output stream used to send the generated HTML. Normally for the output stream parameter, you would type *Response*, which tells CDO to render the HTML to the Response object of the ASP object library. The following code implements this functionality for the Calendar of Events application:

```
oCalContRenderer.TimeZone = objOMSession.GetOption("TimeZone")
'Set Sunday as first day of week
oCalContRenderer.FirstDayOfWeek = 7
oCalContRenderer.Is24HourClock = _
   objOMSession.GetOption("Is24HourClock")
oCalContRenderer.BusinessDayStartTime = _
   objOMSession.GetOption("BusinessDayStartTime")
oCalContRenderer.BusinessDayEndTime = _
   objOMSession.GetOption("BusinessDayEndTime")
oCalContRenderer.BusinessDays = _
   objOMSession.GetOption("WorkingDays")
oCalView.NumberOfUnits = 1
curDay = CDate(curDay)
oCalView.Mode = 1
oCalView.RenderAppointments curDate,Response
```

Displaying a Daily View

When the user is not filtering by category and clicks on the hyperlink for any day in the calendar week except Sunday, a daily view appears, as shown in Figure 12-29.

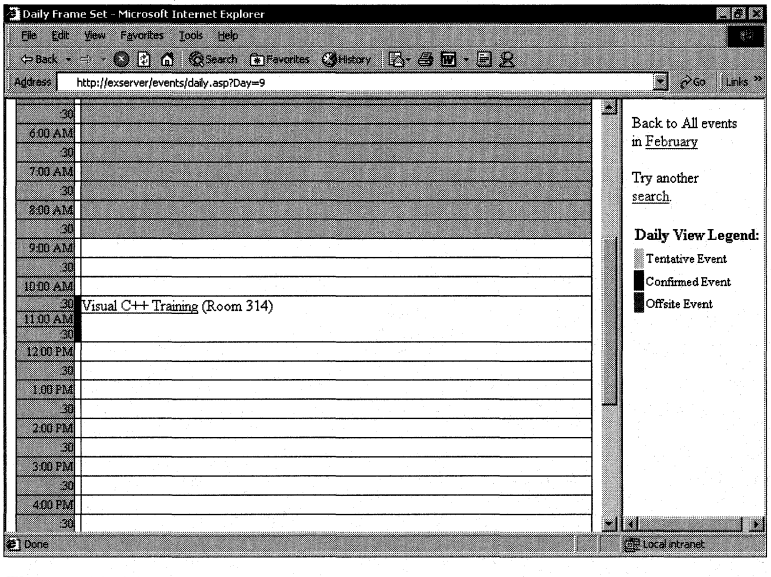


Figure 12-29. *The daily view for the Calendar of Events application allows users to see more details about the events on a specific day.*

The code for rendering the daily view in the file dailyview.asp is similar to the code for rendering a weekly view except for two differences. First, we keep the daily view as the view in the *Mode* property for the CalendarView object rather than change it, as we did in the weekly view rendering code. Second, we must explicitly render all-day events separately from appointments in the view. In the weekly view mode, CDO automatically renders all-day events.

The way to render events separately from appointments in a daily view is to explicitly call the *RenderEvents* method on the CalendarView object before calling the *RenderAppointments* method. The *RenderEvents* method takes the same parameters as the *RenderAppointments* method, which includes the date for which you want to render the events and the output stream that will place the HTML code created by the method. The following code shows you how to render both events and appointments using *RenderEvents* and *RenderAppointments*:

```
oCalView.RenderEvents curDate,Response
oCalView.RenderAppointments curDate,Response
```

Displaying the Details of an Event

When a user clicks on any hyperlink (from monthly, weekly, or daily view) to get the details for an event, details.asp is called. This ASP page displays details about the event so that a user can find the event location and obtain any supporting materials. The ASP page also automatically supports rendering and viewing attachments because it uses the CDO Rendering library to display the text describing the event. The user interface for this page is shown in Figure 12-30.

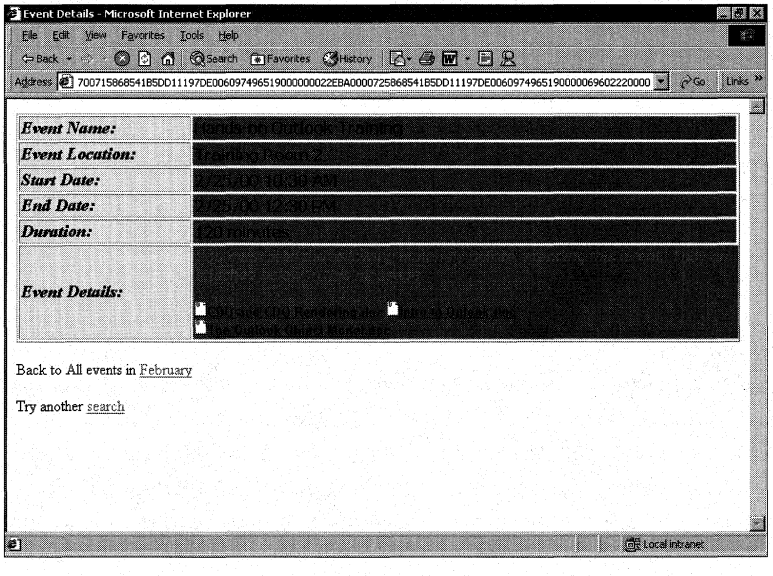


Figure 12-30. *The Details page for an event in the Calendar of Events application can render rich text as well as hyperlinks because it uses the CDO Rendering library.*

The code in the details.asp page is pretty straightforward, but it shows you how to use other objects in the CDO Rendering library, such as the ObjectRenderer object. The ObjectRenderer object (as opposed to the ContainerRenderer object) is used because you are displaying properties from an individual CDO object, such as an appointment. You should use the ContainerRenderer object only if you are rendering a collection of items, such as all the messages in your Inbox. Figure 12-31 shows the ObjectRenderer object hierarchy.

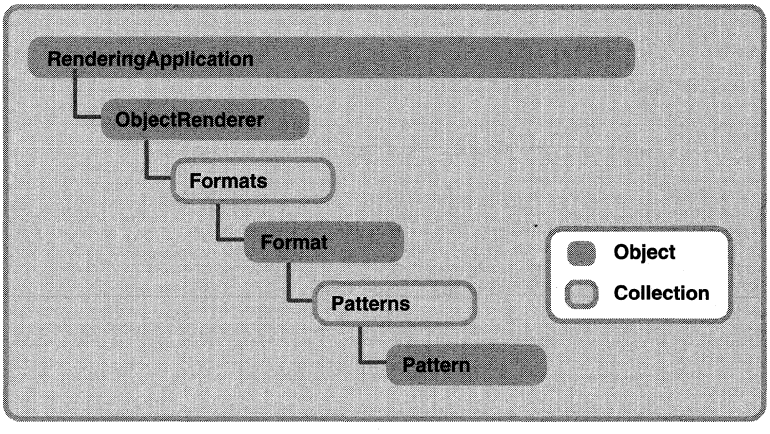


Figure 12-31. *The ObjectRenderer object in the CDO Rendering library is used to display properties of individual items rather than of collections.*

To create an ObjectRenderer object, all we need to do is pass the constant *CdoClassObjectRenderer (2)* to the *CreaterRenderer* method of the RenderingApplication object. (There are a lot of "renders" in that last sentence!) Here is the code from details.asp:

```
'Create an ObjectRenderer
set objObjRenderer = objRenderApp.CreateRenderer(2)
```

After creating the ObjectRenderer, we need to set the *DataSource* property for it. This property is the same as the *DataSource* property for a ContainerRenderer object except for one fundamental difference: the ObjectRenderer object can take only individual items as its data source, such as an AddressEntry, an AppointmentItem, or a Message object.

Now that the data source is set, we can start using some of the methods on the ObjectRenderer object. To render the details of the event, we need to render individual properties off the AppointmentItem object, such as name, location, and details. The ObjectRenderer object gives us a method, *RenderProperty*, that allows us to render individual properties off the object. The *RenderProperty* method takes three arguments:

- The property ID for built-in properties, or the name of the property if it is a custom property that we want to render
- A reserved argument for which you should always pass *0* as the value
- The output stream, to pass the HTML that the CDO Rendering library generates

Normally, you would type *Response* for this argument to return the HTML to the browser. The following code is taken from details.asp. It shows how *RenderProperty* renders the different properties on an AppointmentItem object.

```
<table border="1" width="100%">
<tr>
   <td width="24%" bgcolor="#FFFF80"><big><em><strong>
   Event Name:</strong></em></big></td>
   <td width="76%" bgcolor="#8000FF"><strong><font face="Tahoma">
   <% objObjRenderer.RenderProperty ActMsgPR_SUBJECT, 0, Response %>
   </font></strong> </td>
</tr>
<tr>
   <td width="24%" bgcolor="#FFFF80"><big><em><strong>
   Event Location:</strong></em></big></td>
   <td width="76%" bgcolor="#8000FF"><strong><font face="Tahoma">
   <%
   if objEvent.Location = "" then
      response.write "None specified"
   else
      objObjRenderer.RenderProperty AmPidTag_Location, 0, _
      Response
   end if
   %>
   </font></strong></td>
</tr>
<tr>
   <td width="24%" bgcolor="#FFFF80"><big><em><strong>
   Start Date:</strong></em></big></td>
   <td width="76%" bgcolor="#8000FF"><strong><font face="Tahoma">
   <% objObjRenderer.RenderProperty AmPidTag_ApptStartWhole, 0, _
   Response %>
   </font></strong></td>
</tr>
<tr>
   <td width="24%" bgcolor="#FFFF80"><big><em><strong>
   End Date:</strong></em></big></td>
   <td width="76%" bgcolor="#8000FF"><strong><font face="Tahoma">
```

(continued)

```
   <% objObjRenderer.RenderProperty AmPidTag_ApptEndWhole, 0, _
   Response %>
   </font></strong></td>
</tr>
<tr>
   <td width="24%" bgcolor="#FFFF80"><big><em>
   <strong>Duration:</strong></em></big></td>
   <td width="76%" bgcolor="#8000FF"><strong><font face="Tahoma">
   <% objObjRenderer.RenderProperty AmPidTag_ApptDuration, 0, _
   Response %>
   minutes</font></strong></td>
</tr>

<tr>
   <td width="24%" bgcolor="#FFFF80"><big><em><strong>
   Event Details:</strong></em></big></td>
   <td width="76%" bgcolor="#8000FF"><strong><font face="Tahoma">
   <%
   if objEvent.Text = "" then
      response.write "None specified"
   else
      objObjRenderer.RenderProperty ActMsgPR_RTF_COMPRESSED, _
      0, Response
   end if
   %>
   </font></strong></td>
</tr>
</table>
```

Most of the properties are pretty straightforward, but one of them requires careful handling when rendering because it is quite powerful and can easily cause problems if you do not handle the output correctly. This property is the last one rendered by the application, *ActMsgPR_RTF_COMPRESSED*. It is the message body for the item. When you use the *RenderProperty* method with this property, the CDO Rendering library will automatically convert the rich-text formatting in the message to HTML. This is a powerful feature and one of the primary reasons you should use the CDO Rendering library to display the body of an item.

However, one aspect of this method to watch out for is that in addition to converting the body of an item, the method also converts the attachments in the item to hyperlinks. While this is useful and makes your application very powerful, CDO defaults the hyperlinks it creates to retrieve a file named read.asp in the Exchange virtual root on the IIS server. Remember how IIS defines ASP applications—by virtual root. Now can you see the inherent problem in this? When the user clicks on this default hyperlink, IIS starts a new ASP application under the Exchange virtual root. This causes the Outlook Web Access logon screen to appear, since ASP applications cannot share session and application states, and OWA has no idea that the

user has already authenticated with the Calendar of Events application. Furthermore, if the user enters her security credentials, the attachment will not appear in the browser; instead, OWA will open the Inbox of the user—not the desired functionality.

To fix this problem, we must first change the default virtual root of RenderingApplication, which is the Exchange virtual root. To do this, we use the *VirtualRoot* property on the RenderingApplication. The *VirtualRoot* property takes a string argument that sets the beginning of the URL when you render items. In this case, we need to change the *VirtualRoot* property to point to the virtual root we set in our virtroot.inc file. The following code does this:

```
'Change the virtual root for the rendering application
Set objRenderApp = Application("RenderApplication")
objRenderApp.VirtualRoot = virtroot
```

The next problem we have to resolve is what file to use for the read.asp file that RenderingApplication is creating a hyperlink to. Well, OWA happens to provide a read.asp file that renders attachments you click on in its browser window. The OWA read.asp will automatically download and open the attachment on the machine of the user. In addition, the read.asp file will launch the application in place in the browser if the user's browser supports this option. Otherwise, the read.asp file will prompt the user to download and view the file.

To add the OWA read.asp to the Calendar of Events application, some code has to be modified. The main code modification has to be done to session and local variable names, because the default read.asp for OWA uses variable names different from those in the Calendar of Events application. The code that checks for a valid session also has to be modified since the two applications use different types of session-checking code.

The way attachments are rendered to the browser is the same for both applications. First, the application parses the query string, which contains an *att* variable. This variable contains the attachment record key, which is a unique identifier used to retrieve, from the Attachments collection of the Message object, the particular attachment the user clicked on. Once this attachment is retrieved, the application figures out the attachment's filename so that when the user chooses to save the file after bringing it up, the browser uses the same filename as the filename of the original item. Finally, the application adds a header to the Response object, which tells the browser that it is going to send data to the browser. Then the application uses the *RenderProperty* method of ObjectRenderer to stream the binary data of the attachment to the browser. Once the data is streamed down, the attachment opens. The following code implements this functionality:

```
szAttach = Request.QueryString("att")
nPos = InStr(1, szAttach, "-", 0)
nPos = InStr(nPos+1, szAttach, "-", 0)
```

(continued)

```
nPos2 = InStr(nPos+1, szAttach, "-", 0)
If nPos2 = 0 Then
   nPos2 = Len(szAttach)+1
End If

szRecordKey = Mid(szAttach, nPos+1, nPos2-(nPos+1))
szAttachName = Mid(szAttach, nPos2+1)
szObj = Request.QueryString("obj")
Set objOneMsg = Session("szObj")
If objOneMsg Is Nothing then
   Set objOneMsg = OpenMessage(szObj)
   If objOneMsg is Nothing then
      ReportError1 L_errCannotGetMessageObj_ErrorMessage
   ElseIf objOneMsg.ID = "" then
      ReportError1 L_errMessageDeleted_ErrorMessage
   End If
End If

Set objAttach = objOneMsg.Attachments.Item(szRecordKey)
if objAttach is Nothing then
   ReportError1 L_errFailGettingAttachment_ErrorMessage
End If
bstrFileName = ""
bstrFileName=  objAttach.Fields(ActMsgPR_ATTACH_LONG_FILENAME)
If bstrFileName = "" then
   bstrFileName =  objAttach.Fields(ActMsgPR_ATTACH_FILENAME)
End If
'For short filename compatibility, add these lines
'  If bstrFileName = "" then
'     bstrFileName = objAttach.Name
'  End If

Response.Addheader "Content-Disposition", "attachment;filename=" + _
   bstrFileName
Set objRenderAtt = GetMessageRenderer
objRenderAtt.DataSource = objAttach
objRenderAtt.RenderProperty ActMsgPR_ATTACH_DATA_BIN, 0,Response
```

INTRANET NEWS APPLICATION

The default pages of many intranets today contain corporate news and information. Most of the time, however, the news and information is either manually entered by a Web designer or pulled from some type of database that might not easily support the rendering of attachments for news and general information items. Furthermore, frequently a user must e-mail the Web designer to add new content to the news and information page.

Using Exchange Server Public Folders facilitates the e-mailing of new content for posting to a Web page. By using Public Folders, users can e-mail news items with attachments as well as text using any standard mail client. Because Exchange Server supports auto-expiring of items in Public Folders, users can set how long the item should remain on the news site, which saves the Web designer time and effort. If the user e-mails the news item using Outlook, the user can set the message category type and thus control how the news item will be identified, or categorized, in the Intranet News application. This solution doesn't require any user training because everyone knows how to send an e-mail message!

Using Exchange Server Public Folders also simplifies administrative tasks for the Web designer. The designer can easily set up a group of moderators for the content by using the moderation features built into Exchange Server Public Folders. This allows items to be screened and approved before they are published. (See Chapter 3 for more information about moderating Public Folders.) Figure 12-32 shows the Intranet News application in Internet Explorer. Note that this application is written specifically for the Internet Explorer 4.0 or higher browser and the marquee feature it contains. You can, however, add code to detect the browser you use in your organization and employ the correct display mechanisms for that browser type.

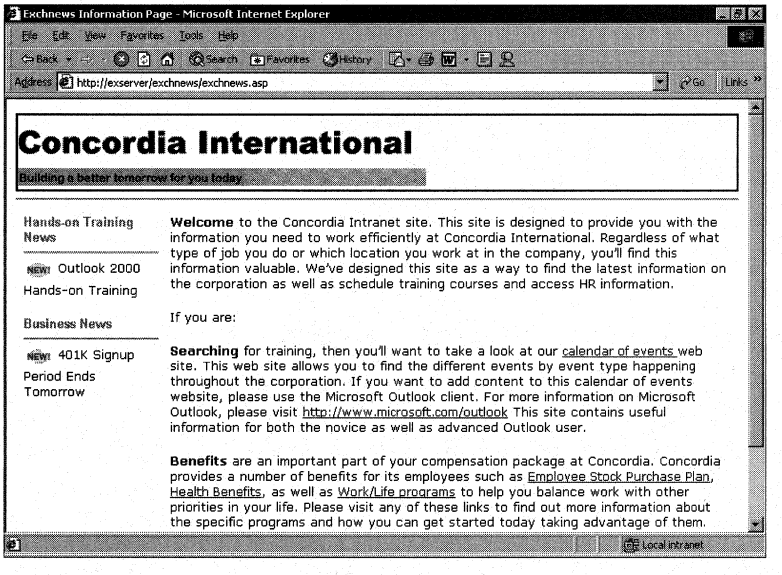


Figure 12-32. *The Intranet News application in Internet Explorer. The application scrolls news and information on a corporation's intranet site and pulls the information dynamically from an Exchange Server Public Folder.*

Setting Up the Application

Before you can install the application, you must have a Windows NT 4.0 Server or Windows 2000 Server and a client with certain software installed. Table 12-4 describes the installation requirements.

Required Software	*Installation Notes*
Exchange Server 5.5 Service Pack 1 with Outlook Web Access	Service Pack 3 is recommended.
IIS 3.0 or higher with Active Server Pages	IIS 4.0 is recommended.
CDO library (cdo.dll) CDO Rendering library (cdohtml.dll)	Exchange Server 5.5 Service Pack 1 installs CDO library 1.21 and CDO Rendering library 1.21. Outlook installs CDO Library 1.21.
For the client: A Web browser or Outlook	You can run the client software on the same machine or on a separate machine.

Table 12-4. *Installation requirements for the Intranet News application.*

To set up the Intranet News application, you first need to install the application. Copy the Intranet News folder from the companion CD to your Web server location where you want to run the application.

Start the IIS administration program. Create a virtual directory that points to the location where you copied the Intranet News files, and name the virtual directory *exchnews*. Enable Execute permissions for the virtual directory. This allows you to use the following URL to access your intranet news site: *http://*yourservername*/exchnews*.

> **NOTE** If you use a different virtual directory name, you will have to edit the file virtroot.inc accordingly.

Included with the Intranet News files is a file named Exchnews.pst. Make sure the Read-Only flag for this file is unchecked. Launch Outlook and, from the File menu, point to Open and then choose Personal Folders File (.pst). In the Open Personal Folders dialog box, select the Exchnews.pst file, and click OK. In the Outlook Folder List, expand the Exchange Intranet News file folder. While holding the Ctrl key, copy the Exchnews folder to All Public Folders.

> **NOTE** You must install the Exchnews folder to All Public Folders or the application will not work. If you cannot install the application there, you can modify the code contained in the Intranet News application so that it looks for the folder in another location, or you can retrieve the folder by using its EntryID.

Right-click on the Exchnews public folder and select Properties. Click on the Permissions tab, and give the Anonymous user Read Items permissions. If you want the folder to be moderated, click on the Administration tab and then click the Moderated Folder button. Fill out the information in the Moderated Folder dialog box. Any new items sent to the folder will first be sent to the moderators you select.

To make it easier for you and your users to e-mail information into the Exchnews public folder, you can enable displaying the folder in the Global Address List (GAL). To do this, launch the Exchange Administrator program. Expand the Folders tree and then the Public Folders tree. Select the Exchnews public folder. Choose Properties from the File menu, click on the Advanced tab, and uncheck the check box named Hide From Address Book. Click OK.

To enable the Intranet News application to anonymously access the Exchnews folder, expand the Configuration tree for your Exchange Server site while still in the Exchange Administrator program. Select the Protocols icon, and then double-click HTTP (Web) Site Settings in the right pane. Make sure that the check box named Allow Anonymous Users To Access The Anonymous Public Folders is checked. Then click on the Folder Shortcuts tab. Click New, and select the Exchnews public folder in the tree. Click OK twice.

You're finished. You can now add news to the Exchnews public folder. Test it from the URL *http://*yourservername*/exchnews*.

Anonymous Logon

The Intranet News application uses the anonymous logon capabilities of the CDO library because the application does not require people to authenticate before being able to read the news ticker on the site. The way anonymous access works in the CDO library is that Exchange Server supports a new anonymous user permission feature, which allows you to set the permissions for all users who access the server anonymously. By using this feature, you can set whether anonymous users can create, delete, read, or modify items in folders.

Developers typically forget that when they set up the application, they need to actually publish the folder as an anonymous folder using the Exchange Administrator program. Even though you might give users anonymous access to the folder, the folder will not appear in the anonymous Public Folder hierarchy through CDO. Exchange Server has a very good reason for making you take this explicit step to publish the folder anonymously: you probably do not want anonymous users browsing through your Public Folder hierarchy. Exchange Server keeps a list of the published anonymous folders in the Exchange Server directory so that you can retrieve them by using the CDO Rendering library, as you'll see in the next section.

The Intranet News application logon is different from the previous two logon methods we've seen because it uses anonymous logons. Instead of storing the CDO

session in an ASP session variable, as in the Calendar of Events application, the Intranet News application stores the CDO session in the Application object so that you can keep sessions for different users distinct. You want to keep the sessions separate because your users are operating in different security and configuration contexts.

With anonymous access, all users are treated in the same way. They have the same security context and are not identified individually by CDO. Anonymous access assumes you do not care when an anonymous user logs on or logs off, because these processes are not unique to the individual user. Therefore, to increase performance, the application stores a valid anonymous CDO session in the ASP application scope and makes all users who access the application share this session.

One more issue to consider for anonymous users is that they cannot access folders other than Public Folders. This means that a user cannot access a mailbox when logging in using the anonymous method. Instead, the user is allowed to access only the published list of Public Folders that you set up in the Exchange Server Administrator program.

When a user logs on anonymously, you call the *Logon* method of the CDO Session object, which is the same method you call to authenticate a user. However, instead of passing in the server name and the mailbox as the profile information, you must pass in the Exchange Server enterprise name, the site name, and the configuration container; the server's container; and the server name. For example, if your Exchange Server organization was CompanyABC, your site was New York, and your server name in that site was Exchange1, the profile information would look like this:

```
/o=CompanyABC/ou=New York/cn=Configuration/cn=Servers/cn=Exchange1
```

You do not have to hard-code the Exchange Server organization and site information into your applications. Instead the CDO Rendering library allows you to query this information dynamically from the registry or from the Exchange Server directory service in your application. This query is accomplished by using the *LoadConfiguration* method on the RenderingApplication object and then calling the *ConfigParameter* method.

The first thing you must do when logging a user on anonymously is retrieve the enterprise, site, and server name from the Microsoft Windows registry. You cannot call the *LoadConfiguration* method with the Exchange Server directory as the source without first retrieving the necessary information from the registry because CDO must know which directory server to read the requested information from. This step is accomplished in the following code:

```
Dim objRenderApp
Set objRenderApp= Application( "RenderApplication")
'1 means load configuration from the registry
objRenderApp.LoadConfiguration 1, _
   "HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\" & _
```

```
   MSExchangeWeb\Parameters"
If Not ReportError( _
"RenderingApplication.LoadConfiguration from registry") Then
   bstrEnterprise= objRenderApp.ConfigParameter("Enterprise")
   bstrSite      = objRenderApp.ConfigParameter("Site")
   bstrServer    = objRenderApp.ConfigParameter("Server")
End If
```

Next you create the profile string, which specifies the anonymous account you want the user to log on as, and create a valid CDO object. Then, you must call the *Logon* method to actually create an anonymous session with the Exchange server. The following code implements these steps:

```
bstrProfileInfo = "/o=" + bstrEnterprise + "/ou=" + bstrSite + _
   "/cn=Configuration/cn=Servers/cn=" + bstrServer + _
   vbLF + "anon" + vbLF + "anon"
Err.Clear
   Set objAMSession1 = Server.CreateObject("MAPI.Session")
If Not ReportError( "create MAPI.Session") Then
   Set objRenderApp= Application( "RenderApplication")
   Err.Clear
   objAMSession1.Logon "", "", False, True, 0, True, bstrProfileInfo
```

Now you have an anonymous session with the Exchange server. However, you must remember to not only store the anonymous session object so that you can share it throughout your ASP application, but also to store the security context handle for the anonymous user session so that when your ASP application ends, CDO can kill the current anonymous sessions cleanly. The following code implements storing the security context handle in an *Application* scope variable:

```
If Not ReportError( "Anonymous Logon") Then
   Set Application("AMAnonSession") = objAMSession1
   Application("hImp") = objRenderApp.ImpID
```

IIS will call the following code, taken from Global.asa, when ending the ASP application to impersonate the correct security context for the anonymous session, and then it will set the *Application* variable for the anonymous session to *Nothing*:

```
Sub Application_OnEnd
   Set objRenderApp = Application("RenderApplication")
   hImp = Application("hImp")
   If Not IsEmpty(hImp) Then
      objRenderApp.Impersonate(hImp)
   End If
   Set Application("AMAnonFolders")= Nothing
   Set Application("AMAnonSession")= Nothing
   Set Application("RenderApplication") = Nothing
End Sub
```

After logging on, your application should attempt to open the Public Folder store to make sure that the anonymous logon was successful. The easiest way to find the Public Folder store with an anonymous logon is to scroll through the InfoStores collection and use the property *PR_STORE_SUPPORT_MASK (&H340D0003)*. This property contains a bitmask of flags that describe the characteristics of an InfoStore object. One of these flags, STORE_PUBLIC_FOLDERS (&H00004000), identifies a Public Folder store. The following code shows you how to use these properties to find the Public Folder store in your anonymous logon:

```
For i = 1 To objStores.Count
   Set objStore = objStores.Item(i)
   'PR_STORE_SUPPORT_MASK
   lMask = objStore.Fields.Item(&H340D0003)
   'Err.Clear
   'STORE_PUBLIC_FOLDERS
   If lMask And &H00004000 Then
      Exit For
   End If
Next
```

Retrieving the Folder and Messages

The next step in the application is to find the Exchnews public folder in the published Public Folder list, access the messages the folder contains, and render the messages to the Internet Explorer 4.0 marquee control. To access the Exchnews folder, you first need to retrieve the list of published Public Folders for anonymous users. To do this, you use the same method that we used to access the enterprise and site information, the *LoadConfiguration* method. This time, however, we pass as a parameter the number *2*, which indicates that we want to load the information from the Exchange Server directory rather than from the registry. Once CDO loads the information from the directory, we can use the *ConfigParameter* method to retrieve specific parameters from the Exchange Server directory. One of these parameters is the list of anonymously published Public Folders. This list is returned as a string array of EntryIDs for the anonymous Public Folders. The following code implements this process:

```
'2 means load configuration from the Exchange Server Directory
objRenderApp.LoadConfiguration 2, ""
If Not ReportError( _
"RenderingApplication.LoadConfiguration from DS") Then
   amFolders = objRenderApp.ConfigParameter( _
      "Published Public Folders")
```

We then must add two items to the list of anonymous folders: a dummy folder, which represents the root of all the public folders; and an InfoStore object, which represents the Public Folder store. Even though we will not use either of these items in this application, you should do this whenever you are accessing anonymous folders

using an anonymous logon. These two items allow the CDO Rendering library to correctly render the anonymous Public Folder information store. If you do not add these items, you might receive an error, or your folder hierarchy might look incorrect when rendered. The code for adding these items takes advantage of dynamic arrays in VBScript, as you can see in this snippet of code:

```
iFolderCount = UBound(amFolders)
ReDim Preserve amFolders(iFolderCount + 2)
'To the list of folders, add two things:
'a name for the pseudofolder we're making up...
amFolders(iFolderCount + 1) = "Public Folders"
'and a store interface so that the renderer can get
'data from the folders
Set objStores = objAMAnonSession.InfoStores
For idx = 1 To objStores.Count
   Set objStore = objStores.Item(idx)
   'PR_STORE_SUPPORT_MASK
   lMask = objStore.Fields.Item(&H340D0003)
   'STORE_PUBLIC_FOLDERS
   If lMask And &H00004000 Then
      Set amFolders(iFolderCount + 2) = objStore
      Exit For
   End If
Next
Application( "AMAnonFolders")= amFolders
```

Now that we have the correct list of items in the array for all the anonymous Public Folders, we need to find the Exchnews public folder. Scroll through the array of folder EntryIDs, retrieve each folder, and then check the name of the folder against the literal "Exchnews". When we find the Exchnews folder, we should break out of the loop because hundreds or thousands of anonymous Public Folders could be available. As you can see in the following code, when the Exchnews folder is found, the folder is set to an object variable, its Messages collection is retrieved, and, using the *Sort* method on the Messages collection, the items are sorted in descending order so that the most recent messages are moved to the top of the Internet Explorer marquee control.

```
If CheckAMAnonSession Then
   Set objAMAnonSession= Application( "AMAnonSession")
End If
If CheckAMAnonFolders Then
   amFolders= Application( "AMAnonFolders")
End If
For iFolder = LBound( amFolders) to (UBound( amFolders) - 2)
   'amFolders is an array of folder IDs.
   'Get the folder ID of the Exchnews public folder.
```

(continued)

```
      Set objFolder= objAMAnonSession.GetFolder( amFolders( iFolder), _
         NULL)
      if objFolder.Name = "Exchnews" then
         exchnewsid = amFolders(iFolder)
         exit for
      end if
   Next
   set objFolder = objAMAnonSession.GetFolder(exchnewsid,NULL)
   set objMessages = objFolder.Messages
   'Sort the messages descending so that newer messages are at the top
   objMessages.Sort 2
```

Displaying the News Items

Once we have retrieved and sorted the items in the folder, we need to put the items into the marquee control. Suppose some users want to categorize their news items so that the items can be read in context. For example, the human resources department might want to submit items to the folder that represent different human resources offerings, which can be broken down into categories such as benefits, work and life balance, and training. Human resources might also want to include a banner on the screen before these messages appear to tell users which category the news corresponds to. To implement this sorting, the application uses the *Categories* property on Outlook messages. A user can assign a single category to a news item, and the application will display the selected category in the marquee control. If the user does not enter a category for the item, the application automatically displays the item as a general news item.

The application automatically detects as newer items any items entered into the news system within seven days of the current date. To highlight all new items in the Public Folder, these items receive a new graphic next to their text.

The next chunk of code implements all of this. To scroll through all news items in the folder, the application uses a For...Each loop on the Messages collection for the folder. The application then checks the *Categories* property on the item to see whether any categories exist. (Remember that the *Categories* property is an array of strings, and to access an individual member, you must specify the index using the following syntax: objMessage.Categories()(Index).) The application then checks the date of the message, and if the message was received within the last seven days, the application adds the new graphic to the item. The marquee control also has hyperlinks to the messages. When the user holds the mouse pointer over a hyperlink or holds the mouse button down while the mouse pointer is over the marquee, the control will stop scrolling so that the user does not have to chase the hyperlinks and can read the text.

```
<TD WIDTH="20%" VALIGN="Top">
<MARQUEE DIRECTION=UP ID="Marquee" BEHAVIOR=SCROLL SCROLLAMOUNT=10
SCROLLDELAY=500 TITLE=
"Hold the mouse down or over an item to stop the News Ticker."
ONMOUSEDOWN="this.stop();"
ONMOUSEUP="this.start();">
<% for each objMessage in objMessages %>
   <DIV CLASS=big>
   <%
   on error resume next
   strCatName = objMessage.Categories(0)(0)
   if strCatName = "" then
      strCatName = "General"
   end if
   %>
   <%=strCatName%> News</DIV>
   <hr>
   <a href="details.asp?id=<%=objMessage.ID%>"
   ONMOUSEOVER ="this.style.textDecorationUnderline=true;
   document.all['Marquee'].stop()"
   ONMOUSEOUT="this.style.textDecorationUnderline=false;
   document.all['Marquee'].start()">
   <% if (datediff("d", objMessage.TimeReceived, Date()) <= 7) then %>
      <img src="newicon.gif" border=0 align="center">
   <% end if %>
   <FONT FACE="VERDANA, ARIAL, HELVETICA" SIZE="2">
   <%=objMessage.Subject%>
   </FONT>
   </a>
   <P>
   <% strCatName = ""
next %>
```

Reading the Details of a Specific News Item

Every news item scrolled through the marquee has a hyperlink to the file details.asp. The details.asp file allows the user to drill into the specifics of a news item and to see any rich text, attachments, or hyperlinks that the author of a news item entered into the Outlook message sent to the Exchnews public folder. This information is rendered to the browser by using the CDO Rendering library. An example of a details page for an item is shown in Figure 12-33. Compare it to the same details page presented as an Outlook message, shown in Figure 12-34.

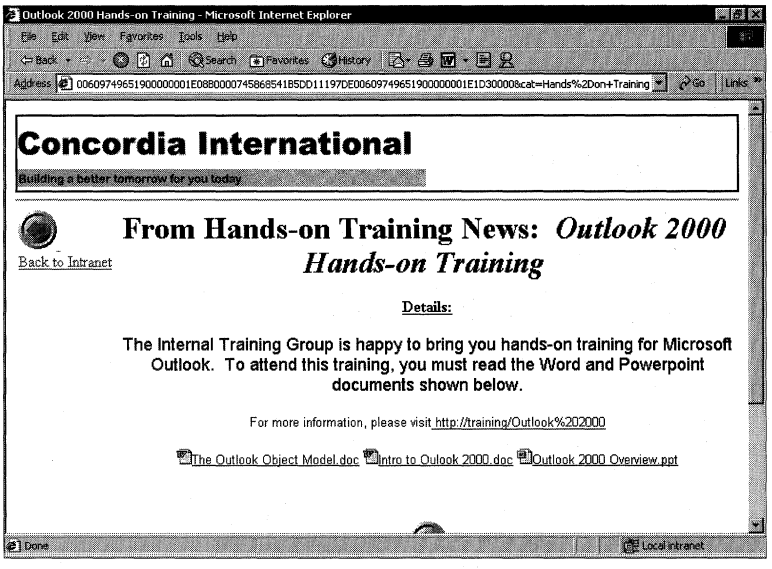


Figure 12-33. *The details of the intranet news item include rich text, hyperlinks, and attachments.*

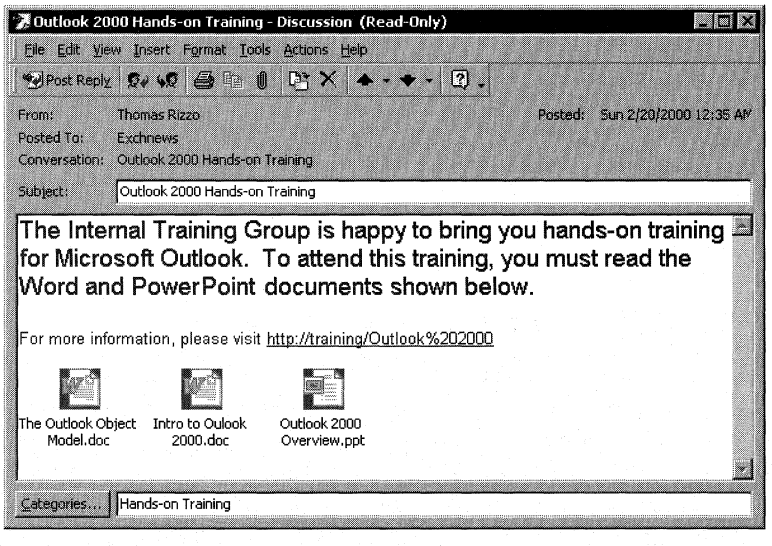


Figure 12-34. *The item in Figure 12-33, shown as an Outlook message. Notice how the Web and Outlook versions look almost identical. This is due to the CDO Rendering library's automatic conversion of rich text to HTML.*

The Calendar of Events and Intranet News applications use similar code to render information to the Web user, but the Intranet News application uses the CDO Rendering library in a slightly different way. In the Calendar of Events application,

the HTML generated by the CDO Rendering library is added to the Response object of the ASP object model. In the Intranet News application, the HTML produced is not added to the Response object but rather is placed into a string so that the application can modify the HTML before it is presented to the user. This modification replaces the generic paper-clip icon that CDO automatically renders for all attachments with the specific application icons for Microsoft Office products. You will see how this functionality is achieved a little later.

Before attempting to render the details of the news item to the browser, we first need to change the virtual root of the Rendering application. If we do not change this root, all virtual roots in the rendered hyperlinks will point to the /Exchange virtual root. Then we have to create an object renderer because we will be rendering two specific properties on the item: the subject and the message body. The following code shows you how to accomplish these tasks:

```
'Change the virtual root for the rendering application
Set objRenderApp = Application("RenderApplication")
objRenderApp.VirtualRoot = virtroot
'Create an object renderer
set objObjRenderer = objRenderApp.CreateRenderer(2)
objObjRenderer.DataSource = objMessage
```

To create the page, we have to render the rich-text message body into a string. To do this, instead of passing a Response object to the *RenderProperty* method, we set a string variable equal to the *RenderProperty* method, as shown here:

```
'Render the HTML into a string
strHTML = objObjRenderer.RenderProperty(ActMsgPR_RTF_COMPRESSED, 0)
```

Now that we have the HTML that the CDO Rendering library would normally display in the browser, we need to check to see whether the message has any attachments. If it does, then we need to scroll through the HTML and change the image source to point to the Microsoft Word, the Microsoft Excel, or the Microsoft PowerPoint icons instead of the generic paper-clip icons. The following code shows how to check for attachments in the message by using the Attachments collection and *Count* property:

```
set oAttachments = objMessage.Attachments
intAttachCount = oAttachments.Count
if intAttachCount > 0 then
'Need to find any Office documents by using the extensions
```

If there are attachments, we need to scroll through the attachments to determine what type of document they are. This is where the code gets into manipulating strings, and the degree to which it's confusing depends on how well you know the string functions in VBScript! I built this code so that you can add your custom extension and image types to it, which enables documents to be displayed with their specific icons rather than with generic icons. If the code does not find the text for

the application in the document, it leaves the generic icon. Following is the code for replacing the images in the message text of an item for Word documents. The code for PowerPoint and Excel attachments is very similar and can be found in the code on the companion CD:

```
'Find all the Word documents
found = 1
Do while (found <> 0 or found <> Null)
   found = instr(found, strHTML, ".doc</A>")
   if found <> 0 then
      strIcon = "Iword.gif"
      revfound = instrrev(strHTML, "generic.gif", found)
      newstrHTML = Replace(strHTML, "generic.gif", strIcon, _
         revfound,1)
      origstrHTML = Left(strHTML, revfound-1)
      strHTML = origstrHTML & newstrHTML
      found = found + 1
   end if
Loop
```

This code sets a variable named *found* equal to *1*. The variable is used as the starting point for the string and also as a Boolean for the Do...While loop, which parses the string. The Do...While loop searches through the string until no .doc extensions representing Word documents are found or until the *InStr* function returns a Null value, which would indicate that the source or string being searched for is Null—in other words, some weird condition has occurred in string processing. When searching through the string, the application knows that the CDO Rendering library always follows the .doc extension with an ending hyperlink tag. Adding *</A>* to the search string almost guarantees that the search will not return random .doc strings in the text of the message.

If the application finds a location where the .doc</A> string occurs, the application uses the *InStrRev* VBScript function to perform a reverse lookup from the location of .doc</A> back through the string to the Word documents corresponding to the generic paper-clip icon. (The CDO Rendering library will always use the generic.gif image for attachments because this image is hard-coded for use in the CDO code.) The code then uses the *Replace* function of VBScript and replaces generic.gif with the Word icon. The final parameter for the *Replace* function, *1*, tells the code to replace only one instance of generic.gif in the string. This stops VBScript from going through the entire string and replacing all references to generic.gif.

You might be wondering why the code then takes the leftmost portion of the string up to the point where the new image string was replaced. The reason is that the *Replace* function does not return the entire string after making the replacements. Rather, this function returns from the point where the replacement started to the end of the string. This means that our HTML string is now missing its entire left-hand

portion up to the point where we replaced the image. For this reason, the code combines the return value from the *Replace* function with the return value from the *Left* function to re-create the original string with our new replacement. Then the code increments the *found* variable so that we do not enter into an infinite loop, finding the same .doc extension at the same point in the string.

To render the final HTML string that we create, the application calls the *Write* method on the Response ASP object to send the string as HTML to the browser. The code also uses the *RenderProperty* method of the CDO Rendering library to display the subject of the news item from the message, as shown in the following code:

```
<td width="980" height="422" valign="top" rowspan="3" align="center">
<p><h1><B>From <%=Request.QueryString("cat")%> News: </B>
<!--
Render the Subject
-->
<I><%objObjRenderer.RenderProperty ActMsgPR_SUBJECT, 0, Response%>
</I></h1></p>
<B><U>Details:<P></U></B>
<!--
Render the body with our replacements
-->
<%response.write strHTML%>
```

CDO VISUAL BASIC APPLICATION

The last application we will look at is a CDO application built using Visual Basic. This application allows users to log on to their Exchange server using CDO, query the server for other users, and retrieve information about those users. This application shows you how to program CDO with Visual Basic, which is different from programming CDO with VBScript and ASP. This application also shows you how to use the AddressEntryFilter object. Figure 12-35 shows the application in action.

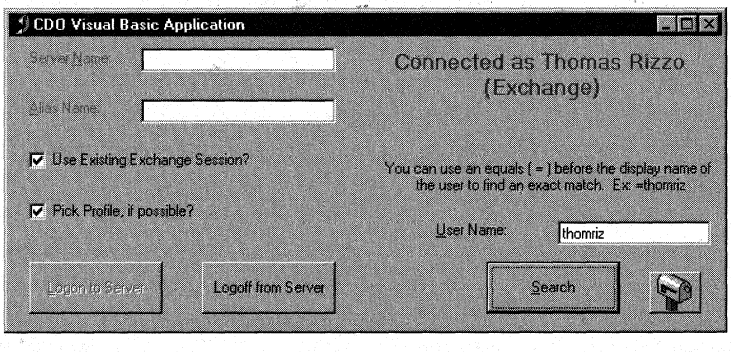


Figure 12-35. *The CDO Visual Basic application.*

Setting Up the Application

Before you can install the application, you must have a Windows NT 4.0 Server or Windows 2000 Server and a client with certain software installed. Table 12-5 describes the installation requirements.

Software Requirements	*Installation Notes*
Exchange Server 5.5 Service Pack 1	Service Pack 3 is recommended.
CDO library (cdo.dll)	Exchange Server 5.5 Service Pack 1 installs CDO library 1.21. Outlook installs CDO library 1.21.
For the client: Outlook	

Table 12-5. *Installation requirements for the CDO Visual Basic Application.*

To install the CDO Visual Basic application, run the Setup.exe file in the CDO VB folder on the companion CD and follow the instructions.

Programming CDO with Visual Basic

The main differences between programming CDO with VBScript and ASP and programming CDO with Visual Basic is that Visual Basic allows you to use early binding of objects in the CDO library. By declaring your variables as a specific type of object, the variables will be bound early. For example, in Visual Basic, you can use the Dim statement to declare a variable as a CDO Session object by using the following statement:

```
Dim oSession as MAPI.Session
```

Once you declare a variable, you can take advantage of some of the powerful features of the Visual Basic development environment, such as Auto List Members, which lists the available properties and methods for an object, and Auto Quick Info, which displays the syntax for a statement. For example, if in the code window you start typing the name of the *oSession* variable and then the dot operator (.), Visual Basic will automatically display the properties and methods for the CDO Session object. Also, using early binding allows your application to execute faster. This is because the binding takes place at compile time rather than at run time. VBScript and ASP cannot use early binding and therefore always default to late binding when creating objects.

To use CDO in Visual Basic, add a reference to the CDO library. By adding this reference, you can declare variables of a specific CDO type in your code, and you make the CDO objects appear in the Visual Basic object browser. You use the object

browser to view information about libraries, such as properties, methods, events, constants, classes, and other information.

To add the reference to the CDO library, in Visual Basic select References from the Project menu. Scroll down until you find Microsoft CDO 1.21 library, and add a check mark next to it. If you want to add a reference to the CDO Rendering library, add a check mark next to Collaborative Data Objects Rendering Library 1.2, and click OK. Now you can take advantage of early binding with your CDO objects, and the CDO library will be available in the Visual Basic object browser. Most of the time, you will not use the CDO Rendering library in your client-based applications. Instead, you will use this library in your Web-based applications.

Logging On the User

As we have discussed throughout this chapter, you cannot create any other objects in the CDO library without first creating a CDO Session object and successfully logging on with that Session object. Because we are developing a Visual Basic application, we do not have to worry about a Global.asa file or authenticating the user—CDO will leverage the Windows NT credentials of the user currently logged on. This makes logging on as a user much easier, as you can see in the following authenticated logon code:

```
Dim oRecipients As MAPI.Recipients
Dim oRecipient As MAPI.Recipient
Dim oInfoStores As MAPI.InfoStores
Dim oInfoStore As MAPI.InfoStore
Dim oInbox As MAPI.Folder
Dim boolUseCurrentSession, boolLogonDialog
Private Sub cmdLogon_Click()
   On Error Resume Next
   Err.Clear
   'Check to see whether user wants to use a current session.
   'If so, piggyback on that session.
   If boolUseCurrentSession = 0 Then
      If (txtServerName.Text <> "") And _
      (txtAliasName.Text <> "") Then
         strProfileInfo = txtServerName & vbLf & txtAliasName
         oSession.Logon NewSession:=True, NoMail:=False, _
         showDialog:=boolLogonDialog, ProfileInfo:=strProfileInfo
         strConnectedServer = " to " & txtServerName.Text
      Else
         MsgBox "You need to enter a value in the " & _
            "Server or Alias name.", _
            vbOKOnly + vbExclamation, "CDO Logon"
         Exit Sub
      End If
```

(continued)

```
    Else
        oSession.Logon NewSession:=False, showDialog:=boolLogonDialog
        strConnectedServer = ""
    End If
    If (Err.Number <> 0) Or _
    (oSession.CurrentUser.Name = "Unknown") Then
        'Not a good logon; log off and exit
        oSession.Logoff
        MsgBox "Logon error!", vbOKOnly + vbExclamation, "CDO Logon"
        Exit Sub
    End If

    'Check store state to see whether online or offline
    Set oInbox = oSession.Inbox
    strStoreID = oInbox.StoreID
    Set oInfoStore = oSession.GetInfoStore(strStoreID)
    If oInfoStore.Fields(&H6632000B).Value = True Then
        strConnectedServer = " Offline"
    End If

    'Enable other buttons on the form
    cmdLogoff.Enabled = True
    cmdLogon.Enabled = False
    txtUserName.Enabled = True
    cmdSearch.Enabled = True
    cmdViewAB.Enabled = True
    lblUserName.Enabled = True
    'Change the label to indicate status
    lblConnected.Caption = "Connected" & strConnectedServer _
        & " as " & oSession.CurrentUser.Name
End Sub
```

To support early binding, a number of variables are declared as specific CDO object types. The code tries to log on to the Exchange server by using the CDO *Logon* method. Unlike the ASP code we saw earlier, in this code we can leverage existing sessions between the client and the Exchange server rather than always create new sessions. The user can enable this functionality by checking the Use Existing Exchange Session check box. (See Figure 12-35.) The existing session, typically an Outlook client session, is used by CDO to connect to the Exchange server.

After the user logs on, the code finds the InfoStore object associated with the user's mailbox. The Fields collection on InfoStore is used to look up a specific property, *PR_STORE_OFFLINE* (*&H 6632000B*), which contains either True or False; True indicates that the current InfoStore is an offline replica. The value for this property is used in the status text, which indicates the connection state of the user, as shown in Figure 12-36.

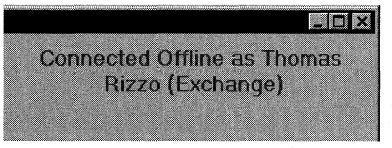


Figure 12-36. *If the user is working offline, the connection status message displays this information.*

Finding the Details of the Specific User

After logging on, the user can type in a name in the User Name text box. The name entered is used by the application to search the directory or distribution list. The search is implemented by using the AddressEntryFilter object in the CDO library. The AddressEntryFilter object is similar to the MessageFilter object, which we examined in the Calendar of Events application. The only difference between them is that the AddressEntryFilter object is used with objects in the directory, and the MessageFilter object is used with messages in a folder. Following is the code that searches for the user by using the AddressEntryFilter object and displays the results:

```
Private Sub cmdSearch_Click()
   On Error Resume Next
   'The On Error is to handle the user canceling the
   'details dialog box
   Err.Clear
   If txtUserName.Text = "" Then
      MsgBox "No User Specified", vbOKOnly + vbExclamation, _
         "User Search"
      Exit Sub
   Else
      Set oAddressList = oSession.GetAddressList(CdoAddressListGAL)
      Set oAddressEntries = oAddressList.AddressEntries
      Set oAddEntryFilter = oAddressEntries.Filter
      oAddEntryFilter.Name = txtUserName.Text
      If oAddressEntries.Count < 1 Then
         MsgBox "No entries found", vbOKOnly, "Search"
      ElseIf oAddressEntries.Count > 1 Then
         MsgBox "Ambiguous entries found", vbOKOnly, "Search"
      Else
         Set oAddressEntry = oAddressEntries.GetFirst
         oAddressEntry.Details
      End If
   End If
End Sub
```

This code gets the GAL, either offline or online, by using the *GetAddressList* method on the Session object. It then instantiates an AddressEntryFilter object by using

the *Filter* property on the AddressEntries collection. To specify the condition for the filter, the *Name* property on the AddressEntryFilter object is set to the name typed in by the user. This name can be either the user's display name, such as *Thomas Rizzo (Exchange)*, or the alias of the user, such as *thomriz*. CDO also supports direct matches when you place the equal sign (=) before your text, as in *=Thomas Rizzo*.

Once the filter is set, the code retrieves the count of the newly restricted AddressEntries collection to determine how many AddressEntry objects were returned. If more than one AddressEntry object was returned, the code displays an ambiguous name error to notify the user that more specific criteria is needed. If less than one AddressEntry object is returned, the code displays that no entries meet the criteria of the user. If exactly one AddressEntry object is returned, the code uses the *Details* method of the AddressEntry object to display the information about the directory object, as shown in Figure 12-37.

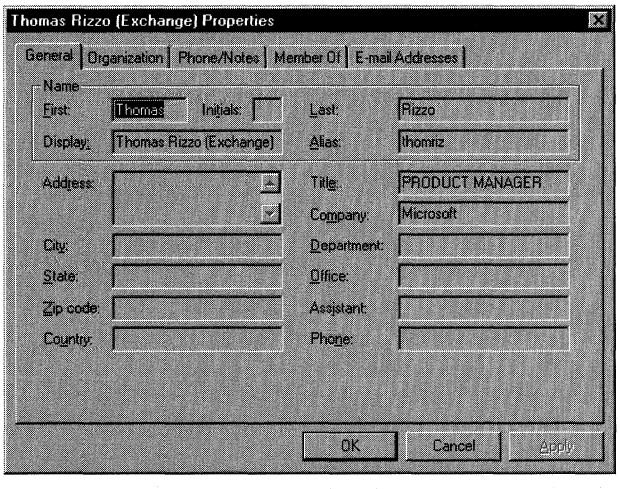


Figure 12-37. *The details page of an AddressEntry object. You can see not only the name and alias of the user but also organizational information such as the manager of the user.*

Finally, a subroutine is included to handle the run-time error thrown by CDO when the user clicks Cancel in the Properties dialog box displayed by the *Details* method.

CDO Tips and Pitfalls

The CDO library is powerful and approachable, but you can run into problems if you aren't careful when writing your code. This section introduces some tips and tricks you should use, and some pitfalls you should avoid. Many of the pitfalls I outline are from personal experience—they are quite frustrating, so I recommend you read this section before attempting to write any CDO code.

Avoid the GetNext Trap

Let's jump right in! Look at the following code and try to figure out what is wrong:

```
MsgBox oSession.Inbox.Messages.GetFirst.Subject
For Counter = 2 To oSession.Inbox.Messages.Count
   MsgBox oSession.Inbox.Messages.GetNext.Subject
Next
```

The same subject will appear in your message box as many times as the number of messages in your Inbox. Despite what the code looks like, it won't recurse through your Inbox, because if you don't explicitly assign an object to a variable, CDO will create needed temporary objects for each statement and then discard the object after the statement. This means that you will instantiate a new object every time you loop in the For...Next loop. Each new object does not maintain the old state of the previous temporary object, so the object will always point to the first message in the collection. So you should set explicit variables to refer to a collection to get the desired functionality. The following listing shows the rewritten code, which behaves as expected:

```
Set oMessages = oSession.Inbox.Messages
Set oMessage = oMessages.GetFirst
MsgBox oMessage.Subject
For Counter = 2 To oMessages.Count
   Set oMessage = oMessages.GetNext
   MsgBox oMessage.Subject
Next
```

Avoid Temporary Objects, If Possible

Whenever possible, avoid the use of temporary objects, as demonstrated in the previous pitfall. Don't spend a lot of time scouring your code to get rid of temporary objects unless you are a major offender of this rule. Sometimes you'll want to use temporary objects to represent the different CDO objects rather than declare variables. However, using temporary objects should be an exception and not a rule in your coding practices.

Use Early Binding with Visual Basic

To improve the performance of your Visual Basic CDO applications, always try to use early binding by declaring your CDO variables as specific CDO objects. Not only will you find that writing your code is easier because Visual Basic can perform type-checking as well as help you finish statements in your code, but you'll also find that your users will thank you for the application's improved performance.

Use With Statements

You use the dot operator to set a property, call a method, or access another object. Essentially, each dot represents additional code that must be executed. If you can reduce the number of dot operators in your code, you can improve performance of your application. One way to do this is to use With statements. For example, consider the following code snippet, which has no With statements and is inefficient both from a performance perspective and an ease-of-reading perspective:

```
MsgBox "Text: " & oSession.Inbox.Messages.GetFirst.Text
MsgBox "Subj: " & oSession.Inbox.Messages.GetFirst.Subject
```

Now consider the next bit of code, which does use the With statement. This code will execute faster:

```
With oSession.Inbox.Messages.GetFirst
   MsgBox "Text: " & .Text
   MsgBox "Subj: " & .Subject
End With
```

The rule of thumb is to think of dots in your code as expensive.

Avoid the Dreaded ASP 0115 Error

When writing CDO applications using ASP, the best tip I can give you is to use the code from this book to handle your logons and logoffs from CDO and ASP sessions. The most common pitfall that new and even experienced CDO developers run into when writing ASP applications is forgetting to insert the correct impersonation code into the Global.asa, which properly destroys the CDO and ASP sessions. When a user attempts to access your Web application after IIS attempts to use the wrong context to destroy these objects, the application returns the ASP 0115 error, which means that a trappable error has occurred in an external object.

Avoid the MAPIE_FailOneProvider or CDOE_FailOneProvider Error

The final pitfall that I can help you avoid in your ASP applications is the CDOE_FailOneProvider error, which occurs when you try to access the root folder of the Public Folder InfoStore object or a folder in the mailbox of a specific user. Many developers have run into this error, especially those who are new to ASP programming. The common cause of this error is not changing the security context that IIS is using to access the Exchange server by authenticating the Web user using either Windows NT Challenge/Response authentication or Windows NT Basic authentication. Therefore, the Web user is trying to access the root of the Public Folder store or a user's mailbox using the Windows NT credentials of the anonymous IIS user

account. Frequently this anonymous account doesn't have security permissions to access the Exchange server. When this is the case, CDO returns CDOE_FailOneProvider to indicate an error in accessing the information.

The easiest way to solve this problem is to use the logon and logoff code from the examples in this book. These examples, especially the Helpdesk application, authenticate users by prompting them for their Windows NT credentials before attempting to access any Exchange Server information.

Learn Your Properties and Their IDs Well

As you have seen throughout the chapter, many of the objects in the CDO library support the *Fields* property. The *Fields* property returns a Fields collection, which allows you to find custom and built-in properties using identifiers supplied by either Exchange Server or MAPI. One of the most powerful yet elusive features is this set of Exchange Server and MAPI properties. These properties allow you to perform operations on Exchange Server and Outlook items in situations where CDO does not provide objects. For example, in the Helpdesk application, user information is pulled out of the *AddressEntry* property by using the unique identifiers for department name, office location, and other properties. If you did not know these properties existed, you would think that their information was inaccessible from CDO because CDO does not provide explicit objects for them.

Another scenario illustrating why these unique properties are valuable is that of setting up folders to work offline. The documentation on this process is hard to find, but MAPI provides a property called *PR_OFFLINE_FLAGS (&H663D0003)*, which contains a zero (0) if the folder is not currently set to synchronize offline and a 1 if it is. By using this property, you can programmatically set any folder in the mailbox of a user to synchronize offline—the user does not have to set synchronization manually through Outlook. If this field does not exist in the Fields collection already, you will need to add it to the collection by using the *Add* method.

The best place to find the information about the properties you can use with the Fields collection is in the CDO help file under "MAPI Property Tags," or in the Platform Software Development Kit (SDK) section of the MSDN Library under "Database and Messaging Services," "Messaging API (MAPI)," "Reference," and then "MAPI Properties." For Exchange Server properties, look in the MSDN Library and perform a keyword find on the Index tab for "Microsoft Exchange Server Message Properties." All of these properties combined can provide new functionality to your applications, even though CDO may not provide explicit objects for this functionality.

Chapter 13

The Event Scripting Agent

One of the most important additions to Microsoft Exchange Server 5.5 and Exchange Server application development is the Microsoft Exchange Event Service and Event Scripting Agent technology. This technology allows developers to write custom scripts or custom agents to capture and respond to events generated by Exchange Server folders. It extends the possibilities for what you can develop on the Exchange platform—from automated administrative tasks to sophisticated workflow applications. In this chapter, you will learn about the architecture of the Exchange Event Service, how to set it all up, and how to develop your own agents and applications that take advantage of the technology.

ARCHITECTURE OF THE EXCHANGE EVENT SERVICE

The Event Service is implemented as a Microsoft Windows NT service that receives notifications from server-based folders about the state of folder items. The service architecture is structured like this: the service—events.exe—passes events, such as the creation of a new message in a folder, to the correct event handler—an agent—with some information about the source of the event, the message, and the folder that caused the event. This architecture is shown in Figure 13-1.

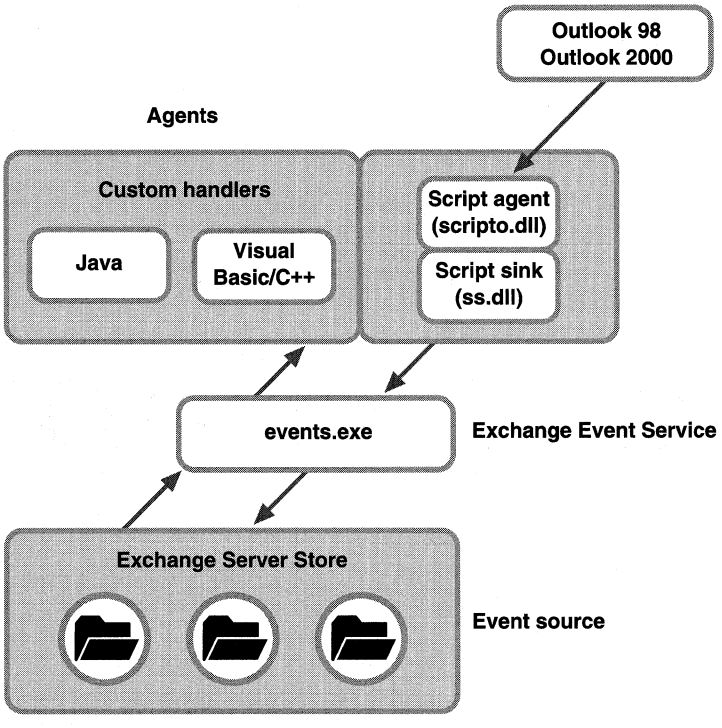


Figure 13-1. *Architecture of the Exchange Event Service.*

How does the Event Service know when an item is added, changed, or deleted in a particular folder? The Event Service is built on the same technology that Microsoft Outlook uses to perform local replication from the Exchange server to your Outlook client. This technology is called Incremental Change Synchronization (ICS). ICS allows the client—in this case, the Event Service—to query the information store on the server and request information about all changes that have occurred in a particular folder since the last synchronization. By using ICS, the Event Service never misses an event, even if it is taken offline. When the Event Service goes back online, it will query for any changes to the folders it is monitoring and then fire the correct events to the corresponding event handlers for that folder.

The Event Service fires events when an item is added, changed, or deleted in a folder, or according to time intervals. The events for adding, changing, and deleting items are self-explanatory, but the fourth event, the timer event, requires a little bit more explanation. You specify intervals indicating when to fire the timer event. These intervals can be hourly, daily, or weekly, depending on the needs of your application—for example, every 15 minutes, every 3 hours, or every week on Monday at 3:00 PM. In the application in this chapter, you will see how to use an interval to check the status of items in a folder.

The items that cause these folder events can be of any message class. For example, dragging and dropping a Microsoft Word document into a monitored public folder will fire the new message event. Notice that I say *public folder*. The Event Service can monitor only folders stored on an Exchange server. It will not monitor folders stored on the local machines of users. So you can monitor events on public folders and in user mailboxes if the user mailboxes are stored on an Exchange server. If you are using .pst files to store the mail of your users, you cannot monitor them for events. Most developers wonder whether .ost files are able to fire events because they are also stored on the client. They can if a user synchronizes her .ost file using the built-in capabilities of the Outlook client. When changes made in her .ost file are replicated to the server, the Event Service can fire events on those changes.

Once the Event Service realizes a change has occurred, it fires an event. Then it looks for a corresponding event handler in the folder. Associating an event handler with a specific event and folder is called binding. The Exchange Event Service ships with one prebuilt event handler, named the Exchange Event Scripting Agent, that you can bind to events. As you would guess by its name, the Event Scripting Agent is an event handler that allows you to write both Microsoft Visual Basic Scripting Edition (VBScript) and JScript scripts to perform actions when specific events occur. These scripts can automatically call Microsoft Collaboration Data Objects (CDO) functions. The scripts are passed a pre-logged-on CDO session, which we'll learn more about later in this chapter. From these scripts, you can also call other COM components such as ActiveX Data Objects (ADO), Active Directory Services Interfaces (ADSI), or even your own custom COM components that are written using Microsoft Visual Basic or Microsoft Visual C++.

> **NOTE** In addition to developing your own custom components to call in scripts, you can write your own event handlers. These custom event handlers must implement the *IExchangeEventHandler* interface as well as register themselves with the COM category CATID_ExchangeEventSink. Custom handlers are beyond the scope of this book. If you are interested in developing custom handlers, you should look at the help file named Agents.hlp, which is included with Exchange Server 5.5.

EVENT SERVICE CAUTIONS

The Exchange Event Service fires events asynchronously rather than synchronously in the context of the Exchange Information Store, so the Information Store won't block your event script or other processes or people from working on the items in the folder if your script hasn't run yet. A user or another process, then, could delete, move, or change an item before an event based on the item is fired and your script is executed. Your scripts will receive the proper events in this situation, but the items might not be available. For this reason, don't use the Event Service to monitor folders such as

your Inbox and Outbox that have very high volumes of items entering, leaving, or being deleted. In these types of folders, the chances are greater that the user or the rules engine on the server will move or delete the item before your script is run.

You shouldn't use the Event Service to provide a mechanism for "house rules" either. House rules are general rules containing business logic that you want installed on every folder in the system. Using the Event Service for a system that uses house rules will bog down the Exchange servers running the Event Service because of the high volume of messages generating events. Also, you would have to manually install the agent in every folder because the Event Service does not provide this capability. The Agent Install application discussed later in this chapter will help you get around the problem of manually installing scripts into folders. The Agent Install program will show you how to programmatically create and bind agents using the components that ship with the Exchange Event Service.

SETTING UP THE EVENT SERVICE

Before starting to work with the Event Service and writing agents, you first must install the service and get it running correctly in your environment. By default, the Event Service is installed when you install Exchange Server 5.5. However, if you are upgrading from a previous version of Exchange Server, you will need to add the Event Service during installation.

By default, the Event Service logs on using the credentials of the Exchange Service Account. While this account has permission to access many of the items stored in the Exchange server, it has very limited Windows NT permissions. You might want to change the Windows NT account under which the Event Service runs to change the access this account has and to audit the account. To change the account, change the Log On As settings in the Services applet of the Control Panel for the Microsoft Exchange Event Service, as shown in Figure 13-2.

If you do change the Windows NT account for the Event Service, make sure that the account you specify for the Event Service truly does have the Log On As A Service permission set in the User Manager For Domains. Also, make sure the account has the proper Exchange permissions so that it can access any of the mailboxes or public folders where scripts will be installed. By default, the Event Service passes a logged-on MAPI session to the Event Scripting Agent, so you do not have to write the logon code in the script. But the Event Service will try to log on to resources using the Windows NT account you specify for the service. If this Windows NT account does not have the proper permissions, your scripting agent will not work. You can set the permissions (such as Mailbox Owner and Send As permissions) for all necessary resources in the Exchange Administrator program.

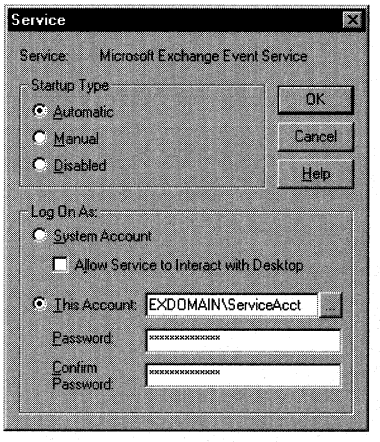


Figure 13-2. *You change the Windows NT account that the Event Service runs under by using the Service applet in the Control Panel.*

In addition to setting up the Windows NT account that the Event Service will run under, you must also give users permission to create agents. This is a two-step process. First, you must give users permission to create and bind agents in the system. This is accomplished by setting their permissions for a system folder named EventConfig_*servername*, which is shown in Figure 13-3.

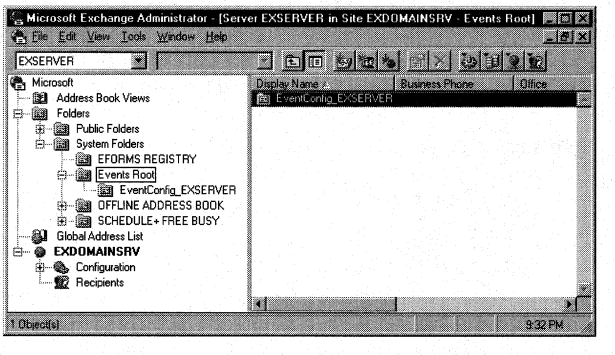


Figure 13-3. *The EventConfig_*servername *folder is found under the Events Root system folder. You must set user permissions for this folder if you want users to write agents.*

Locate this folder in your Exchange Administrator program, and select Properties from the File menu. Click the Client Permissions button, and in the Client Permissions dialog box shown in Figure 13-4, add users or distribution lists and assign them Author or higher permissions.

After you have assigned the proper permissions for the EventConfig_*servername* folder, the second step of the process is to configure the folder. To do this, start Outlook, right-click on a public folder or an Exchange Server folder, and select

Properties. On the Agents tab, as shown in Figure 13-5, you can create, change, disable, or delete agents in your folder. For the Agents tab to appear, you must be the owner of the folder and the Server Scripting add-in must be installed. By default, Outlook does not install the Server Scripting add-in. To install the Server Scripting add-in, select Options from the Tools menu, click on the Other tab, click Advanced Options, and then click Add-In Manager. Check the Server Scripting check box to add the Agents tab to the folders where you have permissions to create agents.

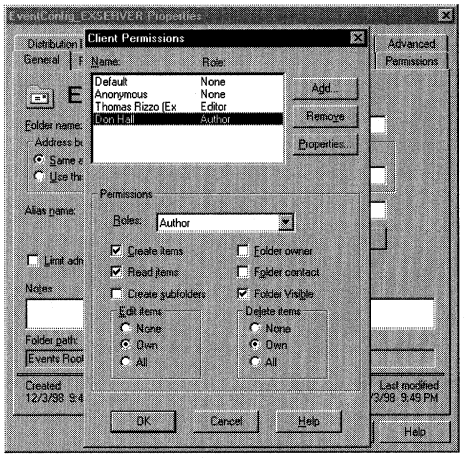


Figure 13-4. *In the Client Permissions dialog box, you assign users or distribution lists permissions to write agents. You must assign Author or higher permissions to these users before they can create agents.*

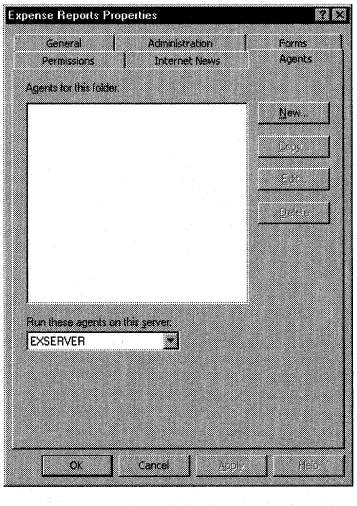


Figure 13-5. *The Agents tab for the Expense Reports public folder. You must have appropriate EventConfig_*servername *permissions, be an owner of the folder, and have the Server Scripting add-in installed to see the Agents tab in Outlook.*

REGISTRY SETTINGS FOR SCRIPT AUTHORS

Before creating your scripts, you should review settings for a few keys in the registry to optimize the debugging and control capabilities in your scripts. These modifications can lower the notification interval for ICS, making events fire faster, and can also increase the amount of information saved to the Windows NT event log. Follow these steps to review the script registry settings:

1. Open the Registry Editor (regedit.exe) on your Exchange server.
2. Locate the following key:

   ```
   HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\
   MSExchangeES\Parameters
   ```

   This key has a DWORD named Logging Level. Logging Level specifies how much information is written to the Windows NT Event Log. The value for Logging Level ranges from 0 through 5, where 0 is the default value. If Logging Level is set to 5, the maximum amount of information is logged. Adjust the Logging Level value according to your preference. Normally when I am developing scripts, I set Logging Level to 5.
3. DWORD Maximum Execution Time For Scripts In Seconds sets the maximum time a script can execute before it is terminated. When developing scripts that need to access data sources at other locations or on the network, such as databases or host systems, you might want to bump up the default value of 900 a bit so that your scripts are not prematurely terminated.
4. DWORD Maximum Size For Agent Log In KB sets the log size for your agents. The default value for this key is 32 KB. The log automatically overwrites older events as necessary when this size is exceeded.
5. Locate the following registry key:

   ```
   HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\
   MSExchangeIS\ParametersSystem
   ```

   If the DWORD value ICS Notification Interval does not already exist, add it and set the value to the number of seconds between each ICS notification to the Event Service. The default value is 60 seconds. However, for testing and production servers, you might want to lower this value to shorten the length of an interval between a change in the store and the Event Service being notified.

WRITING AGENTS BY USING SCRIPTS

The first step in writing scripts that run as part of the Exchange Event Service is to create an agent that acts as the event handler for certain folder events. To reach this interface, however, you are required to run certain versions of Outlook. To be a script writer for the Event Service, you must be running Outlook 97 version 8.03 or higher. The interface for creating new agents in Outlook is the Agents tab in the Properties dialog box, which was shown in Figure 13-5. To create a new agent, follow these steps:

1. Start Outlook 8.03 or a later version.
2. Right-click on a folder you own where you want to create an agent, and select Properties.
3. Click on the Agents tab, which should be visible if you have the correct permissions for the folder and the Server Scripting add-in is installed. At the bottom of the Agents tab is a drop-down menu from which you select the Exchange server where you want to run your agents. Make sure that the correct server is selected in the drop-down list. Only servers with the Event Service installed will appear in this list. Note that all agents in the folder will run on that Event Service computer. You cannot run agents that are in the same folder on different Event Service computers. Your agents don't have to run on the same server as the folders they monitor.
4. To create a new agent, click the New button. The New Agent dialog box appears, as shown in Figure 13-6.

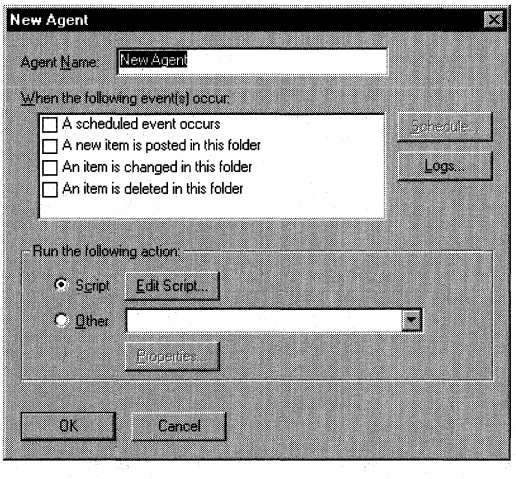


Figure 13-6. *The New Agent dialog box. This dialog box allows you to pick the events that your agent will fire on.*

5. Type in a name for your agent.
6. Select the events that your agent will handle. To create a timer-based agent, select the first option, named A Scheduled Event Occurs, and click the Schedule button. The Scheduled Event dialog box appears, as shown in Figure 13-7.

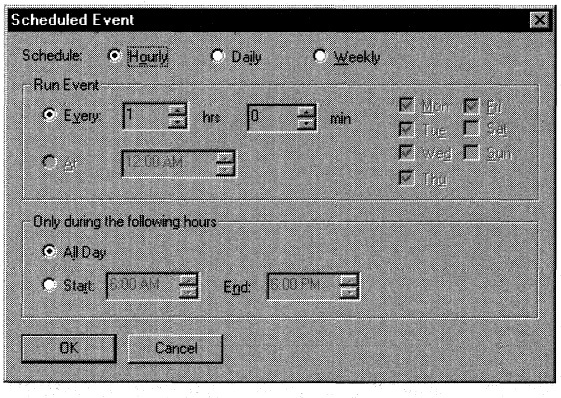


Figure 13-7. *The Scheduled Event dialog box allows you to configure scheduled events for your agents.*

7. In the Scheduled Event dialog box, you can specify that the event should fire hourly, daily, or weekly, as well as limit the hours when the event fires. Limiting the hours when your agent runs is useful if you want to make the agent run when the server is least taxed, usually late at night or early in the morning.
8. After specifying which events the agent should handle, select which action will occur for those events. To do this, select either the Script option or the Other option in the bottom half of the New Agent dialog box. The Other option enables the drop-down list of custom event handlers installed on the server. For example, if you have the custom event handler Exchange Routing Objects installed, the Microsoft Routing Engine Agent will appear in the list.
9. To create a new script, select the Script option, and click the Edit Script button. Windows Notepad will automatically display an ASP file that contains the event procedures to handle the four events supported in the Event Service, as shown in Figure 13-8.

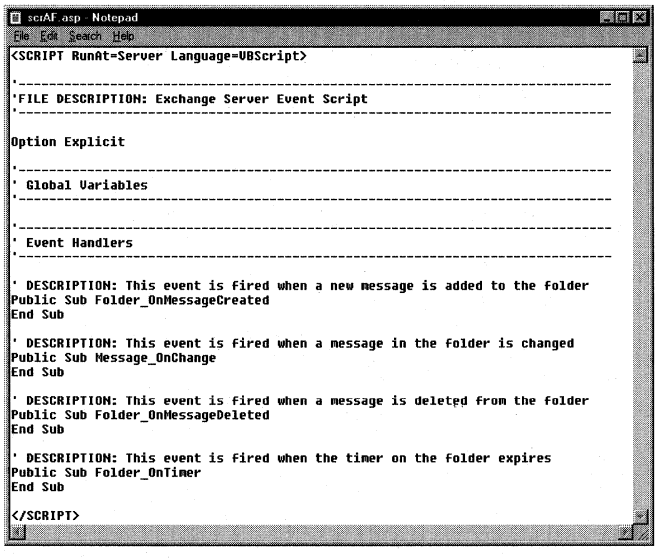


Figure 13-8. *A new script shown in Notepad. Notice how the new agent automatically contains four event procedures to handle the four events supported by the Event Service.*

Supported Event Types

As mentioned earlier, the Event Service supports four different event types: message create, change, delete, and timer. To write a script that implements your functionality for these events, you must modify these four default stub subroutines:

- *Folder_OnMessageCreated*
- *Message_OnChanged*
- *Folder_OnMessageDeleted*
- *Folder_OnTimer*

When you write an Event Scripting Agent, you can also use JavaScript. In JavaScript, these would be the four functions:

- *Folder::OnMessageCreated*
- *Message::OnChanged*
- *Folder::OnMessageDeleted*
- *Folder::OnTimer*

Intrinsic Objects for Scripts

CDO, which you learned about in Chapter 12, contains the intrinsic object model for your scripts. When you are writing agents, the Event Service passes you some objects and variables that you can use to quickly figure out what item triggered the event and in what folder the item is located. To help you access these items quickly as well as access other Exchange Server items, the Event Service also passes you a pre-logged-on CDO session so that you do not have to log on to the Exchange server yourself. The intrinsic objects passed to your script by the Event Service are discussed in the following sections.

EventDetails.Session

The EventDetails.Session object represents the pre-logged-on CDO session for your script. The Event Service decides which identity to use for logging on to your script by using the identity of the author who most recently saved the script. This is important to consider for two reasons. First, the functionality of your application might depend on access to specific items in the Exchange Server Information Store. If the identity of the most recent author does not have access to this information, your script will not work.

Second, any mail you send from your script will use the name of the pre-logged-on CDO session because the Event Service is logging in as this user. The sent messages will also be saved in the Sent Items folder of that user. For these reasons, consider creating unique identities for your agents, and log on as these users to save your script. For example, if you are creating an expense report application, you might want to create a user named Expense Report Administrator and log on to your Exchange server as that user. Then create and save your script using that identity. Any of the e-mail sent by the agent will appear to be from the Expense Report Administrator rather than from your personal account.

Since the CDO Session object is pre-logged-on, you can start accessing CDO objects directly from the EventDetails.Session object. It is a good idea in your script to assign the EventDetails.Session object to another variable for use throughout your script.

EventDetails.FolderID

The *EventDetails.FolderID* variable contains the entry identifier of the folder that the event took place in. By using this variable with the CDO *GetFolder* method, you can quickly retrieve the correct folder for the event. Again, it is a good idea to assign this variable to another variable in your script.

EventDetails.MessageID

The *EventDetails.MessageID* variable contains the entry identifier of the message that triggered the event. By using this variable with the CDO *GetMessage* method, you can quickly retrieve the exact message that the event corresponds to. Be aware, however, that timer events do not pass an *EventDetails.MessageID* variable because no message

triggers the event; rather, an elapsed amount of time triggers the event. Keep this in mind when creating scripts, because an error related to *EventDetails.MessageID* for a timer event can be hard to track down when debugging.

Instantiating Other COM Objects from Your Scripts

In addition to using the CDO object library in your scripts, you can call other COM components by using the *CreateObject* method in VBScript. These components can include server-based object libraries such as ADO for database access and ADSI for directory access. You can even instantiate your own COM components developed using Visual Basic or Visual C++. There are two primary requirements for custom COM components to be used with the Event Service:

- The components must not have any user interface elements. Because the Event Service is running on the Exchange server without an interactive user at the keyboard, the component can't, for example, display dialog boxes or error messages.
- The component must be programmed as an apartment-threaded component.

By remembering these two requirements, you can offload much of the work in your scripts to your COM components and include only the necessary script to instantiate your components. Furthermore, if you write COM components in Visual Basic or another tool, debugging will be much easier for your application because these tools provide richer debugging features than the Microsoft Script Debugger, as we will see shortly.

To send errors from your component to the event script, use the *Error.Raise* method in your component. For debugging purposes, use the arguments of the *Raise* method to pass back the correct error number as well as the source and description of the error.

If your components instantiate other remote COM components, make sure to configure Distributed Component Object Model (DCOM) correctly so that the Windows NT account the Event Service is running under can correctly instantiate them. You can modify the permissions for DCOM using the DCOM Configuration program (dcomcnfg.exe).

In your objects, you can also create custom COM components that use the features of Microsoft Transaction Server (MTS) to make the components more scalable and robust. For example, components created with MTS can handle process isolation, security identity, resource pooling, and distributed transaction coordination. Your script can instantiate MTS objects using *CreateObject* in the same way it instantiates other types of objects.

ERROR TRAPPING AND LOGGING

If you program like me, your applications probably don't work correctly the first time you run them. To help us be more successful, Microsoft has created some error-trapping tools, logging features, and applications that work with the Exchange Event Service.

Microsoft Script Debugger

Your first line of defense against the bugs that always somehow find their way into programs is the Microsoft Script Debugger. We looked at the Script Debugger in the context of debugging Outlook scripts in Chapter 5. The same Script Debugger can be used to debug Exchange event scripts as well. To force your script to hit a breakpoint, use the Stop statement in VBScript and the Debugger statement in JavaScript.

Because the Script Debugger does not support remote debugging (at the time of this writing), you must run the debugger on the machine where the script is executing. For the Event Scripting Agent, this machine is the Exchange server where the script is currently executing. Figure 13-9 shows the Microsoft Script Debugger debugging a script.

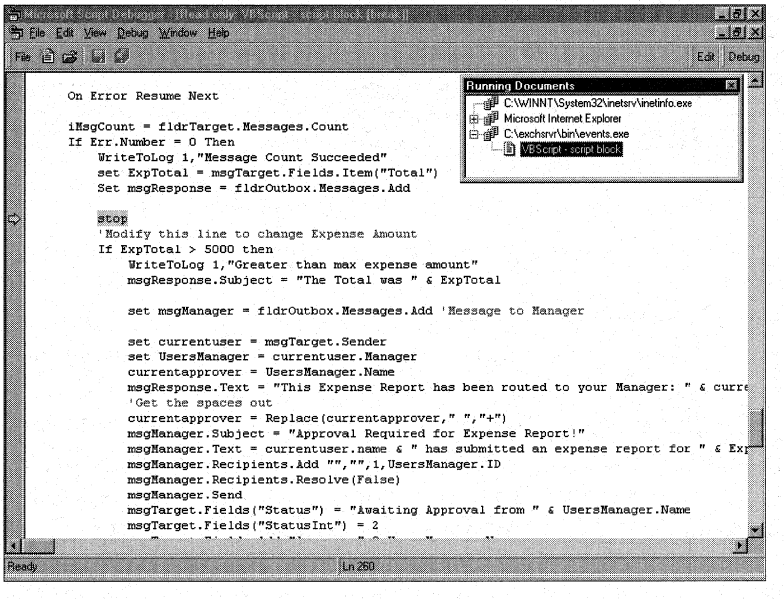


Figure 13-9. *The Microsoft Script Debugger allows you to step through your scripts running on the Exchange server.*

Script.Response and Logging

The Script Debugger is an invaluable tool when developing your Event Scripting applications. However, once you deploy your solutions in your company, you probably do not want to run instances of the Script Debugger on your production servers. This is where your second line of defense comes in: you can call the *Script.Response* method in your scripts to write strings of text to the log files associated with your agents. Figure 13-10 shows an example of an agent log file.

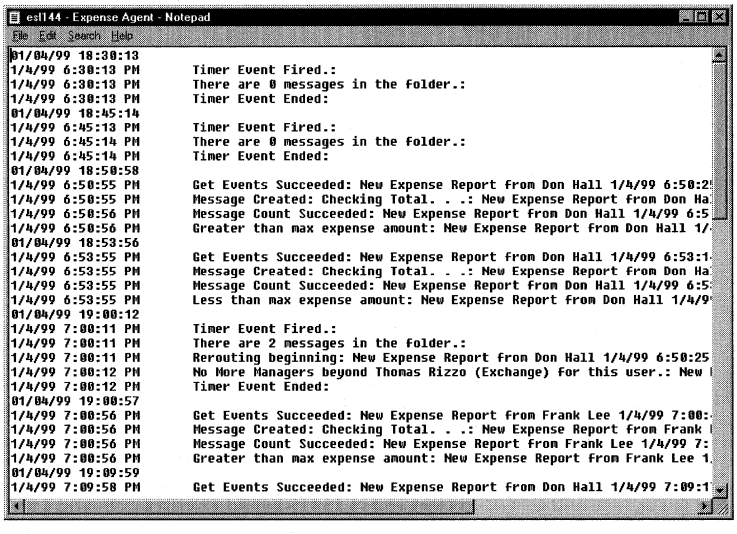


Figure 13-10. *An agent log file in Notepad. Each agent has an associated log file in which you can write your own status or error-logging information.*

You can access the log file for your agent remotely in Outlook by accessing the Agents tab for the folder, selecting the agent, clicking the Edit button, and then clicking the Logs button. By default, your agents will log only errors that occur in your scripts, but you can extend their functionality by using the *Script.Response* method to help you debug problems or track the status of your scripts.

The *Script.Response* method takes a string argument, which allows you to write information into the agent logs. As mentioned earlier, these log files, by default, are 32 KB in size, and older events are written over as necessary when this size limit is exceeded. If you make multiple calls to the *Script.Response* method, the code will write only the most recent string passed to the method into the log. To avoid losing strings when making multiple calls to *Script.Response*, prefix the previous response string with a new response string. The Expense Report sample application shown later in this chapter demonstrates how to use this technique in your applications.

The Windows NT Event Log

One other line of defense that you have in debugging your applications is the Windows NT Event Log. When you set Logging Level in the registry to the maximum value (5) for the Event Scripting Agent, the Windows NT Event Log provides not only error information gleaned from your scripts but also general information about the status of the Event Service and what notifications it has received from the Exchange server. When you use the *Script.Response* method described earlier to track errors and status information for your scripts, the information will be added to the description field in the Event Detail dialog box for an Event Service entry in the Application Log, as shown in Figure 13-11. (To view these entries in Event Viewer, be sure to select Application from the Log menu.) This type of information can make it easier for you to track down bugs or failures in your released application.

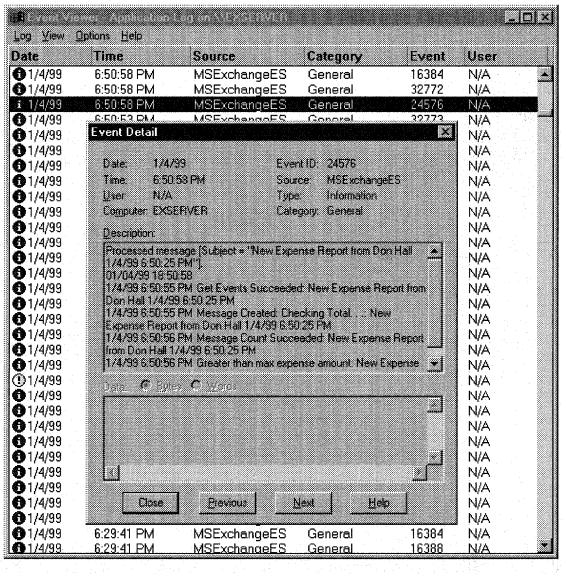


Figure 13-11. *The event details for the Expense Report application. Since the* Script.Response *method was used, the Description field has detailed information.*

EXPENSE REPORT APPLICATION

Although the Event Service could be used for non–Exchange Server–related applications, most developers use the Event Service to automate administrative tasks such as modifying user directory information and replicating data from Microsoft SQL Server into Exchange or vice versa, or to write workflow applications. In this section, we will look at a simple workflow application—an Expense Report application—that uses the Event Service and some custom code written using VBScript.

When most developers think of building a workflow application, the first type that pops into their minds is an expense reporting application, because most expense reporting applications require some type of report status tracking and approval routing, as well as a system of escalation for nonapproved reports. To help you better understand how to write applications using the Exchange Event Service, let's look at a sample expense report application using the Event Service technology.

Setting Up the Expense Report Application

Before you can install the application, you must have a Windows NT 4.0 Server and a client with certain software installed. Table 13-1 describes the installation requirements for the application.

Minimum Software Requirements	*Installation Notes*
Exchange Server 5.5 Service Pack1 or higher with Outlook Web Access	Service Pack 3 is recommended.
Internet Information Server 3.0 or higher with Active Server Pages	Internet Information Services 4.0 is recommended.
CDO library (cdo.dll) and CDO Rendering library (cdohtml.dll)	Exchange Server 5.5 SP1 installs CDO library 1.21 and CDO Rendering library 1.21. Outlook installs CDO library 1.21.
For the client: A Web browser and Outlook 98	For the Web browser, Microsoft Internet Explorer is recommended. You can run the client software on the same machine or on a separate machine.

Table 13-1. *Installation requirements for the Expense Report application.*

To install the Expense Report application, copy the Expense Report folder from the companion CD to the Web server where you want to run the application. Start the IIS administration program. Create a virtual directory that points to the location where you copied the expense report files, and name the virtual directory *expense*. Enable the Execute permissions option for the virtual directory. You will be able to use the following URL to access your Expense Report application: *http://*yourservername*/expense*.

Open the Exchange Administrator program. Open the Properties dialog box for the Folders\System Folders\Events Root\EventConfig_*servername* folder. Click the Client Permissions button, add a user who will administer the Expense Reports folder, and grant the user Author permissions. Click OK twice. Start the Registry Editor on

your server, and then open the key named HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\MSExchangeES\Parameters. Set the Logging Level DWORD to *5* to log the maximum amount of information.

> **NOTE** Be sure to set Logging Level to *0* when you are finished testing the Expense Report application. If you do not, your Application log will be quickly filled up with MSExchangeES logging entries.

Launch Outlook using the user you selected earlier to administer the Expense Reports folder. Create a new public folder named Expense Reports under All Public Folders. Right-click on the Expense Reports folder, and select Properties. On the Agents tab, click the New button. Type *Expense Agent* as the Agent Name. Check the A Scheduled Event Occurs check box and the A New Item Is Posted In This Folder check box. Click the Schedule button, set a 15-minute interval, and click OK.

In the Expense Agent dialog box, click Edit Script to display the event scripting starter code in Notepad. On the companion CD, locate the file named ExpenseAgentScript.txt in the expense report files. Open ExpenseAgentScript.txt, copy all of the code, and paste it into the starter code in Notepad, replacing the existing code. Perform a search in the code, and replace the three instances of the text *localhost* with the name of your Web server. Save and close Notepad. At this point, the Expense Agent dialog box should look like Figure 13-12. Click OK twice to return to Outlook.

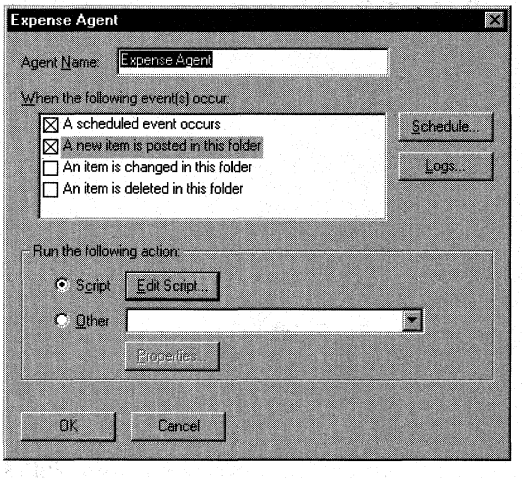


Figure 13-12. *The configured Expense Agent dialog box.*

Open the Exchange Administrator program, and open the Properties dialog box for the Expense Reports public folder. Click on the Advanced tab, and uncheck the Hide From Address Book check box. Click OK. You can now access the Expense Report application using the URL *http://yourservername/expense*.

NOTE Included with the Expense Report files on the companion CD is a .pst file named Expense Reports.pst. This file shows some sample expense reports. To see these samples, clear the read-only flag on Expense Reports.pst and open it in Outlook.

Functionality of the Expense Report Application

After entering a valid mailbox, the main page of the Expense Report application is displayed, as shown in Figure 13-13. From the main page, the user can click the Submit A New Expense Report link to enter and submit an expense report, as shown in Figure 13-14. As you can see, users submit expense reports in this application by using a simple Web page, but you can easily modify an Outlook form or a Microsoft Excel document to implement the same functionality as the Web page.

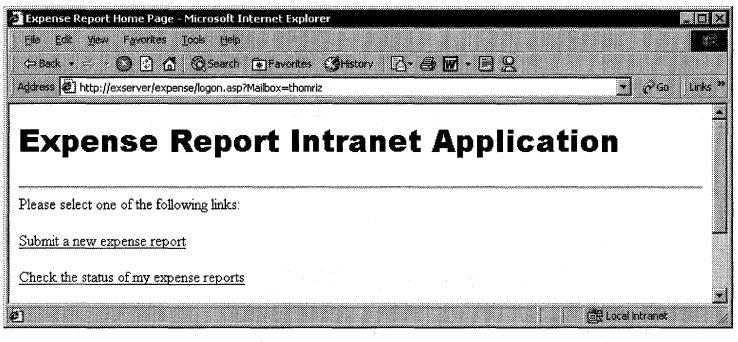


Figure 13-13. *The main page of the Expense Report application.*

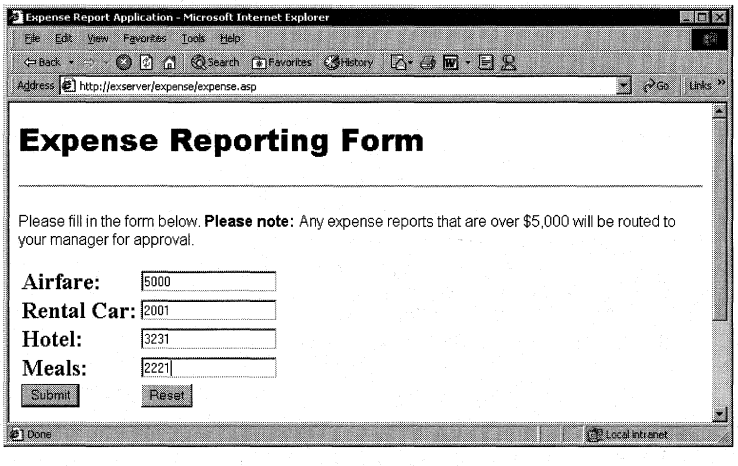


Figure 13-14. *The page used to enter and submit expense reports.*

After the user submits an expense report from the Web page, the report is e-mailed to the Expense Reports public folder that contains the agent, named Expense Agent. This agent fires on two of the four supported events. In the Expense Report application, I assume that expense reports are not normally changed while in process and are not deleted once submitted. Thus, the agent fires for these two events: when a new expense item is created in the folder and when every 15 minutes pass (this is a timer event).

The Expense Agent receives the new expense report and calls the *Folder_OnMessageCreated* subroutine in its associated VBScript file. This subroutine checks the amount of the expense report, and if the amount is over a specific limit—in this case, $5,000—the agent looks up the manager of the user in the directory and sends an e-mail to the manager with a link to the expense report, as shown in Figure 13-15. If the amount is under the limit, the agent automatically approves the expense report and routes it for payment.

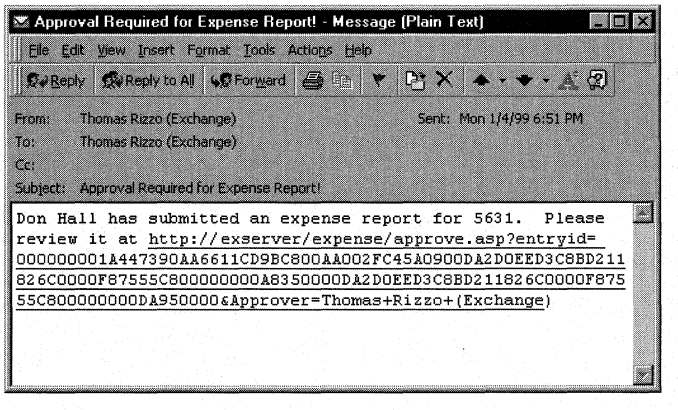


Figure 13-15. *E-mail, with a link to an expense report sent to a manager by the Expense Agent, requesting approval of the expense.*

Now we all know that people sometimes get bogged down in their e-mail and do not always quickly respond to requests for expense report approvals. To help facilitate the responsiveness of managers who need to approve expense reports, the agent fires on a 15-minute timer event. Every 15 minutes, the agent calls the *Folder_OnTimer* subroutine, which checks the current status of all expense reports in the folder. If the subroutine finds an expense report that has not been approved yet and that has been sitting for more than 15 minutes, the agent automatically looks up the manager of the current person who is supposed to approve the expense report and routes the report to that person for approval. This process will continue every 15 minutes until either the expense report is approved or until the agent runs out of

managers to reroute the report to. Each rerouted report sends a polite message to the manager who was supposed to approve the report and updates the user on the status of the routing.

If a report is rerouted to other managers, any manager in the route—from the first manager to the top person in the organization—can approve or reject the expense report at any time. This flexibility allows anyone with authority that sees the report to approve it, not just the current manager the report is routed to.

Throughout the entire application, users can go to a Web page to track the status of their expense reports. As you can see in Figure 13-16, I have used familiar traffic icons for expense report status. A stop sign means the expense report was rejected, a yellow light means that it is currently waiting for approval, and a green light means that the expense report was approved. Also included is text that describes the current report status, such as whether the report is waiting approval, the name of the person in the management chain currently reviewing the report, whether the report was rejected or approved, and who rejected or approved it.

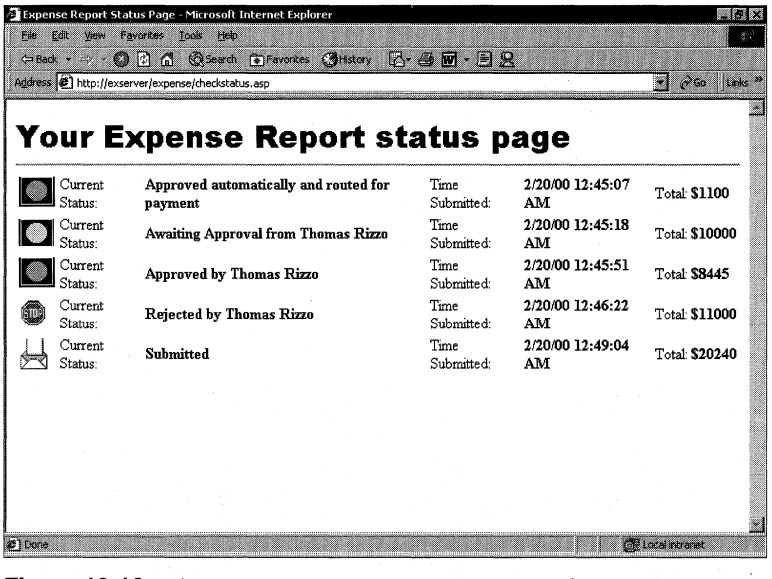


Figure 13-16. *The Expense Report Status Page. From this page, users can check the status of their expense reports as well as find out who is currently reviewing the report.*

Managers see a slightly different view of the information in the application's main screen. By using a CDO MessageFilter object, the Web page figures out whether managers have any reports waiting for approval in the Expense Reports public folder. If there are reports awaiting approval, the page indicates how many, as shown in Figure 13-17.

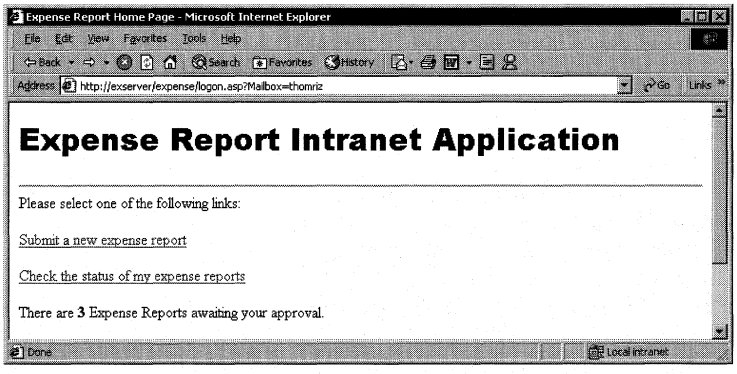


Figure 13-17. *The Web page for managers who have expense reports pending approval. This Web page uses a CDO MessageFilter object to quickly find pending expense reports.*

Expense report status information such as the current approver's name; expense amounts for items such as travel, hotel, and rental car; and which stage of approval the report is in (*1* for Submitted, *2* for Awaiting Approval, *3* for Rejected, *4* for Approved) are all stored with the individual message as custom properties. This means that the agent can update the status of the expense report using only CDO methods, which you will see when we examine the Expense Agent script.

Expense Agent Script

Now that you understand some of the functionality of the Expense Report application, let's look at the code that implements it. The two main pieces of the application are the Web pages that constitute part of the interface and the agent that implements the business logic. We will look at some of the CDO code behind the Web pages for the application, because these pages show you how to use some of the CDO objects in a way that was not demonstrated in Chapter 12.

The Expense Agent, as mentioned earlier, fires on only two events. The script for the agent includes two helper functions, *GetEventDetails* and *WriteToLog*.

GetEventDetails Function

The first helper function is the *GetEventDetails* function, shown in the following code:

```
'DESCRIPTION: Get the details of the event that fired
Private Sub GetEventDetails
   On Error Resume Next
   Dim oStores
   Dim Temp
   Dim idTargetFolder
   Dim idTargetMessage
```

(continued)

```
    idTargetFolder = EventDetails.FolderID
    idTargetMessage = EventDetails.MessageID
    'Some of the above might not exist
    Err.Clear
    Set AMSession = EventDetails.Session
    If Err.Number = 0 Then
       'We're going to send a message, so let's get the
       'Outbox here
       Set fldrOutbox = AMSession.Outbox
       If Err.Number = 0 Then
          Set oStores = AMSession.InfoStores
          If Err.Number = 0 Then
             Set Temp = oStores.Item(1).RootFolder
             Set Temp = oStores.Item(2).RootFolder

             Set fldrTarget = AMSession.GetFolder( _
                idTargetFolder, Null )
             If Err.Number = 0 Then
                Set msgTarget = AMSession.GetMessage( _
                   idTargetMessage, Null )
                If Not Err.Number = 0 Then
                   WriteToLog 0,"Session.GetMessage Failed: " & _
                      Err.Description
                End If
             Else
                WriteToLog 0,"Session.GetFolder Failed: " & _
                   Err.Description
             End If
          Else
             WriteToLog 0,"Session.InfoStores Failed: " & _
                Err.Description
          End If
       Else
          WriteToLog 0,"Outbox.Messages Failed: " & _
             Err.Description
       End If
    Else
       WriteToLog 0,"EventDetails.Session Failed: " & Err.Description
    End If
End Sub
```

The *GetEventDetails* function pulls the intrinsic objects and variables passed to the script and assigns them to other variables. The script then proceeds to get the Outbox of the pre-logged-on CDO user and retrieve both the folder and the message corresponding to the event. If any of these calls fail, the *GetEventsDetails* function calls the second helper function, *WriteToLog*.

WriteToLog Function

The *WriteToLog* function allows you to record custom messages in the agent log file. Earlier in this chapter, we discussed the *Script.Response* method in the context of helping you debug your agents, and you learned that if you make multiple calls to this method, you have to keep building your string by passing it previous strings. The *WriteToLog* function implements this functionality, and it takes two parameters that allow you to customize how events are logged. The first parameter, when set to *1*, records the name of the message when recording your event in the log. The second parameter is the string you want to place in the log. As you will see with the Expense Agent, the *WriteToLog* function is used heavily to insert status messages for the agent in the log. Here is the code for the *WriteToLog* function:

```
Private Sub WriteToLog(boolRecordName,strMessage)
   Dim strResponse
   strResponse = Now & vbTab & strMessage & ":"
   if boolRecordName = 1 then
      strResponse = strResponse & " " & msgTarget.Subject
   else
      strResponse = strResponse & " "
   end if
   Script.Response = Script.Response & vbNewLine & strResponse
End Sub
```

Folder_OnMessageCreated Function

The *Folder_OnMessageCreated* function is called when a new expense report is placed in the folder. When this function is called, it checks the expense total of the new expense reports in the folder by using the Fields collection on the item and then looking up the Total field.

If the expense total is greater than a certain amount, the script looks up the manager of the user issuing the report by using the CDO AddressEntry object. The script sends this manager a message containing a hyperlink to the current expense report. Then the script updates the message's status fields to reflect that the report has been routed to a new person. Finally, the agent e-mails a status update to the user and indicates to whom the report was routed.

If the expense total is less than $5,000, the agent automatically approves the expense report and updates its status. Although the application does not perform any tasks beyond sending an e-mail and updating the status, you could, in your agent, change this function to send an e-mail to the accounting department or update a database to transfer the funds into the user's expense account. The *Folder_OnMessage-Created* code is shown here:

```
'DESCRIPTION: This event is fired when a new message is added to
'the folder
Public Sub Folder_OnMessageCreated
```

(continued)

```
    On Error Resume Next
    GetEventDetails
    If Err.Number = 0 Then
       WriteToLog 1,"Get Events Succeeded"
       WriteToLog 1,"Message Created: Checking Total. . ."
       CheckTotal
    Else
       WriteToLog 0,"GetEventDetails Failed"
    End If
End Sub

'DESCRIPTION: Check the total of the expense report, and if it is
'less than a specific amount, automatically approve the expense
'report
Private Sub CheckTotal
    Dim msgResponse
    Dim iMsgCount
    Dim msgManager
    Dim UsersManager
    Dim currentuser
    Dim currentapprover

    On Error Resume Next
    iMsgCount = fldrTarget.Messages.Count
    If Err.Number = 0 Then
       WriteToLog 1,"Message Count Succeeded"
       set ExpTotal = msgTarget.Fields.Item("Total")
       Set msgResponse = fldrOutbox.Messages.Add
       'Modify this line to change Expense Amount
       If ExpTotal > 5000 then
          WriteToLog 1,"Greater than max expense amount"
          msgResponse.Subject = "The Total was " & ExpTotal
          'Message to Manager
          set msgManager = fldrOutbox.Messages.Add
          set currentuser = msgTarget.Sender
          set UsersManager = currentuser.Manager
          currentapprover = UsersManager.Name
          msgResponse.Text = "This Expense Report has been " & _
             "routed to your Manager: " & currentapprover
          'Get the spaces out
          currentapprover = Replace(currentapprover," ","+")
          msgManager.Subject = "Approval Required for " & _
             "Expense Report!"
          msgManager.Text = currentuser.name & _
          " has submitted an expense report for " & ExpTotal & _
          ".  Please review it at http://localhost/expense/" & _
          "approve.asp?entryid=" & msgTarget.ID & "&Approver=" & _
          CurrentApprover
```

```
            msgManager.Recipients.Add "","",1,UsersManager.ID
            msgManager.Recipients.Resolve(False)
            msgManager.Send
            msgTarget.Fields("Status") = _
               "Awaiting Approval from " & UsersManager.Name
            msgTarget.Fields("StatusInt") = 2
            msgTarget.Fields.Add "Approver",8,UsersManager.Name
            msgTarget.Update
         Else 'Expense Report <= Max Amount
            WriteToLog 1,"Less than max expense amount"
            msgResponse.Subject = _
               "This Expense Report has been Approved"
            msgResponse.Text = "Your expense report for " & _
               ExpTotal & " has been automatically approved. " & _
               "Funds are being transferred!"
            msgTarget.Fields("Status") = _
               "Approved automatically and routed for payment"
            msgTarget.Fields("StatusInt") = 3
            msgTarget.Update
         End If
         If Err.Number = 0 Then
            msgResponse.Recipients.Add "", "", 1, _
               msgTarget.Sender.ID
            If Err.Number = 0 Then
               msgResponse.Recipients.Resolve(False)
               If msgResponse.Recipients.Resolved = True Then
                  msgResponse.Send
                  If Not Err.Number = 0 Then
                     WriteToLog 0,"Message.Send Failed: " & _
                        Err.Description
                  End If
               Else
                  WriteToLog 0,"Recipients.Resolve Failed: " & _
                     Err.Description
               End If
            Else
               WriteToLog 0,"Recipients.Add Failed: " & _
                  Err.Description
            End If
         Else
            WriteToLog 0,"Messages.Add Failed: " & _
               Err.Description
         End If
      Else
         WriteToLog 0,"Messages.Count Failed: " & Err.Description
      End If
   End Sub
```

Folder_OnTimer Function

After 15 minutes, the *Folder_OnTimer* function is called by the agent to check the status of folder items. If any have the value *2*, which indicates that the item is waiting for approval, the script checks the time the item was sent into the folder (as opposed to the current time) by using the VBScript *DateDiff* function. The *DateDiff* function returns the difference between the two dates in numbers of seconds. Once this value is returned, the script checks to see whether it is greater than 400 seconds. (I picked an arbitrary number which is less than 900 seconds, or 15 minutes. In a completed application, you will probably want to give managers more than 15 minutes to approve expense reports before escalating them.)

If the report has been sitting for more than the specified interval, the script looks up the manager of the current approver by using the AddressEntry object in CDO. If this manager has no manager above her, the script sends a friendly reminder to the current approver explaining that an expense report is awaiting approval. The script also informs the user that there are no other managers to route the report to.

If there is a manager above the current approver, the script forwards the report to this manager and informs the user and the current approver that the report has been forwarded to a new manager. The script then updates the report status to reflect the change in state.

Notice in the following code that the script does not try to retrieve the *EventDetails.MessageID* variable because the variable does not exist for timer events. You will receive an error if you attempt to retrieve this variable in your implementation for a timer event.

```
'DESCRIPTION: This event is fired when the timer on the folder
'expires
Public Sub Folder_OnTimer
Dim oMessages
Dim oMessage
Dim Status
Dim currentdate
Dim elapsed
Dim timesent
Dim CurrentApprover
Dim NextApprover
Dim msgResponse
Dim objonerecip
Dim myaddentry
Dim currentuser
Dim msgNewApprover
Dim DestFolder
Dim idTargetFolder
Dim oStores
Dim Temp
```

```
Dim i
Dim StatusInt

'Since timer events do not return a specific message, all the calls to
'WriteToLog must not try to record the message name unless
'the variable msgTarget is explicitly set

On Error Resume Next
   WriteToLog 0,"Timer Event Fired."
   'Set variables using event details
   idTargetFolder = EventDetails.FolderID
   'Clear errors
   Err.Clear
   Set AMSession = EventDetails.Session
   fldrOutbox = AMSession.Outbox
   If Err.Number = 0 Then
      Set oStores = AMSession.InfoStores
      If Err.Number = 0 Then
         Set Temp = oStores.Item(1).RootFolder
         Set Temp = oStores.Item(2).RootFolder
         Set fldrTarget = AMSession.GetFolder( idTargetFolder, _
            Null )
      end if
   end if

   'Need to check all the messages in the folder to see if they
   'are over the 15-minute limit and are awaiting approval
   set oMessages = fldrTarget.Messages
   WriteToLog 0,"There are " & oMessages.Count & _
      " messages in the folder."

   for i = 1 to oMessages.Count
      'Retrieve the message
      set oMessage = oMessages.Item(i)
      'Check the time and status
      StatusInt = oMessage.Fields("StatusInt")
      if StatusInt = 2 then 'Got a live one
         'Figure out how long it has been sitting
         timesent = oMessage.TimeSent
         currentdate = now()
         elapsed = datediff("s", timesent,currentdate)
         if elapsed > 400 then 'been sitting for a while
            'Set another variable to the current message
            set msgTarget = oMessage
            WriteToLog 1,"Rerouting beginning"
            set ExpTotal = oMessage.Fields("Total")
            'Reroute the message
```

(continued)

```
set CurrentApprover = oMessage.Fields("Approver")
set msgResponse = AMSession.Outbox.Messages.Add
'Create the recipient
Set objonerecip = msgResponse.Recipients.Add
objonerecip.Name = CurrentApprover
'Resolve the name against the Exchange directory
objonerecip.Resolve
'Get the address entry so we can pull out
   'template information
Set myaddentry = objonerecip.AddressEntry
'Get the manager from the address entry
set NextApprover = myaddentry.Manager
if NextApprover = Empty then
'We don't have a manager!
   'Send a message to the current user
   set currentuser = oMessage.Sender
   msgResponse.Subject = _
      "No more manager to route to"
   msgResponse.Text = currentuser.name & _
   " has submitted an expense report for " & _
   ExpTotal & _
   ".  There are no other managers to route to!"
   msgResponse.Recipients.Add "", "", 1, _
      oMessage.Sender.ID
   msgResponse.Send
   'Resend a message to the current approver
   Set msgResendtoApprover = _
      AMSession.Outbox.Messages.Add
   CurrentApproverName = Replace( _
      CurrentApprover," ","+")
   msgResendtoApprover.Subject = _
      "Repeat notice for Approval of an " & _
      "Expense Report!"
   'Change the following location to be
   'your Web location
   msgResendtoApprover.Text =
   currentuser.name & " has submitted an " & _
   "expense report for " & ExpTotal & _
   ".  Please review it at http://localhost/" & _
   "expense/approve.asp?entryid=" & msgTarget.ID & _
   "&Approver=" & CurrentApproverName
   'Create the recipient
   set oRecip = msgResendtoApprover.Recipients.Add
   oRecip.Name = CurrentApprover
   oRecip.Resolve
   msgResendtoApprover.Send

   WriteToLog 1,"No More Managers beyond " & _
      CurrentApprover & " for this user."
```

```
            else
               NextApproverName = NextApprover.Name
               'Got the next approver. Send a message to
               'previous approver and user and reroute.
               set currentuser = oMessage.Sender
               msgResponse.Subject = "An Expense Report" & _
               " has been rerouted"
               msgResponse.Text = currentuser.name & _
               " has submitted an expense report for " & _
               ExpTotal & ".  It was rerouted because the " & _
               "15 minute approval time limit has expired. " & _
               " It is now routed to " & NextApproverName
               msgResponse.Recipients.Add "", "", 1, _
                  oMessage.Sender.ID
               msgResponse.Send
               if err.number = 0 then
                  WriteToLog 1,"Successfully rerouted"
               end if
               'Now change the status and reroute to
               'new person
               oMessage.Fields("Status") = _
                  "Rerouted and awaiting Approval from " & _
                  NextApproverName
               oMessage.Fields("Approver") = NextApproverName
               oMessage.Update
               'Now send a message
               Set msgNewApprover = _
                  AMSession.Outbox.Messages.Add
               'Create the recipient.
               'Get the spaces out.
               NextApproverName = Replace( _
                  NextApproverName," ","+")
               msgNewApprover.Subject = _
               "Approval Required for Rerouted Expense Report!"
               msgNewApprover.Text = currentuser.name & _
               " has submitted an expense report for " & _
               ExpTotal & ".  Please review it at http://" & _
               localhost/expense/approve.asp?entryid=" & _
               msgTarget.ID & "&Approver=" & NextApproverName
               msgNewApprover.Recipients.Add "","",1, _
                  NextApprover.ID
               msgNewApprover.Recipients.Resolve(False)
               msgNewApprover.Send
            end if 'Manager!
         end if 'Elapsed
      end if 'Status
   next
   WriteToLog 0,"Timer Event Ended"
End Sub
```

CDO Code in the Application

The Expense Report application contains sections of CDO code that show how to use CDO objects not discussed in detail in Chapter 12, which covers CDO development. The most interesting section of the code is found in the file Logon.asp. The code in this script uses custom properties in a MessageFilter object to filter out all expense reports that have the current user as the current approver, as well as filter out only those expense reports with a status of *2*, which means that the expense report is waiting for approval. As you can see in the following code, to filter custom properties on an item, you must use the *Add* method of the Fields collection for the MessageFilter object. In the *Add* method, you need to specify the name of the custom property; the type, by using a Long constant; and the value that the property should use for the filter. Once you set these properties, the messages collection will contain only those items that meet your specified criteria.

```
<% 'Check to see whether any reports are waiting for this approver

set oMessages = objFolder.Messages
set oMsgFilter = oMessages.Filter
set oApprover = oMsgFilter.Fields.Add( _
   "Approver",8,AMSession.CurrentUser.Name)
set oStatus = oMsgFilter.Fields.Add("StatusInt",8,"2")
iMsgCount = oMessages.Count
if iMsgCount > 0 then
   response.write "<P>There are <B>" & iMsgCount & _
       "</B> Expense Reports awaiting your approval."
end if
%>
```

PROGRAMMATICALLY BINDING AGENTS

Now that you have learned how to create and program agents, you might be wondering how you can bypass the Agents tab in Outlook and programmatically install and bind your agents to events in Exchange Server. The Exchange Event Service provides an object library that allows you to create and delete agents and their respective bindings on your server. This object library makes it easier for you to pull out information about the agents in your system as well as install agents into multiple folders. The following section describes the object library provided for these services and discusses a sample application, named Agent Install, that uses this object library to allow you to programmatically create and delete agents on your Exchange server.

Exchange Event Service Configuration Library

The object library for the Event Service configuration is stored in a file named Esconf.dll. This file is usually installed in the C:\exchsrvr\bin directory on your server. When working with Visual Basic, you can add a reference to this type library either by searching for Esconf.dll or by finding the name Microsoft Exchange Event Service

Config 1.0 Type Library in the Available References list box. Once you add a reference to it, you can use the object browser to browse through the different objects in the library. Figure 13-18 shows the object hierarchy for the Exchange Event Service Configuration library.

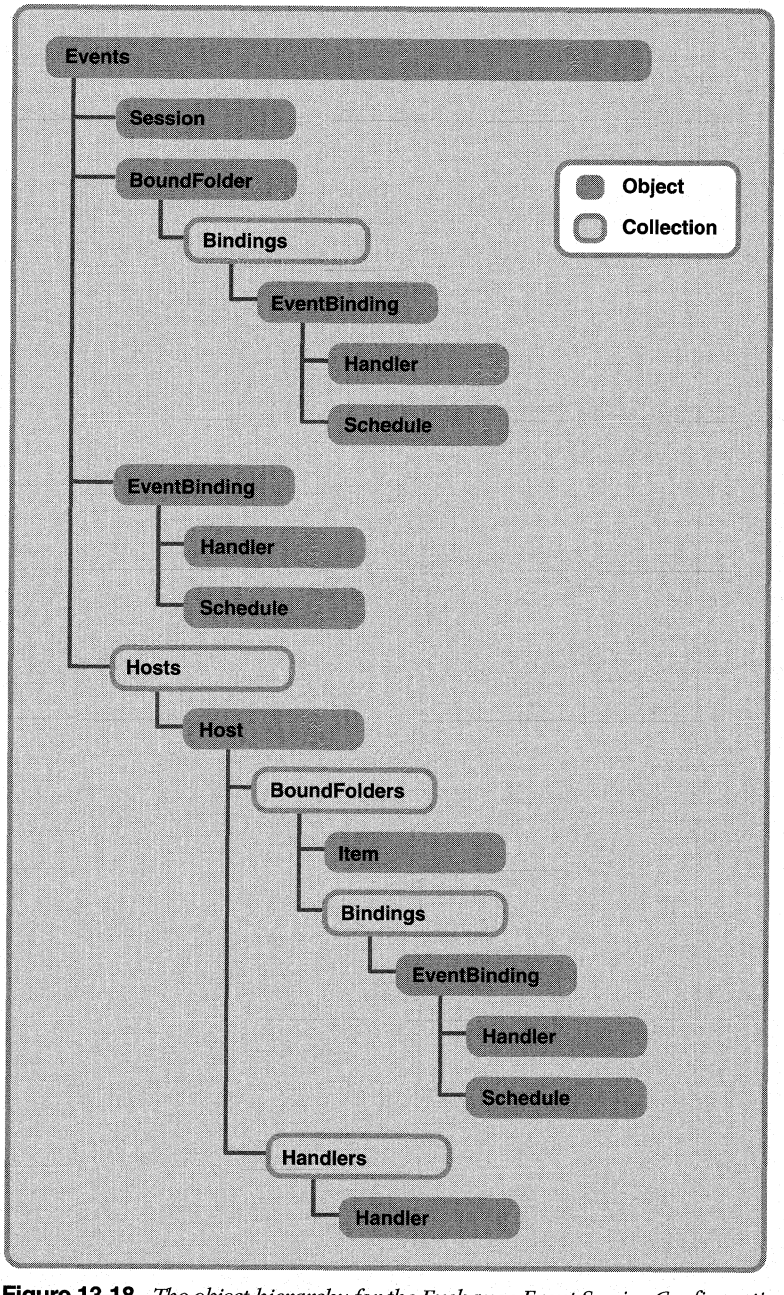


Figure 13-18. *The object hierarchy for the Exchange Event Service Configuration library.*

AGENT INSTALL APPLICATION

The easiest way to learn how to use the objects in the Event Service Configuration library is to look at a sample application that uses them. I created a Visual Basic program, named Agent Install, that allows you to select a folder on Exchange Server, see how many agents are installed in the folder, add or delete agents, and view the scripts of existing agents. (The Agent Install application is available on the companion CD in the Agent Install folder.) The main interface for the application is the tree view of Exchange folders, as shown in Figure 13-19. The interface is based on code from the Exchange Routing Wizard, a sample application included with Exchange Server 5.5 Service Pack 1 that uses Routing objects. (We will learn more about Routing objects in Chapter 14.) This wizard interface has been modified so that when you click on a folder, the application lists the number of agents contained in the folder as well as fills a list box with the names of all the agents. You can then add a new agent to the folder or delete one of the listed agents.

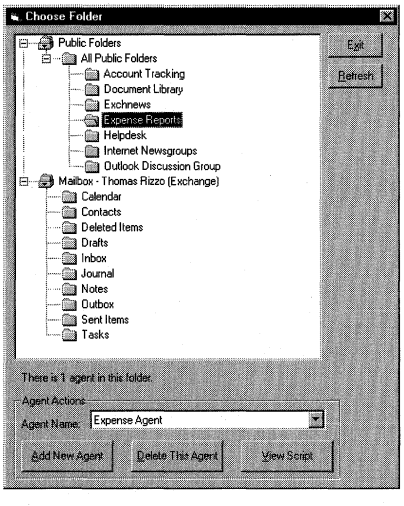


Figure 13-19. *The main interface for the Agent Install application. The tree view allows you to pick a folder that you want to perform actions on.*

If you select a folder that you own and click the Add New Agent button, a dialog box similar to Outlook's New Agent dialog box appears, as shown in Figure 13-20. The difference between the Agent Install New Agent dialog box and the one in Outlook is that the Agent Install version allows you to browse for the script you want to use in the agent. You can still select the events you want the agent to fire on as well as set the schedule for timer events.

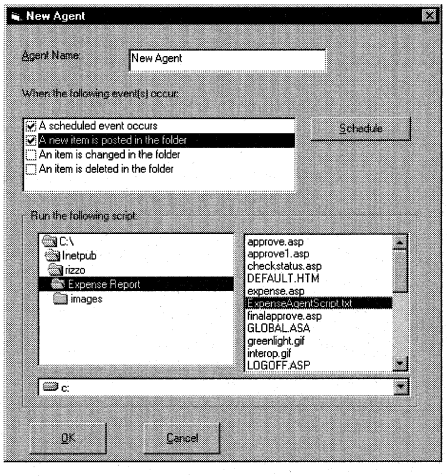


Figure 13-20. *The Agent Install New Agent dialog box. This dialog box allows you to browse for the script you want to install.*

When the user checks the A Scheduled Event Occurs check box, the Schedule button is enabled. The Scheduled Event dialog box, shown in Figure 13-21, mimics the Scheduled Event dialog box found in Outlook. From the Scheduled Event dialog box, you can change when the scheduled agent will run: hourly, daily, or weekly.

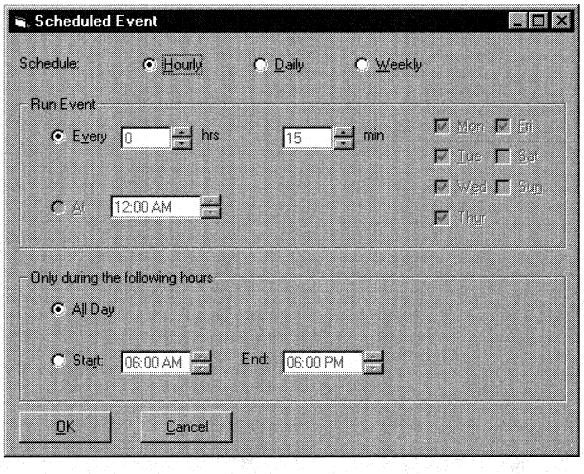


Figure 13-21. *The Scheduled Event dialog box for the Agent Install program. This dialog box mimics the Scheduled Event dialog box in Outlook.*

In the main interface, you can select to view the script for an agent by clicking the View Script button. This option launches Notepad on the local machine and displays the script. The Agent Install application doesn't allow you to modify the script, but the application could be modified to support this editing functionality.

Using the Exchange Event Service Configuration Library

The first task you need to accomplish when working with the Event Service Configuration library is to successfully create an instance of the Events object. To do this, you must call the *CreateObject* function and pass it the MSExchange.Events ProgID. The following line of code shows you how to do this:

```
Set oEvents = CreateObject("MSExchange.Events")
```

After creating an instance of the Events object, you need to set the *Session* property for the Events object to a valid CDO session. Normally, you would log on to the CDO session before attempting to create an instance of the Events object. The following code from the Agent Install program shows you how to perform this step:

```
If CDOClass.LogonStatus = True then
   oEvents.Session = CDOClass.Session
End if
```

Once you set the *Session* property, you can begin to work with the other objects in the library. The following sections step you through the most common tasks you will perform with the library by using the code from the Agent Install application.

Accessing Existing Agents

As shown earlier in Figure 13-19, you can programmatically access the agents contained in a folder on your Exchange server. The Event Service Configuration library provides a number of objects, methods, and properties to help you do this. When attempting to access existing agents, you first set an object variable to the folder containing the bindings for the Event Service by using the *BoundFolder* property. You pass two arguments to the *BoundFolder* property: the CDO Folder object for the folder you are interested in and a Boolean value set to *True*. After setting this bound folder variable, you need to get the actual bindings in the folder by using the *Bindings* property. Your new bindings variable has a *Count* property, which is useful when accessing existing agents because it tells you how many agents exist in the folder. You can retrieve all the names of the agents by using a For...Each construct in Visual Basic. The following code from the Agent Install application shows how the label and combo box on the main interface are initialized. Note that the variable *oFolder* is already set to the CDO folder selected by the user.

```
Set oBoundFolder = oEvents.BoundFolder(oFolder, True)
Set oBindings = oBoundFolder.Bindings
If oBindings.Count = 1 Then
   lblAgentCount.Caption = "There is " & oBindings.Count & _
      " agent in this folder."
ElseIf oBindings.Count > 1 Then
```

```
   lblAgentCount.Caption = "There are " & oBindings.Count & _
      " agents in this folder."
Else
   '0 agents
   lblAgentCount.Caption = "There are " & oBindings.Count & _
      " agents in this folder."
End If
comboAgents.Clear
If oBindings.Count > 0 Then
   For Each oBinding In oBindings
      comboAgents.AddItem CStr(oBinding.Name)
   Next
   comboAgents.ListIndex = 0
End If
```

Accessing the Scripts Contained in Agents

Once you have the agents (bindings) in a particular folder, you might want to access the script for those agents. Before showing you how to do this programmatically, however, I must first explain how the agents and their associated scripts are stored in the folder.

When you create a new agent in a folder and associate a script with that agent, the Event Service creates two hidden messages in the folder. The first has a message class *IPC.Microsoft.ICS.EventBinding*. As you would guess by its name, this message class contains the types of bindings you want the ICS interface to notify the agent of. The second hidden message has a message class of *IPC.Microsoft.EventBinding*. This hidden message class contains the script source in a special property (*&H7102001E*), so before you can even access the script for an agent, you must first retrieve the hidden message associated with the agent containing the script, and then you must pull out the value for this property from that message.

When you have a binding in a folder, you find out the unique ID of the script source message by using the *EntryID* property on the Binding object. The *EntryID* property lets you use the CDO *GetMessage* method to quickly retrieve the script source message in the folder.

The following code shows you how the Agent Install program retrieves the script for an agent by using the methods just described and then saves the script to a text file and opens it in Notepad. Note that the *oBinding* variable already refers to a valid agent in the folder.

```
If Not (oBinding Is Nothing) Then
   On Error GoTo Script_Err
   Set oMessage = oSession.Getmessage(oBinding.EntryID, Null)
   bstrEventScript = oMessage.Fields.item(PR_EVENT_SCRIPT)
   'Write the script to a temporary file with a unique name
```

(continued)

```
    tmpLocation = "c:\temp\"
    Randomize
    tmpFileName = "scr" & Int((99999 - 1 + 1) * Rnd + 1) & ".txt"
    tmpFullPath = tmpLocation & tmpFileName
    Open tmpFullPath For Output As #1
       Print #1, bstrEventScript
    Close #1
    'Notepad opens the temporary file
    retval = Shell("notepad.exe " & tmpFullPath, vbNormalFocus)
End If
```

Creating Agents Programmatically

Once you understand how to access agents, creating agents is a pretty straightforward process. The only challenge when creating agents is understanding what properties you need to set and what the values of these properties should be. To help you with the latter problem, the Agent Install application has a Visual Basic module called MSEventConstants that defines constants for all of the common values for the properties of your agents. Specifically, the module defines constants for the days of the week and the type of events the agent should fire on, and for what the event handler should be when the event is fired—either the scripting engine or the Exchange Routing Objects. The code for the MSEventConstants module is shown here:

```
Public Const MSMonday = 1
Public Const MSTuesday = 2
Public Const MSWednesday = 4
Public Const MSThursday = 8
Public Const MSFriday = 16
Public Const MSSaturday = 32
Public Const MSSunday = 64
Public Const PR_EVENT_SCRIPT = &H7102001E
Public Const MSAllDayStart = 0
Public Const MSAllDayEnd = 0.9999
Public Const MSHourlyAgent = 1
Public Const MSDailyAgent = 2
Public Const MSWeeklyAgent = 3
Public Const MSScheduledEvent = 1
Public Const MSNewItemEvent = 2
Public Const MSChangedItemEvent = 4
Public Const MSDeletedItemEvent = 8
Public Const MSAgentActive = True
Public Const MSAgentDisabled = False
Public Const MSScriptHandlerID = _
   "{69E54151-B371-11D0-BCD9-00AA00C1AB1C}"
Public Const MSRoutingObjectsHandlerID = _
   "{69E64151-B371-11D0-BCD9-00AA00C1AB1C}"
```

Once you have these constants, all you need to do to create an agent programmatically is set the properties on a new binding in the desired folder. By setting an object variable as the return type for the *Add* method on the Binding object, the object model will return to you a new binding in the folder. From this object, you can set all the required properties, in this order:

- *Name*. The *Name* property takes a string that specifies the name of your agent.
- *Active*. The *Active* property takes a signed integer value which, if set to *0*, specifies that the agent is disabled and will remain in the folder but will not fire on any events. Setting this property to *–1* means the agent is enabled and will fire on its specified events. By default, the Agent Install program sets this property to *–1* to make all new agents active.
- *EventMask*. The *EventMask* property takes an integer that specifies which events your agent should fire on, such as when a new item is added to the folder or when an item is deleted in the folder. If you want to fire on multiple events, such as when a new item is created or when an item is changed, you should add together the values of the constants MSNewItemEvent and MSChangedItemEvent, and place this new value in the *EventMask* property. You will see an example of this process in the code you'll look at a little later in this chapter.
- *HandlerClassID*. The *HandlerClassID* property takes a string that corresponds to the globally unique identifier (GUID) that the event handler calls when events fire on the binding. By default, the constants in the sample application include the script engine handler ID as well as the Routing Objects handler ID. If you create your own event handler, you will need to add your own GUID and specify it in this property.
- *Schedule*. The *Schedule* property is a Variant, and you must set an object reference to it. Once you have done this, you can modify the properties for the Schedule object. The main property you want to set is the *Type* property for the schedule. The *Type* property can take the constants MSHourlyAgent, MSDailyAgent, or MSWeeklyAgent. Your agent must be hourly, daily, or weekly—you cannot have an agent that is greater than one of these values.

  The next property you want to set in the Schedule object depends on what you specified for the *Type* property. For example, if you specified that your agent should fire a timer event hourly, then you need to set only the *Interval* property (which specifies, in minutes, the interval of time between the firing of timer events) and the start and end times for these

timer events to occur during the day. You use the *StartTime* and *EndTime* properties, respectively. If you specify that you want a daily agent, you need to specify only at what time each day you want the agent to fire. To specify this, you use the *At* property. Finally, if you specify a weekly agent, you need to set at what time you want your agent to fire during the day by using the *At* property, and you need to set which days of the week the timer event should fire on by using the *Days* property. To set the *Days* property, use the constants MSMonday through MSSunday, and add the values together to calculate the correct integer to place in this property.

After specifying these properties, you should call the *SaveChanges* method on your new Binding object to request that the object model create the corresponding hidden message for the script source. You can see the functionality we just examined implemented in the following code:

```
Private Sub SetAgentName()
   AgentName = txtAgentName.Text
End Sub

Private Sub SetAgentType()
   'Scroll through the events to fire on and set type
   Dim tmp
   tmp = 0
   If boolSchedule Then
      tmp = tmp + MSScheduledEvent
   End If
   If boolNewItem Then
      tmp = tmp + MSNewItemEvent
   End If
   If boolChangeItem Then
      tmp = tmp + MSChangedItemEvent
   End If
   If boolDeleteItem Then
      tmp = tmp + MSDeletedItemEvent
   End If
   AgentType = tmp
End Sub

'Create the agent!
SetAgentType
SetAgentName
Set oBinding = oBindings.Add
oBinding.Name = AgentName
oBinding.Active = MSAgentActive
oBinding.EventMask = AgentType
oBinding.HandlerClassID = MSScriptHandlerID
```

```
'Need to create a schedule, if set
If boolSchedule Then
   Set oSchedule = oBinding.Schedule
   oSchedule.Type = AgentScheduleType
   Select Case AgentScheduleType
      Case MSHourlyAgent
         oSchedule.Interval = AgentInterval
         oSchedule.StartTime = Format(AgentStartTime, "hh:mm AM/PM")
         oSchedule.EndTime = Format(AgentEndTime, "hh:mm AM/PM")
      Case MSDailyAgent
         oSchedule.At = AgentAtTime
      Case MSWeeklyAgent
         oSchedule.Days = AgentDaysofWeek
         oSchedule.At = AgentAtTime
      End Select
End If
'Save changes so message is created
oBinding.SaveChanges
```

After successfully saving the changes, you need to copy a script into the hidden message associated with the new agent. To do this, you must use the CDO *GetMessage* method and the *EntryID* property of your new Binding object. As you can see in the next snippet of code, the program tries to open the file selected by the user to read it, and then it tries to copy the file into the *PR_EVENT_SCRIPT* property in the hidden script source message. Notice, however, that a variable, *tmpInProcess*, is set to *True* (*1*) after the *SaveChanges* call on the Binding object. This is to notify the program that if the file cannot be correctly read—for example, when the file is a binary file—and an error occurs, the agent should be deleted from the folder because it is not a complete agent. In the error handler, you can see how the program calls the *Delete* method on the Bindings collection and passes in the object that corresponds to the half-completed Binding object.

If the script is read properly, you should call the CDO *Update* method on the script source message, the *SaveChanges* method on the Binding object, and the *SaveChanges* method on the BoundFolder object. If these calls succeed, you have programmatically created an agent that fires on events and has a script associated with it. If you do not call the *SaveChanges* method, your agent will not be saved if the Binding object goes out of scope. Calling *SaveChanges* is like calling the *Update* method in CDO—if you don't call *Update* after changing items, your changes will not be saved.

```
'Enter in script here
'Set tmpInProcess to 1 for bad files
tmpInProcess = 1
Set oMessage = oSession.Getmessage(oBinding.EntryID, Null)
tmpFileLocation = fileCurFile.Path & "\" & fileCurFile.FileName
```

(continued)

```
Open tmpFileLocation For Input As #1
bstrEventScript = Input$(LOF(1), #1)
Close #1
oMessage.Fields(PR_EVENT_SCRIPT) = bstrEventScript
oMessage.Update
oBinding.SaveChanges
oBoundFolder.SaveChanges
MsgBox "Agent Successfully Created.", vbInformation + vbOKOnly, _
   "Agent Created"


frmFolders.RefreshAgentCount
Unload Me
Exit Sub

cmdOK_Err:
   MsgBox "Error #" & Err.Number & vbLf & "Error Description: " & _
      Err.Description, vbOKOnly, "Error in cmdOK"
   Close #1
   If tmpInProcess = 1 Then
      'Find the half-created agent and delete it
      oBindings.Delete oBinding
      oBoundFolder.SaveChanges
   End If
   Exit Sub
End Sub
```

Disabling and Deleting Agents

In the section titled "Creating Agents Programmatically," you had a glimpse of how to disable and delete agents. To disable an agent, all you need to do is set the *Active* property on the Binding object to *0* and then call the *SaveChanges* method. To delete an agent, find the Binding object that corresponds to the agent you want to delete, and then call the *Delete* method on the Bindings collection and pass the Binding object to *Delete*. Call the *SaveChanges* method to save the changes.

Agent Hosts

Although not used in the Agent Install application, you can enumerate the Exchange Server hosts capable of running agents. The Event Service Configuration library offers a Hosts collection, which provides you with a *Count* property for the number of available hosts and an *Item* property that will return a specific Host object. The following code fragment shows how you can print out the names of all the available hosts in your system:

```
Set oEvents = CreateObject("MSExchange.Events")
oEvents.Session = oSession    'Assumes valid CDO Session
```

```
Set oHosts = oEvents.Hosts
Msgbox "Count: " & oHosts.Count
For each oHost in oHosts
   Msgbox "Name: " & oHost.Name
Next
```

You can also figure out which host your agents will run on by using the *HostName* property on the BoundFolder object. Remember that all agents in a particular folder must run on the same host. You cannot have different agents in the same folder running on different hosts.

If you want to move agents running on one host to another host system, you must use the *MoveBoundFolder* method on the Events object. This method takes two arguments:

- A string that contains the host name you want to move the bindings in the folder to
- The BoundFolder object that contains the bindings you want to move to the new host

Be careful when using this method, because it will move all bindings for a folder to the new host you specify. They all must run on the same host!

EXCHANGE EVENT SCRIPTING AGENT SERVERS

The Exchange Event Service supports servers that can run only agents and that are separate from the home server where the folders generating the events are located. This support allows you to isolate the agent server from your other servers that host mailboxes or public folder applications. It is good practice to set up these agent servers so that errant scripts do not bog down your standard Exchange servers. While occasionally logic errors might make your scripts enter infinite loops or generate errors, the Event Service and agent technologies have built-in timeout capabilities that will terminate bad scripts after a specified amount of time.

RUNNING THE SCRIPT ENGINE IN MTS

You can place the Event Scripting Agent (Scripto.dll) into MTS, which allows you to run the Scripting Agent using a specific Windows NT account for security purposes and also to run the Scripting Agent in a dedicated and isolated process. MTS will manage instantiating the Scripting Agent as well as shutting it down if any anomalies occur during processing. For those of you running Windows 2000, MTS has been enhanced and its capabilities have been integrated directly into COM+. Therefore, wherever you see MTS referenced here, it also refers to COM+. We'll learn more about

COM+ applications in Chapter 18 when we talk about the new event capabilities of Exchange 2000.

To make it easier for you to install the Event Scripting Agent as an MTS component, Exchange Server 5.5 includes a prebuilt MTS package for you to use. To install the package, follow these steps:

1. If you are running Windows NT 4.0, make sure you have MTS installed on the server where you are running the Event Service. (If you are running Windows 2000 Server, you will install MTS packages into COM+ instead.)
2. Start the MTS Explorer by accessing the Start menu and then selecting Programs, Windows NT 4.0 Option Pack, Microsoft Transaction Server, and Transaction Server Explorer.
3. Locate the name of your computer in the Computers tree.
4. Select the folder named Packages Installed, and from the Action menu, select New and then Package.
5. Click the Install Pre-Built Packages button.
6. Click the Add button, and find the Scripto.pak file on the Exchange Server 5.5 CD in the Server\Support\Collab\Sampler\Scripts folder. Select this package, and click Open.
7. Click Next.
8. Click the This User option. Click the Browse button to find the Windows NT account identity you want the script engine to run under. As discussed earlier, this account should have Log On As A Service privileges. Once you have specified an account, click Next.
9. Verify the Install Directory and click Finish.

The Exchange Scripting Agent package should now be installed in MTS, as shown in Figure 13-22.

> **NOTE** If you are interested in learning more about the Event Service and the types of applications you can develop with this technology, you should look at the four sample scripts included on the Exchange Server 5.5 CD in the Server\Support\Collab\Sampler\Scripts folder.

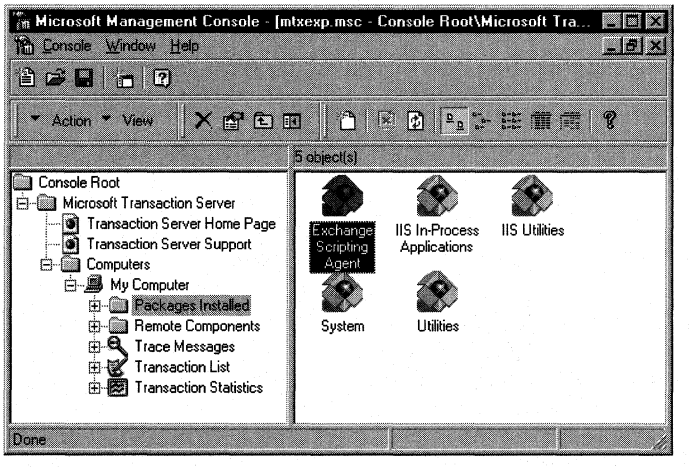


Figure 13-22. *The Exchange Event Scripting Agent installed as an MTS component.*

Chapter 14

Exchange Server Routing Objects

In the previous chapter, we took a look at the Microsoft Exchange Server Event Service technology, which can be used to solve many types of business problems, most commonly those associated with automating administrative tasks and other processes. Business processes usually involve some type of routing, approval, and overall workflow strategy, and while the Scripting Agent technology can handle these routing and workflow applications, it requires developers to write large amounts of code to handle common routing functionality. Most developers don't want to do that. Like you, they'd rather focus on mapping out business processes and have built-in logic implement the most common tasks. To help simplify your development of automated business processes, Microsoft created the Exchange Server Routing Objects.

In this chapter, we will take a look at the architecture for the Exchange Server Routing Objects, which is an extension of the structure for the Event Scripting Agent. Your knowledge of the Event Scripting Agent and the process of creating bindings will enhance your understanding of the Exchange Server Routing Objects architecture.

The easiest way for you to move from creating Event Scripting Agents to creating routing object applications is to convert an existing and applicable Event Scripting application to a routing object application. In this chapter, you will see how the Expense Report application from Chapter 13 can be converted to a routing object application with very little modification. When you first look at the changes, you might wonder what the advantages of creating a routing object application are, but as you look more carefully at the sample, notice how you can modify the flow and logic of the application relatively easily.

EXCHANGE SERVER ROUTING

Of course, before we can dive into the guts of the Exchange Server Routing Objects technology, we first must take a look at the overall architecture. At its highest level, Exchange Server routing technologies, which first shipped in Exchange Server 5.5 Service Pack 1, consists of three components. The first component is a routing engine, which is implemented as a custom event handler for the Event Scripting Agent. The engine itself is a state engine that executes and tracks process instances in a specific Exchange Server folder. A process instance is essentially an item and a corresponding routing map that are part of executing a route. A routing map is a high-level set of instructions that describes the routing. The instructions in the routing map involve intrinsic or custom actions, which are just routing functions to be executed. When events fire within the folder, the engine will process those events and move the process state according to the routing map.

The second component is a set of COM objects called the routing objects. These objects allow you to manipulate the routing map as well as the process instances in your application. Using these objects, you can control what the engine executes and build tools to create routing maps and track current process instances. To demonstrate some of the capabilities of the routing objects, the Agent Install program used in Chapter 13 has been updated to use them. You will see an example of the updated Agent Install program later in this chapter.

The third component is a set of actions. Actions are functions that the routing engine calls as defined by the routing map. When the routing engine executes an action, it will update the state of the process instance according to the results of that action. Actions can be intrinsic actions that the engine understands, such as Goto or Terminate, or they can be custom actions that you write yourself in Microsoft Visual Basic Scripting Edition (VBScript). The Expense Report application from Chapter 13 has been converted to a routing application that uses intrinsic and custom actions. We will look at this Expense Routing application later in this chapter.

Routing Architecture

When combined, these three components form a server-side hub and spoke architecture for routing. Figure 14-1 shows a diagram of Exchange Server routing. The hub, in this case, is a server-based folder that contains your routing map, your custom script for the engine, and an agent on the folder that has the engine as a custom event handler. The hub must meet all the requirements of the Event Scripting Agent, so you cannot create routes in a private folder stored on your local machine. The folders must reside on the server and can be either public or private.

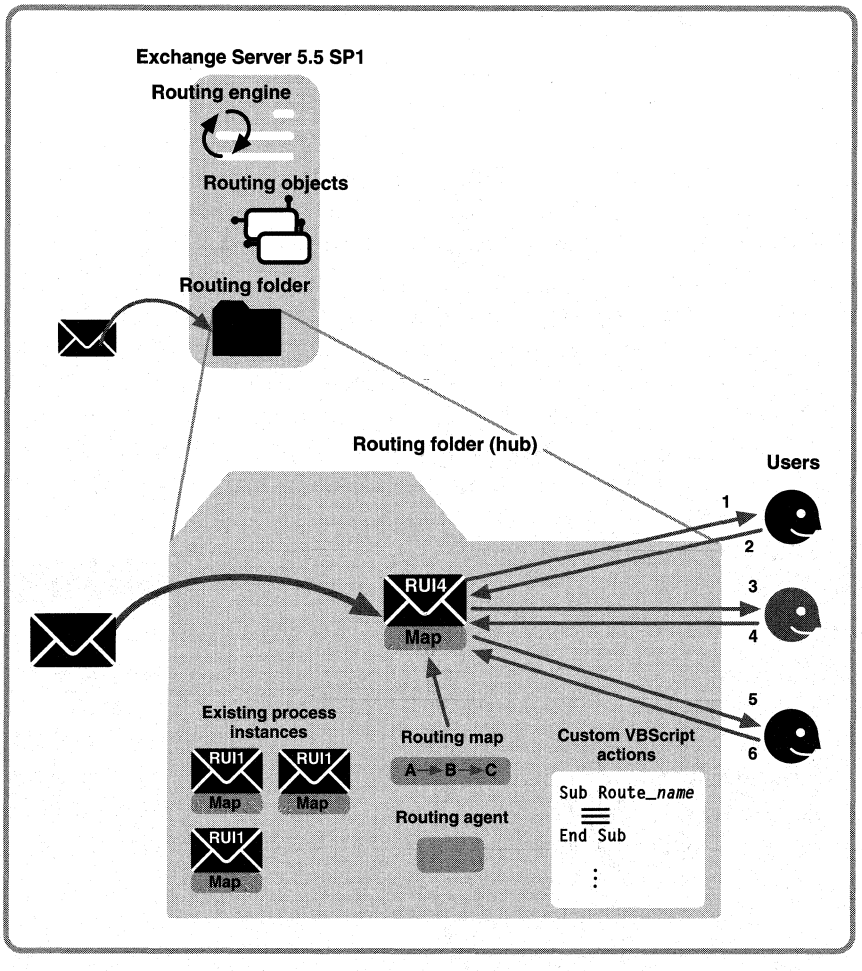


Figure 14-1. *A diagram of Exchange Server routing.*

The spokes in this architecture are the e-mail messages sent from the hub to the recipients or other applications in the route. To track the state of the item after the recipient performs an action, a message must be sent back to the hub so that the engine can update the state of the item and handle any errors that occurred. For this reason, the logical view of your process might be very different from the actual implementation. For example, to route a message between users A, B, and C, the logical view would be to send the message to A, then from A to B, and then from B to C. However, in the actual implementation in the routing architecture, the message must flow through the hub so that the engine can update its status and move the process forward.

Operation of the Routing Engine

As a custom event handler for Event Scripting, the routing engine is dependent on the creation of an agent, or binding, in the folder where you want the engine to run. When creating the agent, the routing engine really uses only two events provided by Event Scripting: the message creation event and the timer event. Whenever a message arrives in the folder or a timer has expired, the engine evaluates the event and determines whether the state of the process needs to be updated.

Recall from the last chapter that the agents in a folder contain the script that executes when the events on that agent occur. Because the routing engine is built on top of the agent architecture, as you would expect, the custom actions you create in VBScript for the engine must be contained in the script for the agent—if they are not, you might receive the error indicating a connection point was not found for the engine. This error usually occurs when you are calling a custom VBScript function, and the script in the agent does not contain it. This is an important point to remember, because you might not immediately associate the script in the agent with the script executed by the routing engine.

Process Instances

You might be wondering how the engine creates a new process instance, and how it tracks these different process instances when hundreds are in the folder. A new process instance is created by adding into a folder a new item that is currently not associated with another process in that folder. For example, suppose you created a new expense report in the folder. First, the engine receives the message creation event. Then the engine tries to correlate the new message with an existing process instance (that is, another message) in the folder.

The engine determines whether a new item is associated with another item in the folder through a unique number called the Route Unique Identifier (RUI). The RUI is assigned to each process instance in a folder. When a new item arrives in a folder, the routing engine can use the RUI to track which process instance the item belongs to. For example, if the folder receives an approval message from a recipient in the route of a process instance, the incoming message contains an RUI so that the routing engine can associate that approval message with the correct process instance. Then the routing engine can update the status and execute the action in the map to perform the necessary functionality. In this example, the most common functionality to execute would be adding the approval to the recipient table on the process instance so that other users can see that the item has been approved.

If the engine cannot find an associated process instance for the new message, it assumes that the new message is a new process instance, and it looks for a routing map on the new item. If a map is found, the engine turns the new message or

item into a new process instance by adding an RUI and some other properties. The engine then starts executing the map on the newly transformed process instance. Since a message can contain its own map, a folder can have items with different maps, allowing routing to be specified for the current need or ad-hoc routing to be implemented. The typical scenario for this type of application is document routing and approval: a document is routed to various people for approval based on the document's content or an individual's expertise. You can implement this functionality by creating a simple form that allows the user to select the document route and then add the map for the route directly to the message.

If a new message does not contain a map, the engine adds to the message the default map for the folder. The routing engine attaches other information to the message, which transforms the message into a new process instance. The default map in the folder is actually stored as a hidden message in the folder. This hidden message contains two important named properties, *RouteMap* and *RouteType*. As you might guess, the *RouteMap* property contains the default map for the folder, and the *RouteType* property can hold the type of route the map is considered to be. With the Routing Wizard sample application in Exchange Server 5.5 Service Pack 1, the *RouteType* property is set by the wizard as either Sequential or Parallel. You can set this type property to be any string you want. You will see later in this chapter how to access the hidden messages in the folder and how to retrieve and set these two properties on your folders.

One of the most important issues to remember is that every item in a particular folder with a routing engine enabled will have some type of map. If the message does not contain a map, the engine will copy the default folder map onto it.

Routing Maps

Now that you know how a new process instance is created and how a map is added to a process instance, you need to take a look at exactly what is contained in a map and how it works. A routing map is a high-level set of instructions written by you that describes the routing process—in other words, a state diagram that contains the logic and flow of a particular business process so that the routing engine can execute it. In the same way that subroutines and functions perform tasks in a program, routing maps reference actions, which are just routing functions that the routing engine calls.

The easiest way to understand routing maps is to take a look at one. The following map in Figure 14-2 is the Expense Report application from Chapter 13, transformed into a routing application. This is a simple example of a map. You can make your maps extremely complex depending on the business process you are modeling in the map.

Default Map

ActivityID	Action	Flags	Parameter1
100	OrSplit	0	CheckTotal
110	Goto	0	7010
120	Wait	0	60
130	OrSplit	0	IsTimeout
140	Goto	0	5000
150	OrSplit	0	ReceivedApprovalMsg
160	Goto	0	6000
170	Goto	0	7000
180	Goto	0	7010
5000	RouteToNextManager	2	
5010	Goto	0	120
6000	UpdateStatus	2	True
6010	Goto	0	7010
7000	UpdateStatus	2	False
7010	Terminate	0	

Figure 14-2. *A routing map for the Expense Routing application displayed with the updated Agent Install program.*

The maps in Exchange Server are required to have three primary fields: ActivityID, Action, and Flags. The ActivityID field is a number value that uniquely identifies a row in the map. It allows you to jump between different rows on the map. If you've ever worked with a programming language that requires line numbering, the concept of the ActivityID field will be very familiar to you.

The second field, Action, identifies the action the engine should perform. This field is a string that must resolve either to an intrinsic action for the engine—such as a Goto action or an OrSplit action—or a custom action, which is a VBScript subroutine in the agent for the folder. (We'll discuss intrinsic actions in the next section.) If neither of these two conditions is met, you will receive an error from the engine.

The final required field, Flags, specifies whether the action in the Action field is an intrinsic action or a custom action. By setting this flag to *0*, you are informing the engine that the action is an intrinsic action. Setting this flag to *2* tells the engine that the action is a VBScript subroutine you implemented.

Depending on the action you select, you can pass parameters in the map. For example, in Figure 14-2, you can see that when the map calls the *UpdateStatus* subroutine, a parameter set to either *True* or *False* is passed to the subroutine, depending on which line of the map is executed. This parameter specifies whether the expense approver approved or rejected the expense report. The VBScript subroutine can then appropriately update the user on the status of the expense report. You are not limited to only one parameter. In fact, you can have multiple parameters in your maps, depending on the needs of your application.

Intrinsic Actions

In a routing map, you can use six intrinsic actions: AndSplit, Goto, New, OrSplit, Terminate, and Wait.

AndSplit

The AndSplit action is used in your map for parallel branching. Parallel branching enables new subprocesses to run independent of one another; the parent process blocks itself until all the subprocesses have finished. The parameters for this action are the ActivityIDs. The engine creates new processes for each of these ActivityIDs in the array and then copies the current map to these new processes. The engine starts execution in each of these new subprocesses at the ActivityID specified in the array.

These new subprocesses will execute until they hit a Terminate action. It is your responsibility to make sure these subprocesses have a Terminate action. It is also your responsibility to copy the necessary properties from the subprocesses and save them onto the parent process before the subprocesses terminate.

Table 14-1 shows an example of a simple map with an AndSplit action. If execution starts at ActivityID 100, this map describes the following routing process. First the process waits 10 minutes and then splits execution at 500 and 700. The 500 branch will send the current message to *Mailbox1* and return. The 700 branch will send the current message to *Mailbox2* and return. With both branches complete, execution resumes at 300, where the process waits 10 minutes and then terminates.

ActivityID	*Action*	*Flags*	*Parameter1*	*Parameter2*
100	Wait	0	10	
200	AndSplit	0	500	700
300	Wait	0	10	
400	Terminate	0		
500	Send	2	*Mailbox1*	
600	Terminate	0		
700	Send	2	*Mailbox2*	
800	Terminate	0		

Table 14-1. *Map using the AndSplit action.*

Goto

Since we are all programmers, I don't need to explain too much about the Goto action. When a Goto action is executed, the process jumps to a specified ActivityID in your map and continues executing. Table 14-2 shows a map illustrating the use of multiple Goto actions—for example, if execution starts at ActivityID 100, the process will jump to 300, then jump to 200, and then jump to 400, at which point it terminates.

ActivityID	*Action*	*Flags*	*Parameter1*
100	Goto	0	300
200	Goto	0	400
300	Goto	0	200
400	Terminate	0	

Table 14-2. *Map using multiple Goto actions.*

New

The New intrinsic action creates a new process instance and begins executing it. As with the AndSplit action, the routing engine copies the current map over to the new process instance. The only parameter you pass to this action is the ActivityID in the new process instance where the engine should start executing. The new process instance will be created in the folder, and when the Terminate action is triggered for the new process, the process instance will be removed from the folder. Both the original and the new process instance run at the same time. Table 14-3 shows a map using the New intrinsic action. Starting at ActivityID 100, this process creates a new process and terminates. The new process begins execution at 300, executes the specified actions, and then terminates.

ActivityID	*Action*	*Flags*	*Parameter1*
100	New	0	300
200	Terminate	0	
300	Your Action Here	2	
400	Terminate	0	

Table 14-3. *Map using the New action.*

OrSplit

The OrSplit intrinsic action is like an If statement in programming. You pass a parameter to this action that is the name of a VBScript subroutine that returns either *True* or *False*. If the subroutine returns *True*, the line immediately following the OrSplit action executes. If the subroutine returns *False*, the next row in the map is skipped and the row after that is executed. You can nest these actions to create nested If statements. The map in Table 14-4 shows you how to use the OrSplit action. Starting at ActivityID 100, if *MySub* returns *True*, the Goto action will be executed and will jump to 400. If *MySub* returns *False*, the Terminate action at 300 will be executed.

ActivityID	*Action*	*Flags*	*Parameter1*
100	OrSplit	0	*MySub*
200	Goto 0	400	
300	Terminate	0	
400	Terminate	0	

Table 14-4. *Map using the OrSplit action.*

Terminate

The Terminate action ends the currently running process instance. This action takes no parameters and can occur anywhere in your map.

Wait

The Wait action causes the engine to wait until a specified amount of time has elapsed. This action takes as its parameter the number of minutes to wait. After the time limit is reached, the next row executes. When you are in a Wait action, your map is not blocked. Use the Wait action in your maps to program timeouts that give participants a finite amount of time in which to respond.

The map in Table 14-5 is a section from the Expense Report application map, and it shows you how to use the Wait action. Starting at ActivityID 120, the engine will wait 60 minutes before executing the next line in the map, which is the OrSplit action. This wait time gives the manager of the person who submitted the expense report time to approve or reject the report. If the manager does not approve the report in one hour, the time limit for the Wait action will expire, and the report will be routed to the manager's manager. If, however, the engine is waiting for the timeout to occur and an approval or rejection message is sent to the folder with the correct RUI for the expense report, the *ReceivedApprovalMsg* subroutine will be called.

ActivityID	*Action*	*Flags*	*Parameter1*
120	Wait	0	60
130	OrSplit	0	*IsTimeout*
140	Goto	0	5000
150	OrSplit	0	ReceivedApprovalMsg

Table 14-5. *Map using the Wait action.*

Custom Script Actions

While the six intrinsic actions control the flow of the engine when processing the map, they really do not implement any application functionality; to do that, you will have to create custom script actions in VBScript. (You must write your script actions in

VBScript because currently it is the only supported scripting language for creating custom actions.) The script actions can be used in your maps and will be called by the routing engine during execution. In your script, you can also call COM objects to perform your work.

Writing a script action requires that you properly name the VBScript subroutine that implements the action. You must prefix the subroutine name with the text *Route_*. For example, if you wanted to have a CheckTotal action, which checks the total of an expense report to see whether it can be automatically approved, you would name your subroutine *Route_CheckTotal*. If you don't use the Route_ syntax, the engine will generate an error stating that a connection point could not be found.

To help you implement the most common types of actions you'll perform with the routing engine, Microsoft provides a script file named Routing.vbs. This script file contains 16 route actions and a number of helper functions. The Routing.vbs file is available from the Microsoft Web site at *http://www.microsoft.com/technet/resource/download/exchange/misc* under Routing Script Source Code (routingsrc.exe). Table 14-6 lists the route actions and describes the functionality of each.

Action	*Description*
AutoSet	Provides autoapprove and autoreject functionality for your routes.
CreateNote	Converts an IPM.Post message to an IPM.Note message so that the status of the message can be tracked using Microsoft Outlook.
Consolidate	Takes the message body and any attachments of a reply message and adds them to the original process instance message.
FinalizeReport	Creates and sends a summary report about the status of the process instance.
IsApprovalMsg	Checks the sender of a message to determine whether the sender is on the recipient list for the route. If the sender is on the list, the action checks whether the sender has approved the item. It returns *True* for approval or *False* for rejection.
IsApprovedTable	Tallies all approval or rejection votes. The action returns *True* if the number of approvals is greater than the number of rejections or *False* if rejections are greater than approvals.
IsInvalidRecip	Checks a received response message to make sure that it is from the correct person in the routing sequence and from a person from whom a response is expected. If the message is not from the correct person, the message is ignored.

Table 14-6. *Route actions of Routing.vbs script file.*

Table 14-6. *continued*

Action	*Description*
IsNDR	Checks to see whether the e-mail received is a nondelivery report. The action returns *True* if it is, *False* if it is not.
IsOOF	Checks to see whether e-mail received is an out-of-office message. The action returns *True* if it is, *False* if it is not.
IsPost	Checks to see whether the item in the folder is a Post message, or IPM.Post. The action returns *True* if it is, *False* if it is not.
IsReceipt	Checks to see if the e-mail received is a receipt message such as a delivery, a read, or a non-read receipt. Returns *True* if it is, *False* if it is not.
IsTimeout	Checks to see whether the process instance timeout has expired. Returns *True* if it has, *False* if it has not.
NOP	Performs a No Operation. You can use this function as a placeholder if you want to modify a map in progress.
PreProcessing	Performs initialization of the route, such as changing IPM.Document and IPM.Post items into IPM.Note.
Receive	Processes reply messages that correspond to a current process instance. This subroutine also handles voting button responses.
Send	Sends the routing message to the recipient. This message can include a work item as an attachment, or it can have an Outlook Web Access (OWA) link to the work item. You can specify either the address of the participant or the role.

What About Roles?

When developing routing applications, you'll frequently want to use a dynamic lookup to locate individuals to route items to. You could implement this lookup as a custom script function that searches in a database or in a flat file. But you could also use Exchange Server's directory, which contains information about the recipients inside or outside your system.

The Exchange Server directory provides one built-in role—Manager. This is the most typically used role; when you use it, items are dynamically sent to the current person's manager in the directory. This role is recognized automatically by the Routing Wizard sample application. If you do not use the script actions from the Routing Wizard in your own applications, you will need to recognize and look up the Manager role in your own custom script.

NOTE The roles discussed in this section are different from the security roles that you see on the Permissions tab in Outlook. The roles discussed in this section are a special type of distribution list, where the owner is called the role performer. Security roles, such as Reviewer, on the other hand, make it easier for you to set permissions on a folder.

You can extend the Exchange Server directory with your own custom roles. Included with Exchange Server 5.5 Service Pack 1 is a Role Administrator program for the Exchange Server directory. Using this program, you can create custom roles that have a role performer, such as expense approver, and people for whom the role performer performs the function, such as Frank, Jane, and Scott. The interface for the Role Administrator program is shown in Figure 14-3.

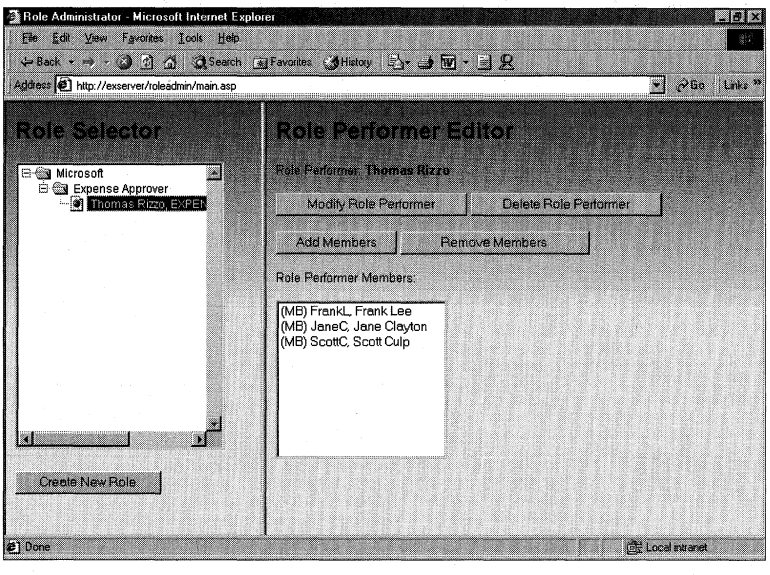


Figure 14-3. *The Role Administrator program is implemented as an Active Server Pages (ASP) application.*

Roles are implemented in the directory as nested distribution lists. The easiest way to understand the relationships between these nested distribution lists is to look at an example of one. Let's imagine that Thomas is the expense approver for Frank, Jane, and Scott. The expense approver role is represented as a distribution list in the directory. In that distribution list are nested other distribution lists that contain the role performers as the owner of the distribution list. In one of those nested distribution lists, Thomas is the owner. Frank, Jane, and Scott are members of that nested distribution list. Thomas, then, is the role performer for Frank, Jane, and Scott. Figure 14-4 shows the Properties dialog box of a nested distribution list in the Exchange Administrator program, with Thomas as the expense approver for Frank, Jane, and Scott.

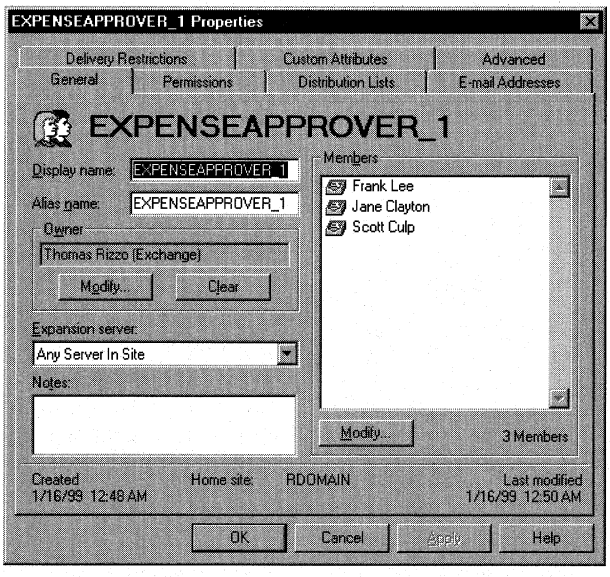


Figure 14-4. *The Properties dialog box for a nested distribution list that was automatically created with the Role Administrator program. Thomas is the role performer of the Expense Approver role, and Frank, Jane, and Scott are members of the Expense Approver role.*

Distribution lists representing roles are unique in that their property *PR_GIVEN_NAME (&H3A06001E)* contains the text *ROLEPERFORMER*. By using this property and the CDO Address Entry Filter object, you can quickly create applications that find and display all the roles in your Exchange Server directory.

EXPENSE ROUTING APPLICATION

To convert the Expense Report application from Chapter 13 to a routing application, a few major changes were made to create a routing map, update the ASP pages so that they send e-mail messages to the expense routing folder rather than update the status of the expense items directly, and implement some custom script actions. Before looking at the changes to the ASP pages and script, you need to set up the Expense Routing application.

Setting Up the Expense Routing Application

Before you can install the application, you must have a Microsoft Windows NT 4.0 Server and a client with certain software installed. Table 14-7 describes the installation requirements.

Required Software	*Installation Notes*
Exchange Server 5.5 Service Pack 1 with Outlook Web Access	Exchange Server 5.5 Service Pack 1 or later installs the routing engine and routing objects.
Microsoft (IIS) 3.0 or later with Active Server Pages	IIS 4.0 or later is recommended.
CDO library (cdo.dll), CDO Rendering library (cdohtml.dll)	Exchange Server 5.5 Service Pack 1 installs CDO library 1.21 and CDO Rendering library 1.21. Outlook installs CDO library 1.21.
For the client:	
A Web browser, Outlook 2000, Visual Basic 6.0	For the Web browser, Microsoft Internet Explorer 4.0 or later is recommended. You can run the client software on the same machine or on a separate machine.

Table 14-7. *Installation requirements for the Expense Routing application.*

To install the Expense Routing application, copy the Expense Routing folder from the companion CD to your Web server where you want to run the application. Start the IIS administration program. Create a virtual directory that points to the location where you copied the expense routing files, and name the virtual directory *expenserouting*. Make sure you enable the Execute permissions option for the virtual directory. You will be able to use the following URL to access your Expense Routing application: *http://*yourservername*/expenserouting*.

Open the Exchange Administrator program. Open the Properties dialog box for the Folders\System Folders\Events Root\EventConfig_*servername* folder. Click the Client Permissions button, add a user who will administer the Expense Routing folder, and grant the user Author permissions. Click OK twice.

Launch Outlook using the user you selected to administer the folder Expense Routing. Create a new public folder named *Expense Routing* under All Public Folders. Next, verify that the Server Scripting add-in is installed. By default, Outlook does not install the Server Scripting add-in. To install the Server Scripting add-in, select Options from the Tools menu, click on the Other tab, click Advanced Options, and then click Add-In Manager. Check the Server Scripting check box in the Add-In Manager dialog box. In the Exchange Administrator program, open the Properties dialog box for the Expense Routing public folder. Click on the Advanced tab, uncheck the Hide From Address Book check box, and click OK.

Start the Registry Editor on your server, and open this key:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\MSExchangeES\
Parameters
```

Set Logging Level DWORD to *5*, to log the maximum amount of information.

NOTE Be sure to set Logging Level to *0* when you are finished testing the Expense Routing application. If you do not, your Application log will be quickly filled up with MSExchangeES logging entries.

Next, you will create a default map using the Exchange Routing Wizard. To install the Routing Wizard on your client machine, run Rwsetup.exe from the Exchange Server 5.5 Service Pack 1 CD. For Microsoft Windows 95 and Windows 98, the location for Rwsetup.exe is C:\Eng\Server\Support\Collab\Sampler\Routing\Win95. For Microsoft Windows NT and Windows 2000 Server, the location is C:\Eng\Server\Support\Collab\Sampler\Routing\Winnt\i386. You must have Outlook installed on the machine on which you are installing the Routing Wizard.

After you have installed the Routing Wizard, run it by selecting it from the Programs menu. The Routing Wizard requires you to log on to Exchange Server. Use the account that you used to create the Expense Routing public folder. In the Step 1 screen, choose the Expense Routing public folder as shown in Figure 14-5. Step through the remaining screens of the wizard. Add recipients when necessary. Since the Routing Wizard is being used only to create a default map, which we will modify later, the remaining settings in the wizard are not important.

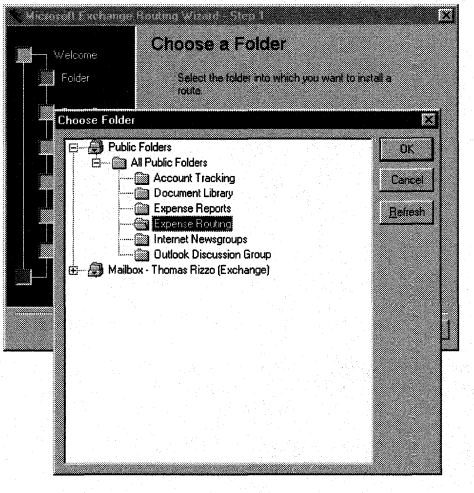


Figure 14-5. *Selecting the Expense Routing public folder to install a default routing map.*

Locate the RoutingAgentScript.txt file included with the expense routing files. Be sure the read-only flag for the file is unchecked, and open it in Notepad. Perform a search in the code, and replace all instances of the text *localhost* with the name of your Web server. Save and close Notepad.

Run the updated Agent Install program in the Agent Install Updated folder on the companion CD. Log on to your Exchange server using the account you used to create the Expense Routing public folder. Select the Expense Routing public folder, as shown in Figure 14-6.

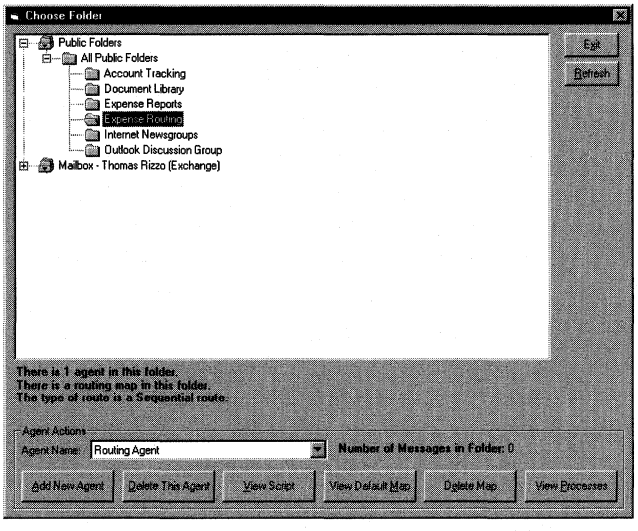


Figure 14-6. *Selecting the Expense Routing public folder in the updated Agent Install program. The agent and map contained in the Expense Routing folder were added using the Routing Wizard.*

Click the Delete This Agent button, and then click OK to delete the agent in the Expense Routing public folder generated by the Routing Wizard. Click No when asked to delete the routing map. Click the Add New Agent button, click Yes when asked to create a routing agent, and then click OK to select a script. In the Select Script dialog box, shown in Figure 14-7, select the RoutingAgentScript.txt file that you modified earlier, and click OK.

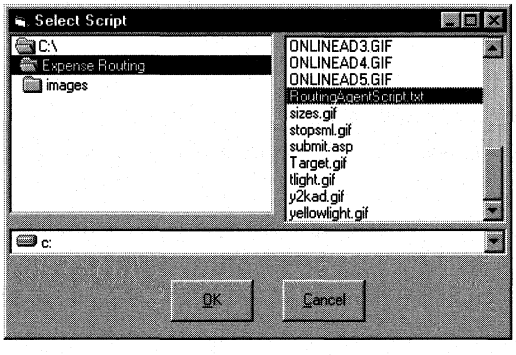


Figure 14-7. *Selecting the script for the Expense Routing agent.*

In the main interface for the updated Agent Install program, click the View Default Map button. Click the Delete All Rows button to clear the map. Use the Delete Column button to delete all parameter columns except Parameter1. Click the Select Script button. When asked whether you want to use the default script in the folder, click Yes. Check the Parse Script For Functions check box, and click OK after the program tells you how many new functions were added from the script. Enter the map instructions shown in Figure 14-8. Double-click your mouse or press the Enter key to edit a cell, and use the Add Row button to add rows. You will be able to select intrinsic and custom actions from a combo box that is displayed in the Action column. When finished, click Save, and then click Exit.

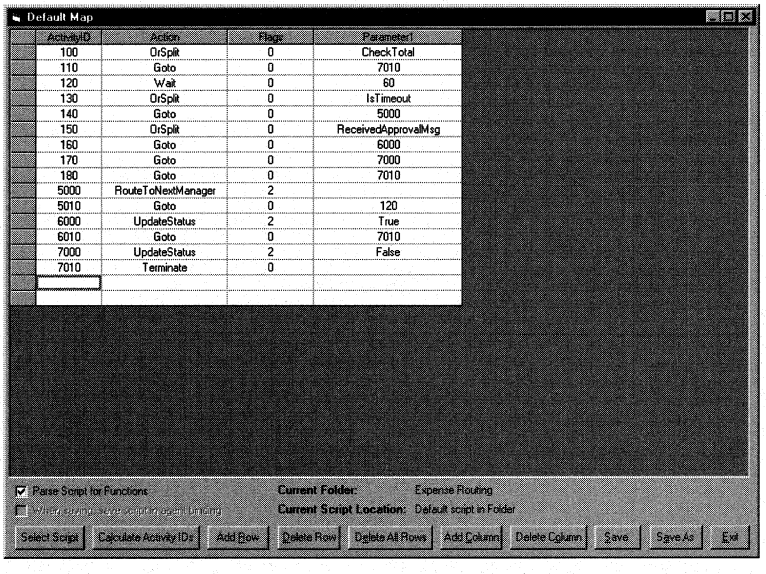


Figure 14-8. *The routing map for the Expense Routing agent.*

To test the Expense Routing application, use the following URL to submit expense reports: *http://*yourservername*/expenserouting*. The application is used in the same way as the Expense Report application in Chapter 13. Figure 14-9 shows the Expense Report status page with some sample expense reports in different routing states. To see sample expense reports, open the Expense Routing.pst file included with the expense routing files in Outlook.

NOTE If the application does not work as expected, check that you have Service Pack 1 of Exchange Server installed. Also, check the Application log of the Event Viewer for any logged errors.

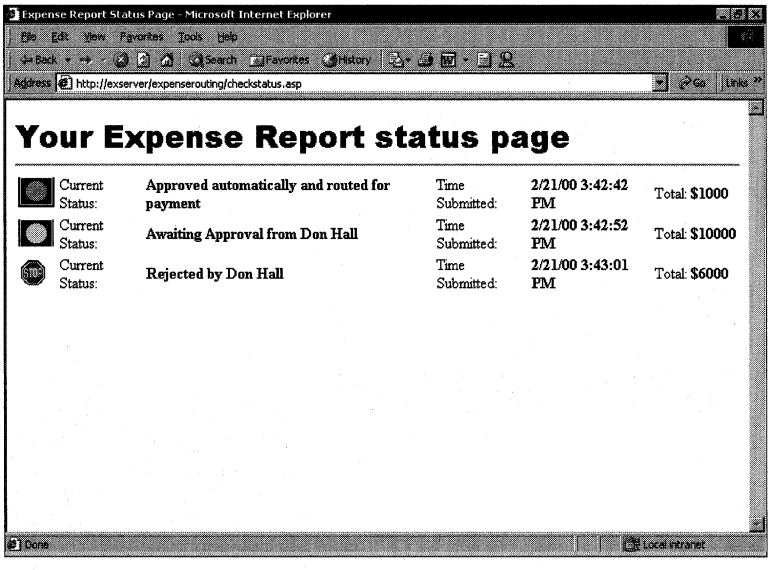


Figure 14-9. *The status page for the Expense Routing application, where expense reports are in different routing states.*

Changes to the ASP Section of the Application

The biggest changes to the ASP section of the application occurred in the approval and rejection code for an expense report. The expense approval and rejection code had to be changed so that it did not update the item directly with the approval or the rejection of the expense report. Instead, the code was modified so that it sent e-mail to the requesting user with the response of the expense approver, regardless of whether this response was an approval or rejection. To implement this functionality, the code creates an e-mail message, and it adds a subject containing the string "*APPWF*" for approval and "*REJWF*" for rejection, as well as the RUI number for the process instance that this approval or rejection is for. The application appends "*RUI=number*" to the subject.

To retrieve the RUI on an approval or rejection in the ASP application, the custom script action that sends the message to the manager for approval has to pass the RUI to the ASP application. To do this, as you will see, the custom script action calls into the Routing Object library and uses a specific property contained on the ProcInstance object, named RUI.

Once the ASP application has all the necessary information, it can create a fully formed approval or rejection message with the RUI and mail this to the Expense Routing public folder. The next code segment, from finalapprove.asp, shows creating the approval or rejection message and sending it to the folder. The code used to implement the Expense Report application in Chapter 13 is commented out in the listing so that you can compare the two implementations.

```
<!--#include file="logon.inc"-->
<%
Dim oMessage
Dim AMSession

CheckAMSession
BAuthenticateUser
Set AMSession= Session( "AMSession")
if AMSession Is Nothing Then
   'CheckSession was unable to retrieve or create a session
   Response.Write( "GetAMSession returned nothing!<br>")
End If
Mailbox = Session("Mailbox")
set objFolder = Session("objFolder")
if objFolder is Nothing then
   Response.Write( _
      "Cannot access the Expense Report folder!<br>")
end if
objInfoStoreID = Session("objInfoStoreID")
'Get the EntryID for the message from the query string
   objMessageID = Session("Entryid")
'Get the message by its ID
   set oMessage = AMSession.GetMessage(objMessageID, objInfoStoreID)
'Get the Fields of the report
   set Total = oMessage.Fields("Total")
'Get the user who posted the message
   set addentry = oMessage.Sender
   set UsersManager = addentry.Manager
   Approver = Session("Approver")

set newExpenseReport = AMSession.outbox.messages.add
set mynewrecipient = newExpenseReport.recipients.add
'Need to change this to a different folder if it is different
mynewrecipient.Name = "Expense Routing"
mynewrecipient.resolve
lRUI = Session("RUI")
if Request.Form("Approve") = "Approve" then
   newExpenseReport.subject = "APPWF:,RUI=" & Cstr(lRUI)
   'newExpenseReport.text = Approver & _
   '" has approved the expense report.  " & _
   '"The total value of this report was " & Total
   newExpenseReport.send
   'oMessage.Fields("StatusInt") = 3 'Approved
   'oMessage.Fields("Status") = "Approved by " & Approver
   'oMessage.Fields("Approver") = Approver
   currentstatus = 3
else
   newExpenseReport.subject = "REJWF:,RUI=" & Cstr(lRUI)
```

(continued)

```
    'newExpenseReport.text = Approver & _
    '" has rejected the expense report.  " & _
    '"The total value of this report was " & Total
    newExpenseReport.send
    'oMessage.Fields("StatusInt") = 4 'Rejected
    'oMessage.Fields("Status") = "Rejected by " & Approver
    'oMessage.Fields("Approver") = Approver
    currentstatus = 4
end if
'oMessage.Update TRUE, TRUE
%>
<head>
<meta http-equiv="Content-Type"
content="text/html; charset=iso-8859-1">
<% if currentstatus = 3 then %>
<title>Expense Report Approved Page</title>
<% else %>
<title>Expense Report Rejected Page</title>
<% end if %>
</head>

<body bgcolor="#FFFFFF">
<p><font size="2" face="Arial Black">
<% if currentstatus = 3 then %>
Routing Final Approval to <%=addentry.Name%></font></p>
<hr>
<font size="2" face="Arial Black">
The following funds have been successfully transferred:
<p align="center">$<%=oMessage.Fields("Total")%></font></p>
<% else %>
Routing Rejection message to <%=addentry.Name%></font></p>
<hr>
<% end if %>
</body>
</html>
```

Changes to the Server Script

Whereas the changes to the ASP application were very minor, the changes to the server-side script that runs on the Exchange server were somewhat more extensive. The first change was to turn the scripting agent script into a routing agent script. We did this by modifying the primary subroutine names so that they had the Route_ naming convention rather than the scripting agent convention. The script also had to be updated so that Boolean values were returned by some of the subroutines. This was necessary to implement branching using OrSplits in the routing map.

The script also needed functionality to update the status of the expense report when an approval or rejection message is received in the folder—functionality originally implemented in the ASP application. The status is defined as custom properties on the message.

The following is the updated script code, RoutingAgentScript.txt, that turns the Expense Report application into a routing application:

```
<SCRIPT RunAt=Server Language=VBScript>

'--------------------------------------------------------------------
'FILE DESCRIPTION: Expense Report Sample Script for routing objects
'
'--------------------------------------------------------------------

'--------------------------------------------------------------------
'Localized strings
'--------------------------------------------------------------------
'Put all localizable strings and constants here

'--------------------------------------------------------------------
'Global Variables
'--------------------------------------------------------------------
Dim AMSession
Dim fldrOutbox
Dim msgTarget
Dim fldrTarget
Dim ExpRentalCar
Dim ExpAirfare
Dim ExpHotel
Dim ExpMeals
Dim ExpTotal
Dim g_oMsgIn          'Incoming Message Object
Dim g_oProcInstance   'Process Instance
Dim g_oPIMsg          'Process Instance Message Object
Dim g_oSession        'Session Object
Dim g_oFolder         'Routing Folder Object
Dim g_oAgentAddEntry  'Address Entry Object used for Inbox mailings

'--------------------------------------------------------------------
'Route Subs
'--------------------------------------------------------------------

'DESCRIPTION: Check the total of the expense report, and if it
'is under a specific amount, automatically approve the expense
'report
Sub Route_CheckTotal(boolSuccess)
   Dim msgResponse
   Dim iMsgCount
   Dim msgManager
   Dim UsersManager
   Dim currentuser
   Dim currentapprover
```

(continued)

```
On Error Resume Next
If InitializeObjects Then
   WriteToLog 1,"Message Count Succeeded"
   set ExpTotal = msgTarget.Fields.Item("Total")
   Set msgResponse = fldrOutbox.Messages.Add
   'Get the RUI
   lRUI = g_oProcInstance.RUI
   'Modify this line to change Expense Amount
   If ExpTotal > 5000 then
      WriteToLog 1,"Greater than max expense amount"
      msgResponse.Subject = "The Total was " & ExpTotal
         'Message to manager
      set msgManager = fldrOutbox.Messages.Add
      set currentuser = msgTarget.Sender
      set UsersManager = currentuser.Manager
      currentapprover = UsersManager.Name
      msgResponse.Text = "This Expense Report has " & _
         "been routed to your Manager: " & currentapprover
      'Get the spaces out
      currentapprover = Replace(currentapprover," ","+")
      msgManager.Subject = _
         "Approval Required for Expense Report!"
      'Need to change this for your server and directory
      msgManager.Text = currentuser.name & _
      " has submitted an expense report for " & _
      ExpTotal & ".  Please review it at http://" & _
      localhost/expenserouting/approve.asp?entryid=" & _
      msgTarget.ID & "&Approver=" & CurrentApprover & _
      "&RUI=" & Cstr(lRUI)
      msgManager.Recipients.Add "","",1,UsersManager.ID
      msgManager.Recipients.Resolve(False)
      msgManager.Send
      msgTarget.Fields("strStatus") = "Awaiting " & _
         "Approval from " & UsersManager.Name
      msgTarget.Fields("StatusInt") = 2
      msgTarget.Fields.Add "Approver",8,UsersManager.Name
      msgTarget.Update

   Else 'Expense Report <= Max Amount
      WriteToLog 0,"Less than max expense amount"
      msgResponse.Subject = _
         "This Expense Report has been Approved"
      msgResponse.Text = "Your expense report for " & _
         ExpTotal & " has been automatically approved." & _
         "Funds are being transferred!"
      msgTarget.Fields("strStatus") = "Approved " & _
         "automatically and routed for payment"
      msgTarget.Fields("StatusInt") = 3
      msgTarget.Update
```

```
            boolSuccess = True
         End If
         If Err.Number = 0 Then
            msgResponse.Recipients.Add "", "", 1, _
               msgTarget.Sender.ID
            If Err.Number = 0 Then
               msgResponse.Recipients.Resolve(False)
               If msgResponse.Recipients.Resolved = True Then
                  msgResponse.Send
                  If Not Err.Number = 0 Then
                     WriteToLog 0, _
                        "Message.Send Failed: " & _
                        Err.Description
                  End If
               Else
                  WriteToLog 0, _
                     "Recipients.Resolve Failed: " & _
                     Err.Description
               End If
            Else
               WriteToLog 0,"Recipients.Add Failed: " & _
                  Err.Description
            End If
         Else
            WriteToLog 0,"Messages.Add Failed: " & _
               Err.Description
         End If
      Else
         WriteToLog 0,"InitializeObjects Failed: " & _
            Err.Description
      End If
      ReleaseGlobalObjects
   End Sub

   Sub Route_IsTimeout(boolSuccess)
      On Error Resume Next
      Dim boolRes          'Boolean Result

      WriteToLog 0, "Starting IsTimeout"
      boolRes = InitializeObjects
      If Not boolRes Then
         WriteToLog 0, "InitializeObjects Failed"
      Else
         boolSuccess = g_oProcInstance.Timeout
         Set g_oProcInstance = Nothing
      End If
      WriteToLog "IsTimeout returns " & boolSuccess
      ReleaseGlobalObjects
   End Sub
```

(continued)

```
Sub Route_RouteToNextManager()
   On Error Resume Next
   Dim boolRes          'Boolean Result

   WriteToLog 0, "Starting RouteToNextManager"
   boolRes = InitializeObjects
   If Not boolRes Then
      WriteToLog 0, "InitializeObjects Failed"
   Else
      'Get the RUI
      lRUI = g_oProcInstance.RUI
      'Clear errors
      Err.Clear
      WriteToLog 0,"Rerouting beginning"
      set ExpTotal = msgTarget.Fields("Total")
      WriteToLog 0,"The Total is: " & ExpTotal
      'Reroute the message
      set CurrentApprover = msgTarget.Fields("Approver")
      WriteToLog 0,"The current approver is: " & CurrentApprover
      set msgResponse = AMSession.Outbox.Messages.Add
      'Create the recipient
      Set objonerecip = msgResponse.Recipients.Add
      objonerecip.Name = CurrentApprover
      'Resolve the name against the Exchange Server
      'directory
      objonerecip.Resolve
      'Get the address entry so that we can pull out template
      'information
      Set myaddentry = objonerecip.AddressEntry
      'Get the manager from the address entry
      set NextApprover = myaddentry.Manager
      if NextApprover = Empty then 'We don't have a manager!
         'Send a message to the current user
         set currentuser = msgTarget.Sender
         msgResponse.Subject = "No more manager to route to"
         msgResponse.Text = currentuser.name & _
            " has submitted an expense report for " &
            ExpTotal & ". There are no other managers " & _
            "to route to!"
         msgResponse.Recipients.Add "", "", 1, _
            msgTarget.Sender.ID
         msgResponse.Send
         'Resend a message to the current approver
         Set msgResendtoApprover = _
            AMSession.Outbox.Messages.Add
         CurrentApproverName = _
            Replace(CurrentApprover," ","+")
         msgResendtoApprover.Subject = _
            "Repeat notice for Approval of an Expense Report!"
```

```
    msgResendtoApprover.Text = currentuser.name & _
    " has submitted an expense report for " & ExpTotal & _
    ". Please review it at http://localhost/" & _
    "expenserouting/approve.asp?entryid=" & _
    msgTarget.ID & "&Approver=" & CurrentApproverName & _
    "&RUI=" & Cstr(lRUI)
    'Create the recipient
    set oRecip = msgResendtoApprover.Recipients.Add
    oRecip.Name = CurrentApprover
    oRecip.Resolve
    msgResendtoApprover.Send
    WriteToLog 0,"No More Managers beyond " & _
       CurrentApprover & " for this user."
else
    NextApproverName = NextApprover.Name
    tmpNextApproverName = NextApprover.Name
    'Got the next approver. Send a message to previous
    'approver and user and reroute.
    set currentuser = msgTarget.Sender
    msgResponse.Subject = _
       "An Expense Report has been rerouted"
    msgResponse.Text = currentuser.name & _
    " has submitted an expense report for " & ExpTotal & _
    ". It was rerouted because the 1-hour approval " & _
    "time limit has expired. It is now routed to " & _
    NextApproverName
    msgResponse.Recipients.Add "", "", 1, _
       msgTarget.Sender.ID
    msgResponse.Send
    if err.number = 0 then
       WriteToLog 0,"Successfully rerouted"
    end if
    'Now change the status and reroute to new person
    WriteToLog 0, "Updating fields on Message"
    msgTarget.Fields("strStatus") = "Rerouted and " & _
       "awaiting Approval from " & NextApproverName
    msgTarget.Fields("Approver") = NextApproverName
    msgTarget.Update
    'Now send a message
    WriteToLog 0, "Creating message to next " & _
       "approver: " & NextApproverName
    Set msgNewApprover = AMSession.Outbox.Messages.Add
    WriteToLog 0, "Added message to outbox"
    'Create the recipient
    'Get the spaces out
    NextApproverName = Replace(NextApproverName," ","+")
    WriteToLog 0, "Got the spaces out of the name"
    msgNewApprover.Subject = _
```

(continued)

```
            "Approval Required for Rerouted Expense Report!"
         WriteToLog 0, "Added Subject"
         msgNewApprover.Text = currentuser.name & _
         " has submitted an expense report for " & ExpTotal & _
         ". Please review it at http://localhost/" & _
         "expenserouting/approve.asp?entryid=" & _
         msgTarget.ID & "&Approver=" & NextApproverName & _
         "&RUI=" & Cstr(lRUI)
         WriteToLog 0, "Added Text"
         Set tmpRecip = msgNewApprover.Recipients.Add
         tmpRecip.Name = tmpNextApproverName
         WriteToLog 0, "Added Recipient"
         msgNewApprover.Recipients.Resolve
         WriteToLog 0, "Resolved Address"
         msgNewApprover.Send
      end if 'Manager!
   end if
   WriteToLog 0,"RouteToNextManager is done."
   ReleaseGlobalObjects
End Sub

Sub Route_UpdateStatus(boolApproved)
   On Error Resume Next

   WriteToLog 0, "Starting UpdateStatus with value: " & _
      boolApproved
   boolRes = InitializeObjects
   If Not boolRes Then
      WriteToLog 0, "InitializeObjects Failed"
   Else
      'Check to see whether approved or rejected, update the
      'status of the message, and send an e-mail
      Approver = g_oMsgIn.Sender
      WriteToLog 0, "Approver: " & Approver
      set newExpenseReport = AMSession.outbox.messages.add
      set mynewrecipient = newExpenseReport.recipients.add
      mynewrecipient.Name = msgTarget.Sender   'Original Sender
      mynewrecipient.resolve
      Total = msgTarget.Fields("Total")
      if boolApproved then   'Approved
         WriteToLog 0, "Sending approve message because " & _
            "boolApproved is " & boolApproved
         msgTarget.Fields("strStatus") = "Approved by " & _
            Approver
         msgTarget.Fields("StatusInt") = 3
         msgTarget.Fields("Approver") = Approver
         newExpenseReport.subject = "Your expense " & _
            "report has been Approved!"
         newExpenseReport.text = Approver & _
```

```
            " has approved the expense report. The total " & _
            "value of this report was " & Total
         newExpenseReport.send
         WriteToLog 0, "Sent Approval Message"
      else   'Rejected
         WriteToLog 0, "Sending reject message because " & _
            "boolApproved is " & boolApproved
         msgTarget.Fields("strStatus") = "Rejected by " & _
            Approver
         msgTarget.Fields("StatusInt") = 4
         msgTarget.Fields("Approver") = Approver
         newExpenseReport.subject = "Your expense " & _
            "report has been Rejected!"
         newExpenseReport.text = Approver & " has " & _
            "rejected the expense report. The total " & _
            "value of this report was " & Total
         newExpenseReport.send
         WriteToLog 0, "Sent Rejection Message"
      end if
      msgTarget.Update TRUE, TRUE
      WriteToLog 0, "Updated Message Status"
   end if
   ReleaseGlobalObjects
end sub

Sub Route_ReceivedApprovalMsg(boolSuccess)
   On Error Resume Next
   Dim varRet           'Variant return value
   Dim boolRes          'Boolean result
   Dim oRecipientEntry  'VoteTable recipient entry object
   Dim bstrUSubject     'Uppercased subject from incoming message

   WriteToLog 0, "Starting ReceivedApprovalMsg"
   boolRes = InitializeObjects
   If Not boolRes Then
      WriteToLog 0, "InitializeObjects Failed"
   Else
      'Notes: Outlook approval/reject buttons place the string
      '      Approve:" on the subject line, the URL version
      '      places APPWF: on the subject line.
      '      Check the subject line to find out whether
      '      it's an Outlook message or a non-Outlook message.
      'In expense routing sample, assume it will always be
      'a non-Outlook message
      bstrUSubject = UCase(g_oMsgIn.subject)
      'Look for APPROVE in subject
      If InStr(1, UCase(g_oMsgIn.subject), "APPWF") Then
         WriteToLog 0, "Message Approval Found."
         boolSuccess = True
```

(continued)

```
      'Otherwise look for REJECT in subject
      ElseIf InStr(1, UCase(g_oMsgIn.subject), "REJWF") Then
         WriteToLog 0,"Message Reject Found."
         boolSuccess = False
      End If
   end if
   ReleaseGlobalObjects
   WriteToLog 0, "ReceivedApprovalMsg Exit returned " & _
      boolSuccess
end sub

'--------------------------------------------------------------------
'Support Functions
'--------------------------------------------------------------------

'Description: WriteToLog
Private Sub WriteToLog(boolRecordName,strMessage)
   Dim strResponse
   strResponse = Now & vbTab & strMessage & ":"
   if boolRecordName = 1 then
      strResponse = strResponse & " " & msgTarget.Subject
   else
      strResponse = strResponse & " "
   end if
   Script.Response = Script.Response & vbNewLine & strResponse
end sub

'++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
'   Name: InitializeObjects
'   Area: Utility
'   Desc: Set Message, Folder, and other globals.
'       Check store that agent is operating in.
'   Parm: None
'   Retn: Boolean Success
'++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 Private Function InitializeObjects()
   On Error Resume Next
   Dim bstrTemp
   Dim oStores          'InfoStores Object
   Dim oStore           'Store Object
   Dim bstrPStoreName   'Public Store Name
   Dim lmask            'Mask

   'Get important session, folder, and message information
   Set g_oProcInstance = RouteDetails.ProcInstance
   Set msgTarget = g_oProcInstance.Message
   idTargetFolder = EventDetails.FolderID
   idTargetMessage = EventDetails.MessageID
   Set g_oPIMsg = g_oProcInstance.Message
```

```
    WriteToLog 0, "Subject: " & msgTarget.Subject
    Set AMSession = EventDetails.Session
    Set fldrOutbox = AMSession.Outbox
    Set g_oSession = EventDetails.session
    Set g_oMsgIn = RouteDetails.Msg
    WriteToLog 0,g_oMsgIn.Subject
    Set g_oFolder = RouteDetails.Folder
    WriteToLog 0,g_oFolder.Name
    Set g_oAgentAddEntry = g_oSession.currentuser
    'Get sender name. If it does not exist (draft message),
    'get originator name.
    g_bstrMsgSender = g_oMsgIn.sender.Name
    If Err Then
       Err.Clear
       g_bstrMsgSender = g_oMsgIn.Fields.Item(g_PR_CREATOR_NAME)
    End If
    WriteToLog 0,g_bstrMsgSender
    'Save message Subject for Trace reasons
    g_bstrInMsgSubject = g_oPIMsg.subject
    'Trap any untrapped failure
    If Err Then
       WriteToLog 0,"InitializeObjects returned False"
       InitializeObjects = False
    Else
       WriteToLog 0,"InitializeObjects returned True"
       InitializeObjects = True
    End If
    'Release objects
    Set oStore = Nothing
    Set oStores = Nothing

End Function
'+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
'   Name: ReleaseGlobalObjects
'   Area: Utility
'   Desc: Release global objects
'   Parm: None
'   Retn: None
'+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Private Sub ReleaseGlobalObjects()
    on error resume next
    Set g_oMsgIn = Nothing        'Release in reverse order
    Set g_oFolder = Nothing
    Set g_oAgentAddEntry = Nothing
    Set msgTarget = Nothing
    Set fldrOutbox = Nothing
    Set g_oPIMsg = Nothing
    Set Item = Nothing
```

(continued)

```
    Set g_oSession = Nothing
    Set AMSession = Nothing
    Set g_oProcInstance = Nothing
End Sub
</SCRIPT>
```

ROUTING OBJECT LIBRARY

The Exchange Server Routing Object library is provided by Microsoft so that you can create process instances, create and edit routing maps, and track the progress of your process instances. The Routing Object library, which is provided in the file exrtobj.dll, is a hierarchical object library like CDO. In fact, the Routing Object library was built with the expressed intention to be used with the CDO object library. Figure 14-10 shows the objects contained in the Routing Object library.

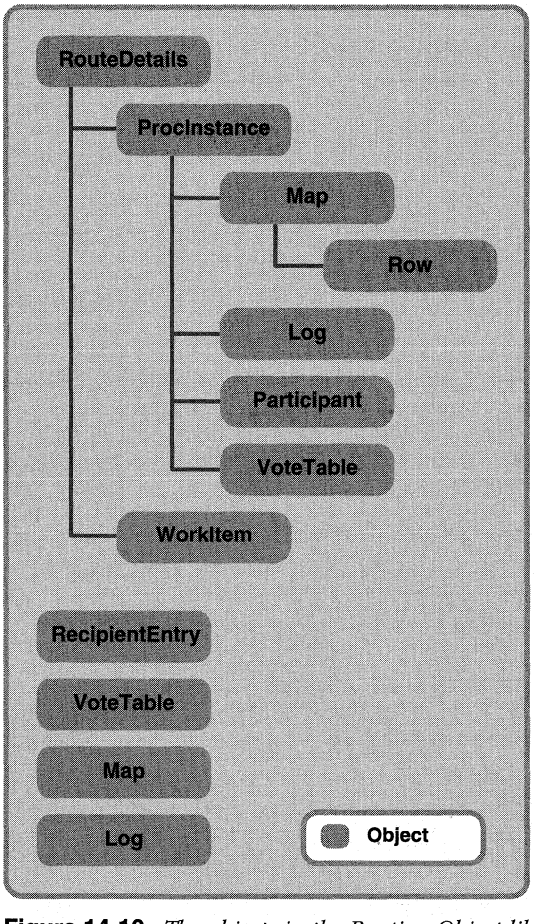


Figure 14-10. *The objects in the Routing Object library.*

With this object library, you can create some interesting applications. To help demonstrate how useful and powerful this object library is, we will take a look at an update to the Agent Install program from Chapter 13. This program has been updated to allow you to edit routing maps and track the state or process instances in a folder. Before looking at the updated Agent Install program, however, you first must take a look at the objects in the library so that you have a firm understanding of their functionality. I'll provide the most important properties and methods you will likely use in your applications. Refer to the Agents.hlp file on the Exchange Server 5.5 Service Pack 1 CD for the full listing of properties and methods for the objects. Also, check out the Exchserv.chm file on the companion CD and the Microsoft TechNet site *http://www.microsoft.com/technet/download/exchange/*.

RouteDetails Object

The RouteDetails object is a top-level object in the library. You should never explicitly create a RouteDetails object; it is passed to the VBScript subroutines that you write for the folder in the same way that the EventDetails object is passed to your VBScript event scripts. Since the routing objects are built on the Event Scripting technology, both the RouteDetails object and the EventDetails object are passed to your VBScript functions.

The RouteDetails object contains a number of properties that you will want to use in your VBScript functions, including the following:

- *ProcInstance property*. This property, when set to an object variable, returns the process instance object currently being executed. You can use this object to retrieve process-specific information, such as the row in the routing map that the process instance is currently executing.
- *Msg property*. This property, when set to an object variable, returns a CDO Message object that corresponds to the message received by the folder for the process instance. In your scripts, this message would be an approval or a rejection, or some other type of status message sent to the folder. This message would have an RUI number representing a currently executing process instance.
- *Folder property*. This property, when set to an object variable, returns a CDO Folder object that represents the current folder where the process instance is running. You can use this Folder object in your script to perform actions on the folder or on messages inside the folder.
- *WorkItem property*. This property returns the WorkItem object for the current process instance.

As you can see, the *RouteDetails* properties return full CDO objects, unlike the EventDetails object, which returns only *EntryID* properties for the message and the folder. Using the *RouteDetails* properties avoids the need to bind to a message or a folder in the Exchange Server store each time a script is run, so your scripts will achieve greater performance.

That said, there is one important object that is not available through the RouteDetails object and available only through the EventDetails object—the CDO Session object, which is the pre-logged-on CDO Session that the script is running under. To retrieve all the objects you want to use in your scripts, you must use both the RouteDetails and EventDetails objects together. The following code shows how to initialize your VBScript objects with all the *RouteDetails* properties:

```
Set g_oSession = EventDetails.session
Set g_oMsgIn = RouteDetails.Msg
Set g_oFolder = RouteDetails.Folder
Set g_oProcInstance = RouteDetails.ProcInstance
```

ProcInstance Object

The ProcInstance object is a top-level object that can be created independently of any other objects in the Routing Object library, or that can be obtained by using the *ProcInstance* property of the RouteDetails object. The ProcInstance object represents a process instance that is a work item and some additional properties for state and map information. When tracking processes in your routing object applications, you will use the ProcInstance object extensively. Some of the important properties and methods for this object include the following:

- *CurrentRow property.* This property returns a Long value that represents the current row that the engine is executing. The value is not the same as the ActivityID of the current row that the engine is processing. To find the ActivityID, you must first scroll through the routing map and then retrieve the specific row the property corresponds to. You can then use the *ActivityID* property of the Row object, which you will see later in this chapter.
- *Log property.* When set to an object variable, this property returns the routing object's Log object. The Log object represents the custom log for a particular process instance. Don't confuse this Log object with the agent log that you saw in the previous chapter. These two logs are stored differently. The Log object in the routing objects should be used to store execution history and auditing information for the process. The Agent log should be used for debugging information about agents as well as general agent information, such as the time the agent completed execution.

Note that if you want to write debugging information to the Agent log from a routing objects VBScript subroutine, you can use the *Script.Response* method in exactly the same way you use it in an Event Agent script.

- *Map property.* This property, when set to an object variable, returns a Map object that represents the map for the current process instance. Since you can have maps for individual items that are different from the default folder map, you should not assume that the default folder map and the process instance map are the same.
- *Message property.* Set this property to an object variable to retrieve the corresponding CDO message for the process instance. You can also set this property to a CDO Message object to tell the Routing Object library which process instance you want to work with—this is the most common use for the *Message* property. You will see how to use this property in the sample application later in this chapter.
- *Participant property.* This property returns the Participant object. The role of the Participant object is to let you quickly resolve custom roles stored in the Exchange Server directory.
- *RUI property.* This property returns a Long value that represents the unique RUI number for this process instance.
- *Terminated property.* This property returns a Long value that represents whether the process instance is terminated—*0* if the process instance is not terminated and is still executing, or *1* if the process instance has hit a Terminate command in the map.
- *Timeout property.* This property returns a Boolean that represents whether the process instance has timed out during a Wait action in a map. Timeout returns *True* if the process instance has timed out, and *False* if it has not timed out. You should use this property and the Wait action together to implement timeouts in your applications. You can then call automatic functions when the timeout occurs to either automatically move the item to the next participant or remind the current participant of the time limit for a response.
- *VoteTable property.* This property returns a VoteTable object that allows you to create and consolidate Outlook voting-button-style messages.
- *Wait property.* This property returns a Long value that represents the time when the process instance will expire. The *Wait* property is used by the engine at run time to determine whether the process instance has timed out. You probably will never use this property directly in your applications but instead will use the *Timeout* property just discussed.

- *Open method.* This method, when it has the *Message* property set and is called on a ProcInstance object, opens the process instance on the specific CDO Message object. You must set the *Message* property and use the *Open* method before attempting to use any of the other routing object properties.
- *Save method.* This method saves any changes you make to the process instance. You must also call the CDO *Update* method on the process instance message to save your changes permanently.

The following code snippet, written in Visual Basic, shows you how to use some of these properties and methods. It assumes you already have a valid CDO Message object, named oMessage, that corresponds to the process instance you are interested in.

```
Set oRTProcInstance = CreateObject("exrt.ProcInstance")
oRTProcInstance.Message = oMessage
oRTProcInstance.Open
msgbox "The current executing row is " & oProcInstance.CurrentRow
if oProcInstance.Terminated = 0 then
   msgbox "The process is not terminated."
Else
   Msgbox "The process is terminated."
End if
Msgbox "The RUI for this process instance is " & oProcInstance.RUI
'This shows how to retrieve the properties that require object
'variables
set oMap = oProcInstance.Map
set oLog = oProcInstance.Log
```

Map Object

The Map object represents the routing map that is evaluated and used by the engine when executing process instances. There must be a default map in every routing folder. This map is copied onto incoming messages when the messages do not already contain maps.

The Map object is a top-level object that can be created independently of other objects in the library, so you can create maps and edit maps without creating a ProcInstance object. The following are the major properties and methods for this object:

- *ActivityCount property.* This property returns a Long value that represents the number of activities in a particular map. You can use this property to scroll through all the activities in a map to search for a specific activity or parameter.

- *Message property.* This property gets or sets the CDO Message object corresponding to the object you want to retrieve or save a map onto. You must set this property before you call the methods on the Map object. You can set this property to an individual message in a folder to retrieve the map stored in the message, or to the event binding message in the folder to retrieve the default map for the folder.
- *CopyTo method.* The *CopyTo* method copies the map from one message to another message. This method takes as its argument the CDO Message object, which corresponds to the process instance you want to copy the current Map object to.
- *DeleteActivity method.* This method takes an ActivityID as its argument and will delete the activity from the map. You must call the *SaveMap* method and the CDO Message object *Update* method to save your changes permanently.
- *DeleteMap method.* This method deletes the entire map from the current CDO Message object.
- *GetRow method.* This method retrieves a specific row from the map. It takes the row number as a Long argument and returns a Row object that corresponds to that row.
- *InsertActivity method.* This method inserts a Row object into the current map. It takes a Long value that indicates the row number preceding the row you will insert. This method also takes a Row object as an argument to indicate the source of the row to insert. The value *–1* tells the library to insert your new row after the last row in the map.
- *OpenMap method.* This method opens the map on the CDO Message object you specify for the *Message* property. You can pass in a Long argument, which takes the value *0* if you want to open the map as read-only, and nonzero if you want to open the map as read/write.
- *SaveMap method.* This method saves the current map on the CDO Message object specified in the *Message* property. Note that you must also change a property on the item and call the CDO Message *Update* method to permanently save the item. If you do not call the CDO *Update* method after calling the *SaveMap* method, your changes will be lost when you destroy the Map object.

The following code snippet shows you how to use some of these methods and properties in your own programs. It assumes you have a valid CDO Message object, named oMessage.

```
Set oRTMap = CreateObject("exrt.map")
oRTMap.Message = oMessage
oRTMap.OpenMap 1    'Read/Write
msgbox "Activity Count is " & oRTMap.ActivityCount
'Retrieve a row by using the GetRow method.
'First, create the Row object.
set oRTRow = CreateObject("exrt.row")
oRTMap.GetRow 1, oRTRow    'Get the first row
'Change the Flags property on the row
oRTRow.Flags = 2
'Need to save the map now
oRTMap.SaveMap
'Need to change a property on the message to have the
'Update method work correctly
oMessage.Subject = oMessage.Subject & " "
'Need to call the CDO Message Update method to persist changes on
'the message
oMessage.Update
```

Row Object

The Row object represents a single row in your map. You can create a Row object independent of the other objects in the library. After creating a Row object, you should set its properties and add it to your map. If you are retrieving a row from an existing map, to store the row, you must pass in a variable that corresponds to your Row object. The following are the most important properties and methods for this object:

- *Action property.* This property takes a String value and can either set or retrieve the action for the current Row object. This action can be the name of an intrinsic action or of a script subroutine you create in VBScript.
- *ActivityID property.* This property is a Long value that contains the unique number that identifies the row in the map. Usually, you start ActivityIDs in a new map at 100 and increment according to your preference for numbering the subsequent rows.
- *Flags property.* This property is a Long value that contains a flag indicating whether the *Action* property is an intrinsic action or a VBScript-implemented action. The possible values for this property are *0* for an intrinsic action and *2* for a VBScript action.
- *CopyTo method.* This method takes a Row object as its argument, and then copies the current row to the new Row object.

- *GetArgs method.* This method returns the parameters for a Row object. The parameters are returned in a string array along with a Long variable that indicates the number of parameters for an action. See the next code snippet to learn how to use this method.
- *SetArgs method.* The *SetArgs* method is the opposite of *GetArgs*. Instead of retrieving the parameters for the Row object, the *SetArgs* method sets the parameters for the Row object. You must pass in a Long value that is the number of parameters for the Row object as well as a string array of those parameters.

The following code snippet shows you how to use the Row object. Note that you must convert to the correct format any arguments you pass to the methods and properties. If a method is expecting a Long value, pass a Long value. This example assumes you already have a valid Map object set to oRTMap.

```
'This example creates a row, fills it in, and adds it to a map
Set oRTRow = CreateObject("exrt.row")
Dim arrParameters(2)        'Two parameters
arrParameters(1) = "Test"
arrParameters(2) = "Test2"
'If the parameters are numbers, use CLng.
'See the AgentInstall update later in this chapter for more
'information.
oRTRow.SetArgs 2, arrParameters

'Use CLng if not already a Long
oRTRow.ActivityID = 100
'Use CStr if not a string
oRTRow.Action = "My Custom Action"
'Use CLng if not already a Long
oRTRow.Flags = 2
'Insert the Row into the Map
oRTMap.InsertActivity -1, oRTRow
oRTMap.SaveMap
'Then update the message, as shown in the previous code snippet
```

Log Object

The Log object allows you to log activities that execute in your map. This object stores its log differently from the way logs are stored in the Event Agent log. The only way you can retrieve the Log object is by using the *Log* property on the ProcInstance object. You cannot create a separate instance of this object. The methods of the Log object are shown on the following page.

- *AddLogEntry method.* This method adds a new log entry to the current log of the selected process instance. This method takes the application name as a string (referred to as the name ID), which you can use to identify custom activities or functions as a string, the date/time as a Long value, and the description you want to store as a string. You probably will use this method to store route-specific information for a particular process instance. This log is useful for storing audit trails and failures in a particular process.
- *GetLogEntry method.* This method retrieves a log entry from the current log of the process instance. It shares the same properties as the *AddLogEntry* method, but it returns them instead of adding them. The parameters for these two methods differ as well: the *GetLogEntry* method has an extra parameter, which is for the LogIndex.
- *OpenLog method.* This method opens the log so that you can retrieve or add items to it.
- *SaveLog method.* This method saves your changes to the current log.

The following code snippet shows you how to insert an item into the log. The code assumes that you have a valid ProcInstance object named oRTProcInstance.

```
Set oRTLog = oRTProcInstance.Log
oRTLog.AddLogEntry "MyApp", "MyNameID", CLng(10/1/1998), "MyDescrip"
'Save the log
oRTLog.SaveLog
```

Participant Object

The Participant object provides a way to refer to and manipulate addresses (which can be actual e-mail addresses or roles) in your routes and resolve roles to actual addresses. For example, you would use the Participant object to find out who the manager or expense approver for a certain user is by passing in the address of that user. The Participant object would then return to you the address of the person who performs the role you specified. The three important methods of this object include the following:

- *RoleName method.* You pass the role name you want to resolve as a string to this method. For example, to resolve the Expense Approver role for a user, you would pass Expense Approver.

- *MemberName method.* You pass this method a string containing either the unique address or unique name of a person. This string should contain the information of the person for whom you want to find the role performer associated with the role name you specified in the *RoleName* method. For example, if you wanted to find the expense approver for Frank Lee, you would pass, as a string to this method, either the name Frank Lee or the e-mail address of Frank Lee.
- *ResolveRole method.* This method returns to you the address of the person who performs the role for the member you specified. You must set a variable to this method to retrieve the address. For example, if Tom Rizzo was the expense approver role performer for Frank Lee, the e-mail address of Tom Rizzo would be returned by *ResolveRole*.

The following sample code shows you how to use this object in the VBScript subroutines for your routes. It assumes that you already have a valid ProcInstance object named oRTProcInstance.

```
Set oParticipant = oRTProcInstance.Participant
'This can also be Manager or another custom RoleName you make
oParticipant.RoleName = "Expense Approver"
'Or you could pass the address
oParticipant.MemberName = "Thomas Rizzo"
'Get the address of the expense approver
ExApproverAddress = oParticipant.ResolveRole
```

VoteTable Object

The VoteTable object allows server-based applications to create Outlook voting-button messages. The Exchange Server Routing Objects do not need to rely on the Outlook object model to do this; they include the functionality to create them. Being able to create voting buttons makes it easy for your Outlook users to select custom responses, such as Approve or Reject, for your routed items. The VoteTable object can also be used to consolidate voting button response messages in the original process instance. This consolidation saves time since you do not have to write this code yourself to consolidate the responses. Furthermore, this object updates the Tracking tab in Outlook without needing code from you. Remember that you can use this object without using the rest of the Routing Objects functionality. This means you can add voting capabilities to any Exchange Server application using this object, even if your application has nothing to do with workflow.

The VoteTable object can be created as a separate object or retrieved by using the *VoteTable* property of a ProcInstance object. The VoteTable object is used in conjunction with the RecipientEntry object, which is discussed later in this chapter. The following list describes the properties and methods for the VoteTable object:

- *Count property.* This property returns the number of recipient entries in the VoteTable object.
- *Item property.* By passing in a number specifying an index, this property will return the corresponding, existing RecipientEntry object in the VoteTable.
- *PIMessage property.* When creating a stand-alone VoteTable object, this property must be set to the CDO Message object, which corresponds to the process instance for which you want to check recipients' votes.
- *AddButtons method.* This method adds voting buttons to your message. The two parameters you must pass to this method are the CDO Message object, which is the message you want to add the voting buttons to, and a string of the voting options, separated by commas, as in "*approve, reject, undecided*".
- *ConsolidateResponse method.* This method consolidates the responses for a voting button message received by the folder. This method takes two parameters, the first being a CDO Message object, which is the voting button response message. The second parameter is a Boolean specifying whether to automatically add a RecipientEntry object to the VoteTable for the user who responded to the voting button message, if one doesn't already exist for the user.
- *Save method.* This method saves your changes to the VoteTable object.

The following code example shows you how to create a voting button message. It assumes you already have a valid CDO Message object called oMessage.

```
Set oRTVoteTable = CreateObject("exrt.VoteTable")
'You could also use the VoteTable property on the ProcInstance
'object to get a VoteTable object.
'Create the buttons.
oRTVoteTable.AddButtons oMessage, "approve,reject,undecided"
'Send the message
oMessage.Send
```

The next code snippet shows you how to consolidate voting button responses received in a folder. This example assumes you have a valid CDO Message object in oMessage.

```
'Use the ProcInstance VoteTable
set oRTVoteTable = oRTProcInstance.VoteTable
'oMessage is the voting response message, True means to autoadd
oRTVoteTable.ConsolidateResponse oMessage, True
oRTVoteTable.Save
```

RecipientEntry Object

The RecipientEntry object is always used to track the response of a recipient in the route and the date that recipient responded. The RecipientEntry object is typically used in conjunction with the VoteTable object. The following properties are available for the RecipientEntry object:

- *Date property.* This property is the date the recipient sent a response for the process instance.
- *Recipient property.* This property is the name of the recipient who responded.
- *Status property.* This property contains the recipient's response, such as Approve or Reject.

The following code shows you how to create and fill in a RecipientEntry object. The code will also show you how to add this RecipientEntry object to the VoteTable object. This code assumes you have a valid oRTProcInstance object.

```
Set oRTVoteTable = oRTProcInstance.VoteTable
Set oRecipientEntry = CreateObject("exrt.RecipientEntry")
oRecipientEntry.Recipient = "My Name"
oRecipientEntry.Date = "11/1/1998 10:00 AM"
oRecipientEntry.Status = "Approve"
oRTVoteTable.ConsolidateResponse oRecipientEntry, False
oRTVoteTable.Save
```

WorkItem Object

The WorkItem object can be retrieved only as a property of the RouteDetails object. The WorkItem object can represent a new item in the folder before a related process instance for the item is found, or a new item in the folder until that item is turned into a process instance. Most of the time, your applications will not call the methods and properties of the WorkItem object directly. Rather, the routing engine will call the methods. The only exceptions to this are the following:

- *Item property.* This property specifies the CDO Message object you want to assign to the WorkItem object. After assigning the CDO Message object, you can use the methods of the WorkItem object with that CDO Message object.

- *ItemConsolidate method.* This method merges the message content of one CDO Message object with another CDO Message object. To specify the target message, set the *Item* property for the WorkItem object to it. The *ItemConsolidate* method takes three parameters, the first being an array of MAPI property tags that you want to merge with the target message. These properties can be attachments, the message body, or your own custom properties. The second parameter is the CDO Message object, which is the source message you want to consolidate your specified properties with. The third argument, or parameter, is a Boolean value; you specify *True* if you want to append the content to the target message, or *False* if you want to overwrite any existing content in the target message's properties. The next code snippet shows you how to use the *ItemConsolidate* method.
- *EmbedMsg method.* This method makes it easier for you to embed the CDO Message object into the CDO Message object you specify in the *Item* property for the WorkItem object. The only parameter this message takes is the source CDO Message object that you use to embed in the WorkItem CDO Message object.

To illustrate how these methods and properties work, the following sample code sets a WorkItem object to a CDO Message object. It then embeds another message in the existing CDO Message. The code also consolidates all attachments from another message into the WorkItem CDO Message object. This sample assumes you have a RouteDetails object.

```
'Get the WorkItem object
Set oWorkItem = RouteDetails.WorkItem
'Set the Item property to the current process instance
oWorkItem.Item = oProcInstanceMessage
'Embed another message into the oProcInstanceMessage
oWorkItem.EmbedMsg oAnotherMessage
'Consolidate all the attachments from another message into
'the WorkItem message.
'Create an array of property IDs.
PropArray = Array(&HE13000D)    'For message attachments
'Append items by selecting True
oWorkItem.ItemConsolidate PropArray, omsgSource, True
```

UPDATED AGENT INSTALL APPLICATION

Now let's take a look at the updated Agent Install application. I added some new functionality that allows you to view and edit maps, copy the maps into other folders, and track process instances in folders. The code in this section uses objects in

the Routing Object library and the Event Config library. I had to make a number of enhancements to make the program work with process instances and routing maps.

Overview of the Updated Agent Install Application

The enhancements for the Agent Install application fall into four main areas:

- Agent enhancements, such as creating and deleting routing agents
- Routing map enhancements, such as the ability to edit and delete maps
- Process instance enhancements, such as the ability to view process instances and see which row in the map the process instance is currently executing
- User interface enhancements, such as being able to see which folders contain agents, routing maps, or nothing

Now we'll examine the code that implements each type of enhancement.

Agent Enhancements

When you start the new version of Agent Install, you'll notice more options to choose from on the main page. These new options, shown in Figure 14-11, include viewing the default map for the folder, deleting a map, and viewing the processes executing in the folder. System enhancements also include the counting of messages contained in the folder, which will help you estimate how long opening all processes will take when you click the View Processes button.

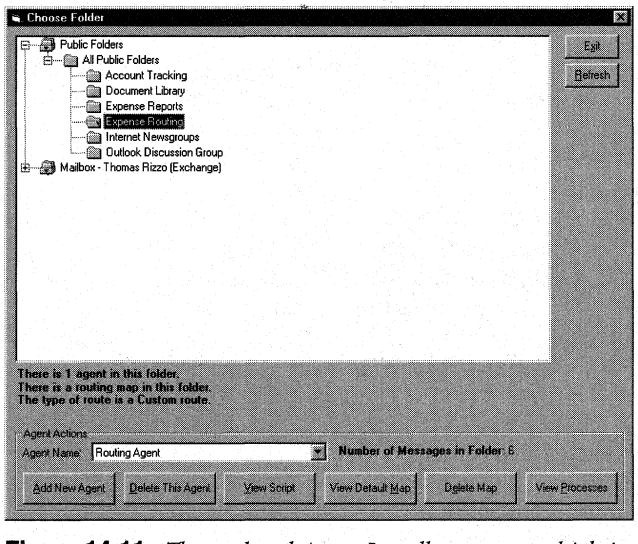


Figure 14-11. *The updated Agent Install program, which includes routing object functionality.*

Detecting Default Routing Maps in a Folder

When you select a folder that does not contain a routing map, the enhancements, with the exception of viewing processes, will be disabled. The functionality to detect a default routing map in the folder is implemented in the *RefreshAgentCount* subroutine. This subroutine is able to detect not only agents in the folder but also maps, as shown in the following code listing:

```
Public Sub RefreshAgentCount()
On Error GoTo RefreshAgentCount_Err
   GetMessageCount

   Dim bRouteMap
   Dim bRouteType
   Dim intFoundRoutingMap, intFoundRoutingAgent
   intFoundRoutingMap = 0
   intFoundRoutingAgent = 0
   lblMap.Caption = ""
   Set oRouteBinding = Nothing
   Set oRTMessageMap = Nothing
   cmdViewMap.Enabled = False
   cmdDeleteMap.Enabled = False
   cmdAddAgent.Enabled = True
   cmdDeleteAgent.Enabled = True
   cmdViewScript.Enabled = True

   Set oBoundFolder = oEvents.BoundFolder(oFolder, True)
   Set oBindings = oBoundFolder.Bindings
   If oBindings.Count = 1 Then
      lblAgentCount.Caption = "There is 1 agent in this folder."
   Else
      lblAgentCount.Caption = "There are " & oBindings.Count & _
         " agents in this folder."
   End If
   comboAgents.Clear
   'Clear the binding
   If oBindings.Count > 0 Then
      For Each oBinding In oBindings
         comboAgents.AddItem CStr(oBinding.Name)
         'Assume there is one binding in the folder
         'for routing
         Set oRouteBinding = oBinding
      Next
   End If
      'Check for routing maps as well
      On Error Resume Next
      If Err.Number = 0 Then
         'Found one
```

```
            Set oHidden = oFolder.HiddenMessages
            For Each oHide In oHidden
               Err.Clear
               tmpTest = oHide.Fields("RouteAgent")
               If Err.Number = 0 Then
                  intFoundRoutingAgent = 1
               End If

               Err.Clear
               bRouteMap = oHide.Fields("RouteMap")
               If Err.Number = 0 Then
                  cmdViewMap.Enabled = True
                  cmdDeleteMap.Enabled = True
                  intFoundRoutingMap = 1
                  Set oRTMessageMap = oHide
               End If

               Err.Clear
               bRouteType = oHide.Fields("RouteType")
               If Err.Number = 0 Then
               End If
            Next
            If intFoundRoutingMap Then
               lblMap = "There is a routing map in this " & _
                  "folder." & vbLf & "The type of route " & _
                  "is a " & bRouteType & " route."
            End If
         End If
      Err.Clear
      comboAgents.ListIndex = 0
   Exit Sub

   RefreshAgentCount_Err:
      If Err.Number = -2147221245 Then
         MsgBox "Outlook Calendar folders are not supported.", _
            vbOKOnly + vbExclamation, _
         "Select Folder"
         lblAgentCount.Caption = "Not Supported"
         lblMap.Caption = ""
         comboAgents.Clear
         cmdAddAgent.Enabled = False
         cmdDeleteAgent.Enabled = False
         cmdViewScript.Enabled = False
         Exit Sub
      End If
   End Sub
```

As you can see in the code, to detect default routing maps in the folder, we first must get the HiddenMessages collection from the CDO Folder object. All default routing maps are stored as hidden messages with a special property, so after we retrieve the HiddenMessages collection, all we need to do is traverse it to look for the *RouteMap* custom property.

In the code, we also look for the *RouteType* property on any hidden message. Both the Routing Wizard sample application and this application insert a string to tell the user the route type contained in the folder. In the Routing Wizard application, the only possible values are *Sequential* and *Parallel*. In this application, when we modify a map created by the Routing Wizard, the *RouteType* property changes to Custom so that you know it is not the original map.

If a default routing map is found in the folder, some of the other command buttons, such as the one for viewing the default map, are enabled. The *RefreshAgentCount* subroutine also sets some variables that will be used throughout the rest of the application, such *oRTMessageMap*, which correspond to the hidden message containing the default message map as well as to *oRouteBinding*, which stores the binding for the folder. Recall from earlier in the chapter that the default map is copied onto all incoming messages that do not correspond to a process instance in the folder, or onto items that do not currently contain a *RouteMap* property with an ad-hoc routing map.

Adding New Agents That Already Have a Routing Map in the Folder

Now that the code can detect default routing maps in the folder, the program has to be able to detect when a user is adding an agent to the folder when a routing map is in the folder—you do not want users to add multiple agents to a folder if they want to use the folder for routing. Instead, you want in the folder only one agent, which fires on all the events. This agent should also contain the default script for the custom actions for the routing map.

To prevent users from adding an arbitrary number of agents to a folder with a routing map, the *cmdAddAgent_Click* subroutine had to be updated to detect the routing map. A snippet of this code is shown here:

```
'Check for message map
If Not (oRTMessageMap Is Nothing) Then
   'Check for existing agent
   If Not (oRouteBinding Is Nothing) Then
      MsgBox "An Agent already exists in this folder with a " & _
         "Routing Map.  You cannot add another one." _
         , vbOKOnly + vbExclamation, "Add Agent"
      Exit Sub
   Else
      result = MsgBox("There is already a Routing Map in " & _
         the folder. This application can" _
```

```
            & " only create Routing Agents in folders that " & _
            have Routing Maps already installed." _
            & vbLf & "Do you want to create a Routing Agent?", _
            vbYesNo + vbQuestion, "Routing Map Found")
         If result = vbYes Then
            MsgBox "You must now select a script to associate " & _
               "with your Routing Map and Agent. Please use" _
               & " the following dialog box to select a script.", _
               vbOKOnly + vbInformation, "Add Agent"
            intCaller = 2
            Load frmSelectScript
            frmFolders.Visible = False
            frmSelectScript.Visible = True
            Exit Sub
         Else
            'They said no
            Exit Sub
         End If
      End If
   End If
```

This code first checks to see whether *oRTMessageMap* exists. If it does, a default routing map is in the folder. Then the code checks to see whether *oRouteBinding* exists. If *oRouteBinding* does exist, a routing agent is in the folder. This code stops the user from adding another agent to the folder.

If only the default message map exists in the folder without an agent, the code prompts the user to select a script, which the system will associate with the new agent it will create by opening the frmSelectScript form. After the user has selected the script to associate with the new routing agent in the folder, the code creates a new agent that uses the default message map in the folder, as shown in this code from the MSRTVars module for the Agent Install program:

```
Public Sub CreateRoutingAgent_MapExists(otmpFolder)
   Set otmpEvents = oEvents
   Set otmpBoundFolder = otmpEvents.BoundFolder(otmpFolder, True)
   Set otmpBindings = otmpBoundFolder.Bindings
   Set otmpBinding = otmpBindings.Add
   otmpBinding.Name = "Routing Agent"
   otmpBinding.Active = MSAgentActive
   otmpBinding.EventMask = MSScheduledEvent + MSNewItemEvent + _
      MSChangedItemEvent + MSDeletedItemEvent
   otmpBinding.HandlerClassID = MSRoutingObjectsHandlerID
   otmpBinding.SaveChanges

   Set otmpSchedule = otmpBinding.Schedule
   otmpSchedule.Interval = 60
```

(continued)

```
    otmpSchedule.Type = MSHourlyAgent
    otmpSchedule.Days = MSMonday + MSTuesday + MSWednesday + _
       MSThursday + MSFriday + MSSaturday + MSSunday
    otmpSchedule.StartTime = MSAllDayStart
    otmpSchedule.EndTime = MSAllDayEnd
    otmpBinding.SaveChanges

    'Get the new binding message
    Set oSaveAsMessage = CDOClass.Session.GetMessage( _
       otmpBinding.EntryID, Null)

    'Add the RouteAgent property
    oSaveAsMessage.Fields.Add "RouteAgent", VT_BOOL, True
    bstrEventScript = OpenScriptFile(strFileLocation)
    oSaveAsMessage.Fields.item(PR_EVENT_SCRIPT) = bstrEventScript
    oSaveAsMessage.subject = "Routing Agent"
    oSaveAsMessage.Update
    otmpBinding.SaveCustomChanges oSaveAsMessage
    otmpBinding.SaveChanges
    otmpBoundFolder.SaveChanges
    MsgBox "Successfully created new Agent with the existing " & _
       "Routing Map in Folder.", vbOKOnly + _
       vbInformation, "Create Agent"
End Sub
```

The only difference between this code and the code for creating event agents, which we looked at earlier, is that this code adds a custom property to the agent, named *RouteAgent*, which identifies the agent as a routing agent. Other than that, all this code should be familiar to you from the previous chapter.

Deleting an Agent with a Default Routing Map in the Folder

Now that we have taken care of adding a new agent when a default routing map is in the folder, what do we do about deleting an agent when the same condition exists? We could ignore the routing map in the folder and just delete the agent. But to give the user more control, after deleting the agent, we should prompt the user about whether she also wants to delete the default routing map in the folder.

Implementing this functionality is quite easy. Because we have the variable *oRTMessageMap*, which is a CDO Message object containing the default map, all we need to do is call the *Delete* method on *oRTMessageMap* to delete the map, as shown here:

```
Private Sub cmdDeleteAgent_Click()
On Error GoTo cmdDeleteAgentClick_Err
If comboAgents.Text <> "" Then
   'Agent is selected
```

```
   response = MsgBox("Are you sure you want to delete the " & _
      comboAgents.Text & " agent?", vbQuestion + vbOKCancel, _
      "Delete Agent")
   If response = vbOK Then
      'Delete agent
      For Each oBinding In oBindings
         If comboAgents.Text = CStr(oBinding.Name) Then
            Exit For
         End If
      Next
      oBindings.Delete oBinding
      oBoundFolder.SaveChanges
      MsgBox "Agent Successfully Deleted.", vbInformation + _
         vbOKOnly, "Delete Agent"
      'Check for map as well
      If Not (oRTMessageMap Is Nothing) Then
         result = MsgBox("There is a Routing Map in this " & _
            folder. Do you want to delete it as well?", _
            vbYesNo + vbQuestion, "Routing Map Found")
         If result = vbYes Then
            oRTMessageMap.Delete
         End If
      End If
      RefreshAgentCount
   Else
      MsgBox "The Agent will not be deleted.", vbInformation + _
         vbOKOnly, "Cancel Deletion"
   End If
   Else
   MsgBox "You must first select an agent.", vbExclamation + _
      vbOKOnly, "No agent selected."
End If
Exit Sub

cmdDeleteAgentClick_Err:
   If Err.Number = &H46 Then
      MsgBox "You do not have permission to delete agents in " & _
         "this folder. Please choose another folder.", vbOKOnly + _
         vbCritical, App.Title
   Else
      Call CDOClass.MapiErrorHandler("cmdAddAgent execution " & _
         "in frmFolders while trying to access folder " & _
         "information.")
   End If
Exit Sub
End Sub
```

Routing Map Enhancements

The enhancements to this application include new code that implements routing map editing functionality. This functionality includes the ability to view routing maps, edit them, parse their script for routing functions, and save them to other folders.

The user interface for viewing and editing the routing maps is a grid control in Visual Basic. This grid control, shown in Figure 14-12, makes it easy for you to scroll through and look at your maps as well as add or delete rows and columns to maps.

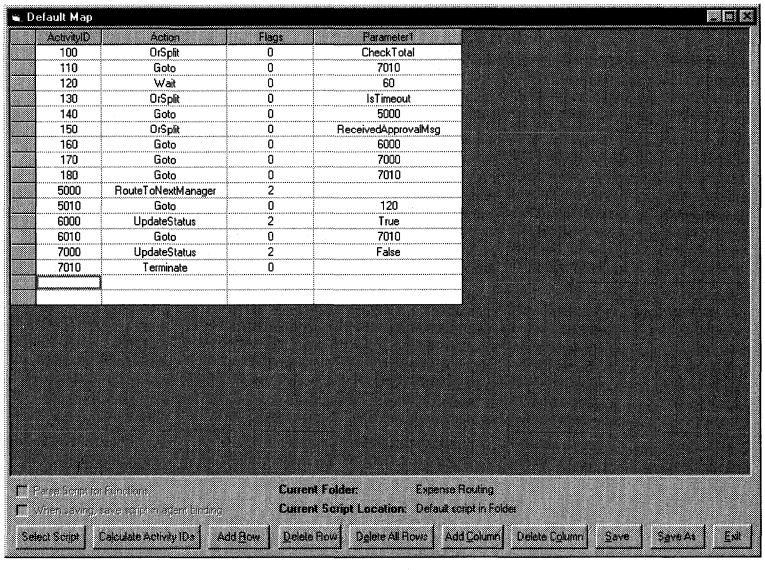


Figure 14-12. *The grid control in the updated Agent Install program to view and edit routing maps.*

I do not want to dive into the ins and outs of using the grid control, but I will tell you that it is very useful. You should consider learning more about it either by examining the code for this program or by reading the online help.

Viewing a Map

When you click on the view default map command on the main screen of the application, you open the default map form called frmRouting. This form contains a lot of code to implement the map viewing, editing, and save features in the application. The first feature we'll look at is the map viewing functionality.

As we saw with the Routing Object library, we can retrieve the activity count and individual rows as well as other information about a routing map in a folder. The viewing capabilities of the Agent Install application are implemented through these routing objects. This is the code for the *Form_Load* function:

```
Private Sub Form_Load()
Dim arrArgs As Variant
   MAXCOLUMNS = 5
   COLCONSTANT = 5
   IsDirty = 0
   cTab = Chr(9)
   'Clear the grid
   flexMap.Clear
   'Add the headers to the grid
   FormatColumnHeaders
   'Fill intrinsic actions array
   FillIntrinsicActionsArray
   'Populate the commands combo box.
   'This should also pull from script.
   PopulateCombo
   'Set bValidScript to indicate no script
   bValidScript = 0
   If oRouteBinding Is Nothing Then
      lblScriptLocation.Caption = "None"
   End If
   bstrEventScript = ""
   'Fill in folder name
   lblCurrentFolder.Caption = oFolder.Name

   'Load the map and the activities into the grid
   Set oRTMap = CreateObject("exrt.map")
   If intMapViewer = 0 Then
      oRTMap.Message = oRTMessageMap
   ElseIf intMapViewer = 1 Then
      oRTMap.Message = oRTMessageMap
   End If
   oRTMap.OpenMap TBL_OPEN_READWRITE
   lActivityCount = oRTMap.ActivityCount
   For i = 1 To oRTMap.ActivityCount
      Set oRTRow = CreateObject("exrt.row")
      oRTMap.GetRow i - 1, oRTRow
      flexMap.AddItem ""
      flexMap.TextMatrix(i, 1) = CStr(oRTRow.ActivityID)
      flexMap.TextMatrix(i, 2) = oRTRow.Action
      flexMap.TextMatrix(i, 3) = CStr(oRTRow.Flags)
      'Figure out how many parameters there are
      arrArgs = Array()
      oRTRow.GetArgs 1, arrArgs
      If (UBound(arrArgs) + COLCONSTANT) > MAXCOLUMNS Then
         flexMap.Cols = UBound(arrArgs) + COLCONSTANT
         MAXCOLUMNS = UBound(arrArgs) + COLCONSTANT
         FormatColumnHeaders
      End If
```

```
            For tmpCounter = 0 To UBound(arrArgs)
               tmpArg = arrArgs(tmpCounter)
               flexMap.TextMatrix(i, tmpCounter + (COLCONSTANT - 1)) = _
                  CStr(tmpArg)
            Next
         Next
         If intMapViewer = 1 Then
            'Point to the current row
            tmpCurrentRow = oRTProcInstance.CurrentRow
            flexMap.Col = 0
            'If it can't figure out the current row,
            'skip to the next one
            On Error Resume Next
            flexMap.Row = tmpCurrentRow
            Set flexMap.CellPicture = LoadPicture("Arw05rt.ico")
            flexMap.CellPictureAlignment = flexAlignRightCenter
            flexMap.Col = 1
            flexMap.Row = tmpCurrentRow

            'Disable SaveAs
            cmdSaveAs.Enabled = False

            'Check to see if there is a RouteBinding object.
            'If there is, enable Open Script.
            If Not (oRouteBinding Is Nothing) Then
               cmdOpenScript.Enabled = True
            Else
               cmdOpenScript.Enabled = False
            End If
         End If
      End Sub
```

First the code clears the grid so that no previous map information is shown. Then the code calls the *FormatColumnHeaders* function, which places the names of the columns such as ActivityID at the top of the grid. *FillIntrinsicActionsArray* is called next, and then *PopulateCombo* is called. These two functions are not critical—all they do is fill an array with the name of the intrinsic map actions and then populate a combo box with those actions. However, they do provide convenient features for users of the application, who can quickly select the different actions for the map by using the newly populated combo box.

The interesting part of the code occurs after the *FillIntrinsicActionsArray* and *PopulateCombo* functions are called. Some variables are set so that the application knows the user hasn't yet selected a valid script. Then, after more initialization, the grid is filled in with the rows from the map. This is accomplished by creating a Map object and setting its *Message* property to the correct CDO Message object. Why is there an *intMapViewer* variable? This variable tells the frmRouting form which map to view: the default map in the folder, or a specific map on a particular message. You

will see how this is implemented in the "Process Instance Enhancements" section of this chapter.

After setting the *Message* property, we need to open the map by using the *OpenMap* method and specifying that we want to open the map as read/write. Then we use the *ActivityCount* property to find out how many activities are in the map. A For...Next loop is created using the *ActivityCount* property as the control variable. This loop creates a new Row object. The *GetRow* method is called on the Map object; the row number is passed in as well as the new Row object in which the row will be placed. Specific information is pulled from the returned Row object using the *ActivityID*, *Action*, and *Flags* properties for the Row object.

To retrieve the parameters for the rows, the program creates a new array variable. Then it gets the parameters by passing the array variable to the *GetArgs* method on the Row object. Now that the program has the rows, it needs to see whether the number of parameters returned is greater than the number of columns in the grid control. If it is, the code adds the required number of new columns and reformats the column headers.

Let's skip the If statement, which checks to see whether intMapViewer = 1, for right now. We will look at this later in the chapter, because this section of the code implements custom functionality for process instances rather than the default routing map in the folder.

Editing the Map

The code for editing the map, which involves adding or deleting rows and columns and changing the text in a specific cell, does not have much routing object functionality because it's mostly automation of the grid control. For this reason, this functionality will not be covered. However, you can look at the source code for the application on the companion CD to see how it is implemented.

Selecting and Parsing Scripts

To allow users of the application to change the script they want to use with the routing map, the application allows selection of a script from the file system. It also allows the user to decide whether to save the script in the agent as the default script for the folder. This option is available only when an agent is actually in the folder, and not when just a routing map is in the folder without an agent. The application will also allow you to parse the script for route functions.

To select a script, the user clicks the Select Script button on the map viewer. The application detects whether an agent is already in the folder and asks the user whether he wants to use the default script already in the folder. If the user chooses not to, the application launches a separate form that allows the user to select a script from the file system, as shown in Figure 14-13. Once the user selects the script, the application tries to read the script and prompts the user about whether he wants to save the script in the agent binding or try to parse the script for functions.

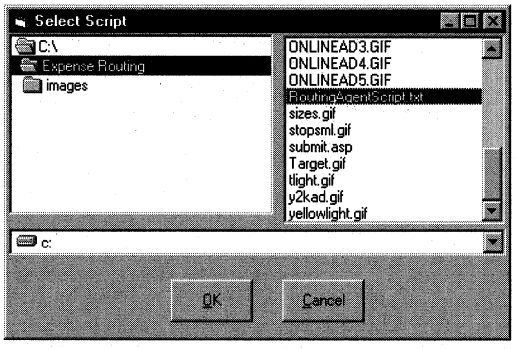


Figure 14-13. *The Select Script dialog box allows users to find and select scripts from the file system.*

The code that implements the Select Script button is shown here:

```
Private Sub cmdOpenScript_Click()
   If intMapViewer = 1 Then
      MsgBox "You can load only the default script for the " & _
         "folder. This script will be loaded. You can " & _
         "then parse the script for functions.", vbOKOnly _
         + vbInformation, "Select Script"
      strFileLocation = "Message"
      checkParseScript.Value = vbUnchecked
      checkParseScript.Enabled = True
      checkSaveScript.Value = vbUnchecked
      checkSaveScript.Enabled = False
      lblScriptLocation.Caption = "Default script in Folder"
      Exit Sub
   End If
   If Not (oRouteBinding Is Nothing) Then
      'There is actually an agent in the folder
      result = MsgBox("Do you want to use the default script " & _
         "in the folder?", vbYesNo + vbInformation, _
         "Use Default Script")
      If result = vbYes Then
         strFileLocation = "Message"
         'Enable only Parse Script, and remove old script actions
         checkParseScript.Value = vbUnchecked
         checkParseScript.Enabled = True
         checkSaveScript.Value = vbUnchecked
         checkSaveScript.Enabled = False
         lblScriptLocation.Caption = "Default script in Folder"
         Exit Sub
      End If
   End If
   frmRouting.Visible = False
   intCaller = 1
   frmSelectScript.Visible = True
End Sub
```

When the user clicks the OK button in the Select Script dialog box, frmSelectScript, the following code is called:

```
Private Sub cmdOK_Click()
   'Check for script selected
   If fileCurFile.FileName = "" Then
      MsgBox "You must first select a script for the agent.", _
         vbExclamation + vbOKOnly, "No Script Selected"
      Exit Sub
   End If
   tmpFileLocation = fileCurFile.Path & "\" & fileCurFile.FileName
   'Try to open the script
   bstrScriptTest = OpenScriptFile(tmpFileLocation)
   If bstrScriptTest <> "" Then
      strFileLocation = tmpFileLocation
      If intCaller = 1 Then
      'Remove old script functions
      frmRouting.checkParseScript.Value = vbUnchecked

         'Only for frmRouting
         If Not (oRouteBinding Is Nothing) Then
            result = MsgBox("When saving your map, would " & _
               "you like to save this script with it?", _
               vbYesNo, "Save Script with Map")
            If result = vbYes Then
               frmRouting.checkSaveScript.Value = vbChecked
            Else
               frmRouting.checkSaveScript.Value = vbUnchecked
            End If
         Else
            frmRouting.checkSaveScript.Value = vbUnchecked
         End If
         result = MsgBox("Would you like the program to try " & _
            "and parse the script for functions?", _
            vbYesNo, "Parse Script")
         If result = vbYes Then
            frmRouting.checkParseScript.Value = vbChecked
         End If
         'Mark that we have a valid script
         bValidScript = 1
         frmRouting.checkParseScript.Enabled = True
         If Not (oRouteBinding Is Nothing) Then
            frmRouting.checkSaveScript.Enabled = True
         Else
            frmRouting.checkSaveScript.Enabled = False
         End If
         'Replace the \\ in the location
         strFileLocation = Replace(strFileLocation, "\\", "\", _
```

(continued)

```
            1, 1)
         'Display it in the form
         frmRouting.lblScriptLocation = strFileLocation
         If Not (oRouteBinding Is Nothing) Then
            frmRouting.MakeDirty
         End If
      End If
      If intCaller = 2 Then
         CreateRoutingAgent_MapExists oFolder
         frmFolders.RefreshAgentCount
      End If
      Unload Me
   End If
End Sub
```

One of the neat aspects of the application is that it parses the script for route functions because it can easily find them. How? Recall that all route functions must begin with the word *Route_* before the function name. All the code has to do is search for the script file for Sub Route_, and then grab the characters appearing between the location for Sub Route_ and a Return or an open parenthesis, which would indicate that an argument list is coming next. The application then populates the same combo box populated by the intrinsic script actions so that you can easily select either an intrinsic action or a custom action from the script without having to look at reference materials or the script itself. The code for this is shown here:

```
Sub ParseScriptforFunctions(tmpFileLocation)
   'Return value is array of Sub and Function names found in
   'the script.
   'If none are found, returns empty.
   Dim arrSubNames(100)
   arrSubNames(0) = ""
   arrCounter = 0
   If tmpFileLocation = "Message" Then
      Set oMessage = oSession.GetMessage( _
         oRouteBinding.EntryID, Null)
      bstrEventScript = oMessage.Fields.item(PR_EVENT_SCRIPT)
   Else
      bstrEventScript = OpenScriptFile(tmpFileLocation)
   End If
   found = 1
   Do While (found <> 0 Or found <> Null)
      found = InStr(found, bstrEventScript, "Sub Route_")
      If found <> 0 Then
         'Got one, now look for the name after the _.
         'First look for a ( after found.
         tmpUnderScore = 0
         tmpUnderScore = InStr(found, bstrEventScript, "_")
         If tmpUnderScore <> 0 Then 'Is should never be 0
            'Now start picking off characters until you
```

```
            'hit a left parenthesis or a new line.
            'Move to next character.
            tmpCurLocation = tmpUnderScore + 1
            tmpChar = Mid(bstrEventScript, tmpCurLocation, 1)
            tmpSubName = ""
            Do While tmpChar <> "(" And tmpChar <> Chr(13)
               tmpSubName = tmpSubName & tmpChar
               tmpCurLocation = tmpCurLocation + 1
               tmpChar = Mid(bstrEventScript, tmpCurLocation, 1)
            Loop
            arrSubNames(arrCounter) = tmpSubName
            arrCounter = arrCounter + 1
         End If
      found = found + 1
      End If
   Loop
   PopulateComboCustomActions arrSubNames, arrCounter
   If arrSubNames(0) <> "" Then
      MsgBox "Successfully added " & arrCounter & _
         " functions from the script.", _
         vbOKOnly + vbInformation, "Parse Script"
   Else
      MsgBox "No Route functions were found when parsing " & _
         "the script.", vbOKOnly + vbInformation, "Parse Script"
   End If
End Sub

Sub PopulateComboCustomActions(arrActions, arrCounter)
   For i = 0 To arrCounter - 1
      frmRouting.comboCommands.AddItem arrActions(i)
   Next
End Sub

Function OpenScriptFile(tmpFileLocation) As Variant
On Error GoTo OpenScriptFile_Err
   Open tmpFileLocation For Input As #1
      bstrEventScript = Input$(LOF(1), #1)
   Close #1
   OpenScriptFile = bstrEventScript
   Exit Function

OpenScriptFile_Err:
   MsgBox "There was an error opening the script file. " & _
      "Please select a different file." & vbLf & _
      Err.Description, vbOKOnly + vbExclamation, _
      "Open Script"
   bValidScript = 0
   Close #1
   OpenScriptFile = ""
End Function
```

An example of this functionality is shown in Figure 14-14.

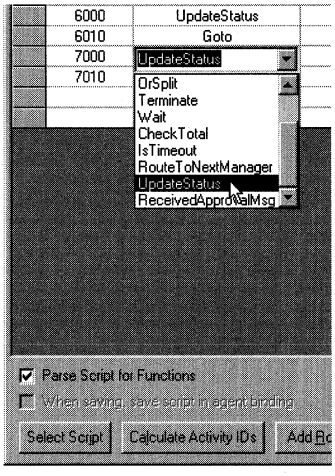


Figure 14-14. *The combo box populated with the intrinsic as well as custom script actions for a routing map.*

If you uncheck the Parse Script For Functions check box, the code will automatically remove the custom actions from the combo box and leave only the intrinsic actions.

The application also makes it easier for you to create new maps after deleting all the rows in the old map because it automatically calculates the ActivityIDs for your rows starting at *100* and incrementing by *10*. You should be careful when using this functionality, because the code cannot detect whether a Goto has jumped to the correct place after you run the calculation.

The application can automatically detect when you type a custom action or intrinsic action in the Action column, and it will set the *Flags* property correctly for you (to *0* for an intrinsic action and *2* for a custom action). You must parse the script for functions to make the custom action detection code work. This action detection functionality, illustrated in the following code, is implemented using the string functions provided in Visual Basic. The code shows you how to perform an exact match of names by using the *InStr* function and flipping the arguments.

```
Private Sub ApplyIntrinsic(tmpNum)
   'This subroutine applies the flags and any custom settings
   'for intrinsic objects
   flexMap = arrIntrinsic(tmpNum)
   SetFlag RT_Flag_Intrinsic
End Sub

Private Sub ApplyScriptAction(tmpNum)
   'This subroutine applies the flags and any custom settings
```

```
   'for intrinsic objects
   flexMap = comboCommands.List(tmpNum)
   SetFlag RT_Flag_VBFunction
End Sub

Private Sub SetFlag(intFlag)
   tmpNextCol = flexMap.Col + 1   'Should be flags
tmpCurRow = flexMap.Row
   flexMap.TextMatrix(tmpCurRow, tmpNextCol) = intFlag
End Sub

Private Sub CheckforIntrinsic()
   'Checks to see if user typed Intrinsic Action
   tmpString = comboCommands.Text
   If tmpString <> "" Then
      tmpString = UCase(tmpString)
      For i = 0 To (intNumIntrinsicActions - 1)
         result = InStr(1, UCase(arrIntrinsic(i)), tmpString)
         If result = 1 Then
            resultotherway = InStr(1, tmpString, _
               UCase(comboCommands.List(i)))
            If resultotherway = 1 Then
               ApplyIntrinsic (i)
               'Set the flags automatically for the user
               Exit Sub
            End If
         End If
      Next
      'No intrinsic
      SetFlag RT_Flag_VBFunction
   End If
End Sub

Private Sub CheckforScriptAction()
   'Checks to see if user typed Intrinsic Action
   tmpString = comboCommands.Text
   If tmpString <> "" Then
      tmpString = UCase(tmpString)
      If intNumIntrinsicActions = comboCommands.ListCount Then
         'No script actions
         Exit Sub
      Else
         For i = intNumIntrinsicActions To _
         (comboCommands.ListCount - 1)
            'Check the strings both ways to get an exact match
            result = InStr(1, UCase(comboCommands.List(i)), _
               tmpString)
```

(continued)

```
            If result = 1 Then
               resultotherway = InStr(1, tmpString, _
                  UCase(comboCommands.List(i)))
               If resultotherway = 1 Then
                  ApplyScriptAction (i)
                  Exit Sub

               End If
            End If
         Next
      End If
   End If
End Sub
```

Saving the Routing Map

The application automatically detects when the user makes changes to a cell in the map and enables the Save button. To save your map in the folder, click the Save button. If you have the check box named When Saving, Save Script In Agent Binding enabled, the application will also save the selected script as the new script for the agent. Here is the code that handles this functionality:

```
Private Sub PrintoutMapError(strError)
   MsgBox strError, vbOKOnly, "Map Error"
End Sub

Private Function CheckforValidMap() As Boolean
   'This subroutine checks to make sure the map is somewhat
   'valid before saving.
   'It checks to make sure columns with activities have IDs.
   'It also checks to make sure the first 3 columns - ID,
   'Flags, and Action - are still available by checking the
   'column headers.
   'You should never be able to delete these columns anyway.
   If flexMap.TextMatrix(0, 1) <> "ActivityID" Then
      PrintoutMapError "You do not have an ActivityID column " & _
         "in the right place!"
      CheckforValidMap = False
      Exit Function
   End If
   If flexMap.TextMatrix(0, 2) <> "Action" Then
      PrintoutMapError "You do not have an Action column in " & _
         "the right place!"
      CheckforValidMap = False
      Exit Function
   End If
   If flexMap.TextMatrix(0, 3) <> "Flags" Then
      PrintoutMapError "You do not have a Flags column in the " & _
         "right place!"
```

```
        CheckforValidMap = False
        Exit Function
    End If

    tmpNumRows = flexMap.Rows - flexMap.FixedRows
    tmpNumCols = flexMap.Cols - flexMap.FixedCols
    For ltmpRow = 1 To tmpNumRows
        tmpDataExists = 0
        For ltmpCol = flexMap.FixedCols To tmpNumCols
            If flexMap.TextMatrix(ltmpRow, ltmpCol) <> "" Then
                'tmpDataExists contains the first column with data
                tmpDataExists = ltmpCol
                Exit For
            End If
        Next
        If tmpDataExists Then
            'Check for ActivityID
            If flexMap.TextMatrix(ltmpRow, 1) = "" Then
                PrintoutMapError "No Activity IDs for Row #:" & _
                    ltmpRow & vbLf & "The first set of data in " & _
                    "this row is " & flexMap.TextMatrix( _
                    ltmpRow, tmpDataExists) _
                    & vbLf & "The map will not be saved."
                    CheckforValidMap = False
                Exit Function
            End If
            If flexMap.TextMatrix(ltmpRow, 2) = "" Then
                PrintoutMapError "No Activity set for Row #:" & _
                    ltmpRow & vbLf & "The first set of data in " & _
                    "this row is " & flexMap.TextMatrix( _
                    ltmpRow, tmpDataExists) _
                    & vbLf & "The map will not be saved."
                    CheckforValidMap = False
                Exit Function
            End If
            If flexMap.TextMatrix(ltmpRow, 3) = "" Then
                PrintoutMapError "No Flag is set for Row #:" & _
                    ltmpRow & vbLf & "The first set of data " & _
                    "in this row is " & flexMap.TextMatrix( _
                    ltmpRow, tmpDataExists) _
                    & vbLf & "The map will not be saved."
                    CheckforValidMap = False
                Exit Function
            End If
        End If
    Next
    CheckforValidMap = True
End Function
```

(continued)

```
Public Sub SaveCurrentMap(otmpMessage)
   Dim oRTRow As Variant
   Dim oRTTmpMap As Object
   Dim lParamCount As Long
   Set RTTmpMap = Nothing
   Dim tmpArr As Variant
   'Array to hold current row parameters
   Dim arrParameters As Variant
   tmpRowCount = flexMap.Rows - flexMap.FixedRows
   tmpParamColumns = MAXCOLUMNS - COLCONSTANT
   Set oRTTmpMap = CreateObject("exrt.map")
   'Clear the map
   oRTTmpMap.DeleteMap
   If otmpMessage = -1 Then
      oRTTmpMap.Message = oRTMessageMap
   Else
      oRTTmpMap.Message = otmpMessage
   End If
   oRTTmpMap.SaveMap
   oRTTmpMap.OpenMap TBL_OPEN_READWRITE
   'Scroll through all the rows in the table and
   'write them out
   For tmpRow = 1 To tmpRowCount
      'Create the Row object
      Set oRTRow = Nothing
      Set oRTRow = CreateObject("exrt.row")
      'Get columns with data in current row, skip blanks
      intFoundSomething = 0
      ReDim arrParameters(tmpParamColumns)
      ReDim tmpArr(0)
      For tmpCol = flexMap.FixedCols To (flexMap.Cols - 1)
         If flexMap.TextMatrix(tmpRow, tmpCol) <> "" Then
            intFoundSomething = 1
            Exit For
         End If
      Next
      If intFoundSomething Then
         lParamCount = 0
         tmpArrayCount = 0
         For tmpCol = (COLCONSTANT - 1) To (MAXCOLUMNS - 1)
            tmpdata = flexMap.TextMatrix(tmpRow, tmpCol)
            If tmpdata <> "" Then
               'Data in the column
               arrParameters(tmpArrayCount) = tmpdata
               'Increase the parameter/argument count
               lParamCount = lParamCount + 1
               tmpArrayCount = tmpArrayCount + 1
            End If
```

```
            Next 'Column
            If lParamCount <> 0 Then
               ReDim tmpArr(lParamCount - 1)
               For i = 0 To (lParamCount - 1)
                  If IsNumeric(arrParameters(i)) Then
                     tmpArr(i) = CLng(arrParameters(i))
                  Else
                     tmpArr(i) = CStr(arrParameters(i))
                  End If
               Next
            End If
            oRTRow.SetArgs lParamCount, tmpArr
            oRTRow.ActivityID = CLng(flexMap.TextMatrix(tmpRow, 1))
            oRTRow.Action = CStr(flexMap.TextMatrix(tmpRow, 2))
            oRTRow.Flags = CLng(flexMap.TextMatrix(tmpRow, 3))
            oRTTmpMap.InsertActivity -1, oRTRow
         Else
            'Didn't find anything
            Set oRTRow = Nothing
         End If
      Next 'Row

      'Need to change something on the message
      If otmpMessage = -1 Then
         oRTMessageMap.Fields.Add "RouteType", vbString, "Custom"
         oRTTmpMap.SaveMap
         oRTMessageMap.Update
      Else
         On Error Resume Next
         Dim strRouteType As String
         strRouteType = oRTMessageMap.Fields.item("RouteType")
         If strRouteType = "" Then
            strRouteType = "Custom"
         End If
         otmpMessage.Fields.Add "RouteType", vbString, strRouteType
         oRTTmpMap.SaveMap
         otmpMessage.Update
      End If
   End Sub

   Private Sub RemoveDirtyFlag()
      'Reset dirty bit, and disable save until next change
      IsDirty = 0
      cmdSaveChanges.Enabled = False
   End Sub

   Private Sub cmdSaveChanges_Click()
      If CheckforValidMap() Then
```

(continued)

```
      If intMapViewer = 1 Then
         SaveCurrentMap oRTMessageMap
      Else
         SaveCurrentMap -1   'Don't need to pass a message
      End If
      If checkSaveScript.Value = vbChecked Then
         SaveRoutingAgentScript
         lblScriptLocation.Caption = "Default script in Folder"
      End If
      RemoveDirtyFlag
   End If
End Sub

Private Sub SaveRoutingAgentScript()
   Set otmpMessage = oSession.GetMessage( _
      oRouteBinding.EntryID, Null)
   bstrEventScript = OpenScriptFile(strFileLocation)
   otmpMessage.Fields.item(PR_EVENT_SCRIPT) = bstrEventScript
   otmpMessage.Update
   oRouteBinding.SaveCustomChanges otmpMessage
   oRouteBinding.SaveChanges
   oBoundFolder.SaveChanges
End Sub
```

The first subroutine called is *cmdSaveChanges_Click*. It calls the *CheckforValidMap* function, which tries to ensure the user is not entering invalid information into the map. (Note that the *CheckforValidMap* function does not check every possible error the user could make when creating maps.) After checking for errors, the code checks the *intMapViewer* value. This value is *0* if the map being viewed is the default folder map, and it is *1* if the map is for a particular process instance.

The next subroutine called is *SaveCurrentMap*, which has a parameter of *–1*. A *–1* value indicates that *SaveCurrentMap* should use oRTMessageMap as the CDO Message object as the location in which to save the current routing map. This subroutine then performs the opposite function of the *Form_Load* subroutine; rather than reading in the information from a map, it writes it out.

You should be aware of a few key issues for the *SaveCurrentMap* code. First, notice that the *SaveCurrentMap* subroutine creates an array to keep track of the number of filled-in parameters for each row. If there are seven total columns for parameters in the map and a particular row has only two parameters, the array created is only two items long. If you created an array that is seven items long with five empty items, you would probably get an error when trying to insert your row into a map.

Second, the code converts all the variables in the array to either Long or String. It is a good idea to do this so you do not get errors when executing your maps after saving them.

The third issue to keep in mind involves the last part of the *SaveCurrentMap* code. The application either adds the field RouteType if it does not already exist on the message or it updates the field if it does exist to indicate the type of route this routing map implements.

If the check box When Saving, Save Script In Agent Binding is enabled and checked, the final subroutine *SaveRoutingAgentScript* is called in cmdSaveChanges_Click. This subroutine gets the agent message, opens the script file and reads it into a variable, replaces the script on the agent message, and saves the changes.

Process Instance Enhancements

One of the most powerful aspects of the updated Agent Install program is that it gives you the ability to see which messages are process instances in a folder, which users have responded to messages, what the users' responses are, and where in the routing map the process instance is currently executing.

The user interface for the process instance enhancements are shown in Figure 14-15. You can see all the messages in a particular folder and whether or not each message is a process instance. From this interface, you can also view the routing map on the selected process instance as well as view the recipient table for the process instance.

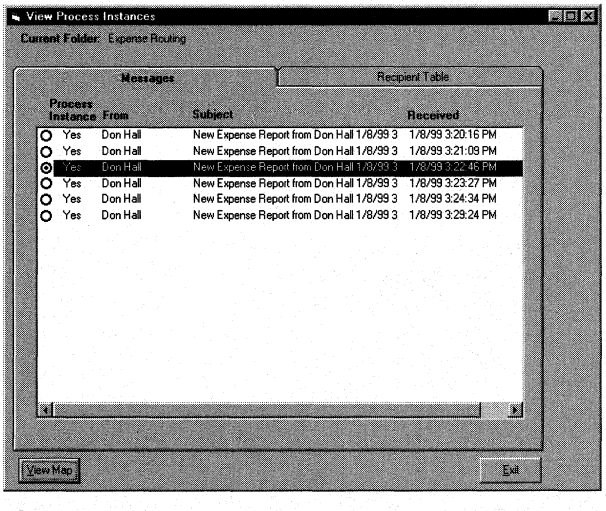


Figure 14-15. *The View Process Instances form, which allows you to quickly see all the process instances in a folder.*

Detecting Process Instances

To implement the process instance enhancements, the application must be able to detect in a particular folder which messages are process instances and which are not. To do this, the application, or more specifically, the frmViewProcInstances form,

creates a ProcInstance object and sets the current message it is looking at in the folder as the object for the *Message* property on the ProcInstance object. Then the application tries to open the map on the message by using the *Map* property on the ProcInstance object. If the application fails to open the map, the item is marked as not being a process instance, as the following code illustrates:

```
Private Sub Form_Load()
'Update the status
   Set tmpoRTMessageMap = oRTMessageMap
   frmProgress.pbMessages.Min = 0
   frmProgress.pbMessages.Max = intMsgCount
   frmProgress.Visible = True
   frmProgress.Refresh
   lblCurrentFolder.Caption = oFolder.Name
   Dim arrTest()
   ReDim arrTest(intMsgCount - 1, 4)

   On Error Resume Next
   For i = 1 To intMsgCount
      frmProgress.pbMessages.Value = i
      arrTest(i - 1, 1) = oMessages.item(i).Sender.Name
      arrTest(i - 1, 2) = oMessages.item(i).subject
      arrTest(i - 1, 3) = oMessages.item(i).TimeReceived
      arrTest(i - 1, 4) = oMessages.item(i).ID
      'Check for maps on the message
      Set oProcInstance = Nothing
      Set oProcInstance = CreateObject("exrt.ProcInstance")
      oProcInstance.Message = oMessages.item(i)
      oProcInstance.Open
      'An error will be raised. Just ignore it.
      Err.Clear
      Set tmpoMap = oProcInstance.Map
      tmpoMap.OpenMap TBL_OPEN_READONLY
      If Err.Number = 0 Then
         'Found another one
         arrTest(i - 1, 0) = "Yes"
      Else
         arrTest(i - 1, 0) = "No"
      End If
   Next
   lbMessages.List() = arrTest
   Unload frmProgress
   frmViewProcInstances.Refresh
   lbMessages.ListIndex = 0
End Sub
```

Viewing the Recipient Table for a Process Instance

After figuring out which items are process instances and which are not, the application allows the user to view the responses on the Recipient Table tab of the View Process Instances dialog box. Figure 14-16 shows the Recipient Table tab for a sample created with the Routing Wizard in a folder named Routed Documents. This tab shows the responses received from the people in the route.

> **NOTE** The users and their responses in your application are shown by the Agent Install program only if your application uses the RecipientEntry object to track user responses. Because the Expense Routing application does not use the RecipientEntry object to track responses, the Recipient Table tab will display "Unknown" for the status when viewing the Expense Routing folder. However, because applications generated by the Routing Wizard use the RecipientEntry object, you can view users and their responses on the Recipient Table tab for applications generated by the Routing Wizard.

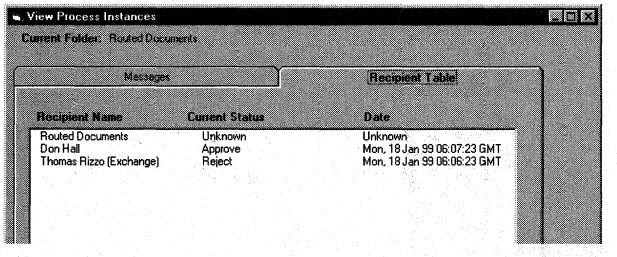


Figure 14-16. *The Recipient Table tab on the View Process Instances dialog box shows responses received for a sample created with Routing Wizard. Since the Expense Routing application does not use the RecipientEntry object, the Recipient Table tab will not display recipient information.*

To retrieve the response information, the application uses the VoteTable and RecipientEntry objects. The application scrolls through the number of recipient entries contained in the message and sets a variable to a RecipientEntry object by using the *Item* method of the VoteTable object. Once the RecipientEntry object is set, the application pulls off the properties for that recipient, such as the recipient name, the status, and time that status was updated.

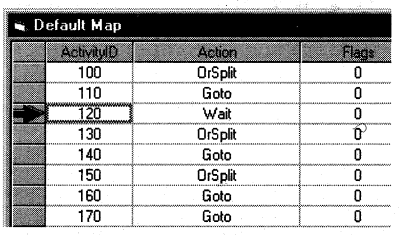
```
Private Sub ssTabProcInstance_Click(PreviousTab As Integer)
   If SSTabProcInstance.Tab = 1 Then
      'Clicked on Recipient Table
      'Check to see if user selected a message
      If lbMessages.ListIndex = -1 Then
         'No selected message
         MsgBox "Please select a message before clicking on " & _
            "the Recipient & "Table tab.", vbOKOnly + _
```

(continued)

```
            vbExclamation, "Recipient Table"
         SSTabProcInstance.Tab = 0
         Exit Sub
      Else
         Dim arrRecips()
         Set oRTVote = CreateObject("exrt.VoteTable")
         'Get the message
         tmpID = lbMessages.Column(4, lbMessages.ListIndex)
         Set otmpMessage = oSession.GetMessage(tmpID, Null)
         oRTVote.PIMessage = otmpMessage

         If oRTVote.Count = 0 Then
            MsgBox "There is no Recipient Table for this " & _
               "message.", vbOKOnly + vbInformation, _
               "Recipient Table"
         Else
            ReDim arrRecips(oRTVote.Count, 2)
            For i = 1 To oRTVote.Count
               Set oRTRecipient = oRTVote.item(i)
               arrRecips(i - 1, 0) = oRTRecipient.Recipient
               If oRTRecipient.Status = "" Then
                  arrRecips(i - 1, 1) = "Unknown"
               Else
                  arrRecips(i - 1, 1) = oRTRecipient.Status
               End If
               If oRTRecipient.Date = "" Then
                  arrRecips(i - 1, 2) = "Unknown"
               Else
                  arrRecips(i - 1, 2) = oRTRecipient.Date
               End If
            Next
            lbRecips.List() = arrRecips
         End If
      End If
   End If
End Sub
```

Viewing the Currently Executing Row in Process Instance

When working with Exchange Server Routing, one of the problems you might run into is not knowing which row a particular process instance is executing. To help you debug your application, you might want to know which row the engine is currently at and see the surrounding rows. The Agent Install application allows you to view the map for a process instance, and it places an arrow next to the currently executing row in the map of the process instance so that you know exactly which row the engine is executing. This arrow is shown in Figure 14-17. When you view the map of a process instance, you will find that almost every time, the current row is either a Terminate or Wait action.



Default Map

ActivityID	Action	Flags
100	OrSplit	0
110	Goto	0
120	Wait	0
130	OrSplit	0
140	Goto	0
150	OrSplit	0
160	Goto	0
170	Goto	0

Figure 14-17. *Viewing the routing map for a particular process instance. The arrow points to the current state of the process instance.*

To indicate which row is executing, the application changes the *intMapViewer* variable to *1* so that the frmRouting form knows that the form calling it is a process instance map rather than the default map in a folder. The frmRouting form points to the currently executing row and disables some functionality that should not be used on process instance maps. The following code, first taken from frmViewProcInstances and then from frmRouting, creates a process instance object and sets the *Message* property to be the currently selected message in the list box on the frmViewProcInstances form. Then the code figures out the currently executing row by using the *CurrentRow* property on the ProcInstance object, and it loads an arrow graphic to point to that row.

```
Private Sub cmdViewMap_Click()
   'This is from frmViewProcInstances
   intMapViewer = 1
   tmpID = lbMessages.Column(4, lbMessages.ListIndex)
   Set oRTMessageMap = oSession.GetMessage(tmpID, Null)
   Set oRTProcInstance = Nothing
   Set oRTProcInstance = CreateObject("exrt.ProcInstance")
   Set otmpMessage = oSession.GetMessage(tmpID, Null)
   oRTProcInstance.Message = otmpMessage
   oRTProcInstance.Open
   Load frmRouting
   frmViewProcInstances.Visible = False
   frmRouting.Visible = True
End Sub

Private Sub Form_Load()
'This is from frmRouting
Dim arrArgs As Variant
   MAXCOLUMNS = 5
   COLCONSTANT = 5
   IsDirty = 0
   cTab = Chr(9)
   'Clear the grid
   flexMap.Clear
   'Add the headers to the grid
```

(continued)

```
FormatColumnHeaders
'Fill Intrinsic Action array
FillIntrinsicActionsArray
'Populate the commands combo
'This should also pull from script
PopulateCombo
'Set bValidScript to indicate no script
bValidScript = 0
If oRouteBinding Is Nothing Then
   lblScriptLocation.Caption = "None"
End If
bstrEventScript = ""
'Fill in folder name
lblCurrentFolder.Caption = oFolder.Name

'Load the map and the activities into the list box
Set oRTMap = CreateObject("exrt.map")
If intMapViewer = 0 Then
   oRTMap.Message = oRTMessageMap
ElseIf intMapViewer = 1 Then
   oRTMap.Message = oRTMessageMap
End If
oRTMap.OpenMap TBL_OPEN_READWRITE
lActivityCount = oRTMap.ActivityCount

For i = 1 To oRTMap.ActivityCount
   Set oRTRow = CreateObject("exrt.row")
   oRTMap.GetRow i - 1, oRTRow
   flexMap.AddItem ""
   flexMap.TextMatrix(i, 1) = CStr(oRTRow.ActivityID)
   flexMap.TextMatrix(i, 2) = oRTRow.Action
   flexMap.TextMatrix(i, 3) = CStr(oRTRow.Flags)
   'Figure out how many parameters there are
   arrArgs = Array()
   oRTRow.GetArgs 1, arrArgs
   If (UBound(arrArgs) + COLCONSTANT) > MAXCOLUMNS Then
      flexMap.Cols = UBound(arrArgs) + COLCONSTANT
      MAXCOLUMNS = UBound(arrArgs) + COLCONSTANT
      FormatColumnHeaders
   End If
   For tmpCounter = 0 To UBound(arrArgs)
      tmpArg = arrArgs(tmpCounter)
      flexMap.TextMatrix(i, tmpCounter + (COLCONSTANT - 1)) = _
         CStr(tmpArg)
   Next
Next
```

```
    If intMapViewer = 1 Then
       'Point to the current row
       tmpCurrentRow = oRTProcInstance.CurrentRow
       flexMap.Col = 0
       'If it can't figure out the current row,
       'skip to the next one
       On Error Resume Next
       flexMap.Row = tmpCurrentRow
       Set flexMap.CellPicture = LoadPicture("Arw05rt.ico")
       flexMap.CellPictureAlignment = flexAlignRightCenter
       flexMap.Col = 1
       flexMap.Row = tmpCurrentRow

       'Disable SaveAs
       cmdSaveAs.Enabled = False

       'Check to see if there is a RouteBinding object.
       'If there is, enable Open Script.
       If Not (oRouteBinding Is Nothing) Then
          cmdOpenScript.Enabled = True
       Else
          cmdOpenScript.Enabled = False
       End If
    End If

End Sub
```

User Interface Enhancements

The final set of enhancements we'll discuss is the user interface enhancements to the Agent Install application. You might be asking yourself, "Who cares about user interface enhancements?" You should! These enhancements show you how to detect agents and routing maps in folders as well as how to copy maps and agents to other folders. The main enhancement is in the frmSaveTo form, which is similar to the folder list main screen of the application except that the images representing folders with only routing maps and folders with agents installed differ, as shown in Figure 14-18. This form can be accessed from the main interface of the updated Agent Install application by clicking the View Default Map button and then the Save As button.

The user interface enhancements are implemented only in the Save Routing Agent To Folder form, because they have performance implications in that every folder must be checked for a routing agent and a routing map—imagine the amount of time it would take to check each folder if you had hundreds of folders. The code on the following page shows how these enhancements are implemented.

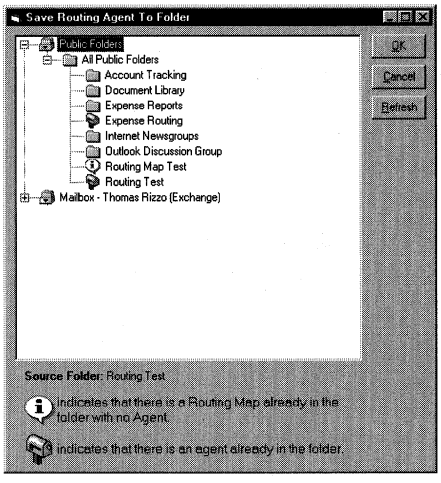


Figure 14-18. *The folder list for the frmSaveTo form. Notice the different icons indicating which folders have agents and which folders have only routing maps.*

```
Private Sub SetFolderImage(tmpAgent)
   Select Case tmpAgent
      Case 0  'Nothing
         tmpImgClosed = FOLDER_CLOSED_IMG
         tmpImgOpened = FOLDER_OPENED_IMG
      Case 1  'Agent
         tmpImgClosed = ROUTINGAGENT_CLOSED_IMG
         tmpImgOpened = ROUTINGAGENT_OPENED_IMG
      Case 2  'Map
         tmpImgClosed = ROUTINGMAPEXISTS_IMG
         tmpImgOpened = ROUTINGMAPEXISTS_IMG
   End Select
End Sub

'*********************************************************************
' Sub:        LoadStores
'
' Description:  Loads the stores into the FolderView.
'
'*********************************************************************
Private Function LoadStores() As Boolean
   On Error GoTo LoadStores_Err:
   Dim bReturnStatus As Boolean

   'Messaging Objects
   Dim objFoldersCol As Object       'Folders Collection
   Dim objFolder As Object           'Folder Object
   Dim objstore As Object            'Store Object
   Dim objStoreRoot As Object        'RootFolder Object
```

```
Dim objInfoStores As Object       'InfoStores Collection
Dim objChildFoldersCol As Object  'Folders Collection
Dim objChildFolder As Object      'Folder Object
Dim objTemp As Object             'Temporary Object
Dim nodTopofStore As Object       'Node Object
Dim lmask As Long
Dim iStoreKey As Integer
Dim strFolderID As String
Dim iLoop As Integer
Dim sLongTermID As String

Me.MousePointer = vbHourglass
'Assume successful
bReturnStatus = True
'Clear Treeview control
tvwFolders.Nodes.Clear
'Get InfoStore object
Set objInfoStores = CDOClass.Session.InfoStores
'Open all stores to speed access later
For Each objstore In objInfoStores
   Set objTemp = objstore.RootFolder
Next
'Iterate through stores
For Each objstore In objInfoStores
   If objstore.ProviderName = "Personal Folders" Then
      'We don't allow them to place agent in a personal folder
   Else
      Set objStoreRoot = objstore.RootFolder
      'Exit if store or root isn't found
      If (objstore Is Nothing) Or (objStoreRoot Is Nothing) Then
         'Problem here
         bReturnStatus = False
         GoTo ObjectCleanup
      End If
      'Set the top node.
      'Loop through main folders.
      Set nodTopofStore = tvwFolders.Nodes.Add(, , _
         objstore.ID, objstore.Name, ROOT_IMG, ROOT_IMG)
      iStoreKey = nodTopofStore.Index
      Set objFoldersCol = objStoreRoot.Folders
      Set objFolder = objFoldersCol.GetFirst
      If Not objFoldersCol Is Nothing Then ' loop through All
         'Add first-level folders to outline
         While Not objFolder Is Nothing
            'Don't display favorites
            If objFolder.Name <> "Favorites" Then
               sLongTermID = objFolder.ID
```

(continued)

```
                    If (Left$(sLongTermID, 2) = "EF") Then
                       'High:  PR_LONGTERM_ENTRYID_FROM_TABLE =
                       '&H6670
                       'Low:   PT_BINARY = &H0102
                       sLongTermID = objFolder.Fields.item( _
                          &H66700102)
                    End If
                    tmpIsAgent = CheckForRoutingAgent(objFolder)
                    tmpIsAgent = CheckforRoutingMap(objFolder, _
                       tmpIsAgent)
                    SetFolderImage tmpIsAgent
                    'Add node to Treeview control
                    Set nodTopofStore = tvwFolders.Nodes. _
                       Add(objstore.ID, tvwChild, sLongTermID, _
                       objFolder.Name, tmpImgClosed, _
                       tmpImgOpened)
                    'Check for subfolders
                    Set objChildFoldersCol = objFolder.Folders
                    'May not have access
                    If Not (objChildFoldersCol Is Nothing) Then
                       'Add the subfolders for this node
                       LoadFolders objstore.ID, _
                          nodTopofStore.Key, _
                          nodTopofStore.Index
                    End If
                 End If
                 Set objChildFoldersCol = Nothing
                 Set objFolder = objFoldersCol.GetNext
              Wend
           End If
        End If
     Next 'store

  ObjectCleanup:
     Set objFoldersCol = Nothing
     Set objFolder = Nothing
     Set objstore = Nothing
     Set objStoreRoot = Nothing
     Set objInfoStores = Nothing
     Set objChildFoldersCol = Nothing
     Set objChildFolder = Nothing
     Set objTemp = Nothing
     Set nodTopofStore = Nothing
     LoadStores = bReturnStatus
     Me.MousePointer = vbNormal
     'Make the Treeview control visible
     tvwFolders.Visible = True
     Exit Function
```

```
LoadStores_Err:
   AppActivate App.Title
   Select Case Err.Number
      Case CdoE_NO_ACCESS, CdoE_NOT_FOUND, CdoE_FAILONEPROVIDER, _
      35602
         Err.Clear
         Resume ObjectCleanup
      Case Else
         Call CDOClass.MapiErrorHandler( _
            "LoadStores in FrmFolders")
         Err.Clear
         'Resume next
         Me.MousePointer = vbNormal
         bReturnStatus = False
         Resume ObjectCleanup
   End Select
End Function

Public Function CheckForRoutingAgent(objFolder) As Integer
'This function checks for any type of agent in the folder.
'It returns 1 if there is an agent or 0 if there is not.
   Dim otmpEvents
   Dim otmpBoundFolder
   Dim otmpBindings
   On Error GoTo CheckForRoutingAgent_Err
   Set otmpEvents = oEvents
   Set otmpBoundFolder = otmpEvents.BoundFolder(objFolder, True)
   Set otmpBindings = otmpBoundFolder.Bindings
   If otmpBindings.Count > 0 Then
      CheckForRoutingAgent = 1
   Else
      CheckForRoutingAgent = 0
   End If
Exit Function

CheckForRoutingAgent_Err:
   'Most likely the error is from hitting the user's Outlook
   'calendar folder
   CheckForRoutingAgent = False
End Function

Public Function CheckforRoutingMap(objFolder, tmpCurrentType) _
As Integer
'This checks for routing maps only in the folder.
'It returns 2 if there is only a map.
   On Error Resume Next
   If tmpCurrentType <> 1 Then
```

(continued)

```
        'Check only folders with no agents
        Set oHidden = objFolder.HiddenMessages
        For Each oHide In oHidden
           Err.Clear
           tmpRouteMap = oHide.Fields("RouteMap")
           If Err.Number = 0 Then
              'There is a map
              CheckforRoutingMap = 2
              Exit Function
           End If
        Next
     End If
     CheckforRoutingMap = tmpCurrentType
  End Function
```

Chapter 15

Programming Exchange Server Using ADSI

Messaging and communication technologies are necessary to build collaborative applications, but they are not the only technologies you need. Having a robust directory is a key requirement for almost any collaborative application, whether it be a simple messaging application or a full-blown workflow system. A directory not only holds communication information such as e-mail addresses and phone numbers, but also holds organizational and hierarchical information, such as managers and direct reports. It stores facility and personal information, such as building location, cost center, and pictures of users.

Being able to retrieve this range of information is important functionality for the application and beneficial to any organization. Recall our Collaboration Data Objects (CDO) Helpdesk application in Chapter 12, which used a directory to obtain a user's personal information. In Chapter 13, the directory in the Event Scripting Expense Report application figured out the identity of the manager whose direct report submitted an expense report. Because Microsoft Exchange Server supports an extensible directory, you can add your own fields to the directory to store the type of custom information we've retrieved in the applications we've looked at so far.

WHAT IS ADSI?

We've seen one way to access the directory—by using the CDO library. CDO provides a set of objects that allows you to query information stored in the Exchange Server directory under the AddressEntry section of the CDO hierarchy. Another way to access the Exchange Server directory is to use Microsoft Active Directory Services Interfaces (ADSI).

ADSI is a set of COM interfaces that allow you to manipulate objects in different directories, including Exchange Server 5.5. ADSI supports different protocols including Lightweight Directory Access Protocol (LDAP), NetWare Directory Services (NDS), and Microsoft Windows NT Directory Services (NTDS). The advantage of ADSI is that it is not designed based on a specific Application Programming Interface (API), which makes it more flexible. Also, ADSI can be used with multiple programming languages, such as Microsoft Visual Basic, Microsoft Visual Basic Scripting Edition (VBScript), Java, and Microsoft Visual C++.

ACCESSING THE DIRECTORY: CDO OR ADSI?

The big question is how to access the directory provided by Exchange Server. Consider using CDO to access the Exchange Server directory when you are running earlier versions of Exchange Server such as Exchange Server 4.0, because Exchange Server 4.0 does not support LDAP, which is required to support the ADSI object library. CDO offers you a limited subset of features because it can access only the properties that have corresponding messaging API (MAPI) unique IDs. Therefore, if CDO provides access to the directory properties that you need in your application, you should use it for directory access. Using CDO has the benefit of allowing you to write your application using a familiar object library, and you do not have to write to a second object library, which can save debugging time.

The other way to access the Exchange Server directory is by using the ADSI object library, which gives you more flexibility. Using ADSI, you can access not only the properties used by CDO but also other properties stored in the Exchange Server directory—for example, configuration information, such as which connectors are installed on the Exchange Server, and whether the Exchange Server is anonymously publishing its directory entries to http clients.

To use ADSI, you have to run a version of Exchange Server that supports LDAP. Exchange Server 5.0 supports LDAP read-only access, and Exchange Server 5.5 supports LDAP read/write access. I recommend that if you want to use ADSI with your Exchange Server, you should install or upgrade to Exchange Server 5.5. With CDO, you can use any version of Exchange Server, from version 4.0 through version 5.5.

DESIGN GOALS OF THE ADSI OBJECT LIBRARY

The ADSI library is the strategic library for accessing directory objects and information from Microsoft, and it doesn't matter whether this information is in Exchange Server or in Active Directory. When designing ADSI, Microsoft had three primary goals in mind. First, ADSI had to be based on COM and on providers to access directory information. In a provider-based model, the client interacts with the COM interfaces exposed by the ADSI library, and the installed providers convert the calls in the interfaces into function calls that access the targeted directory provider. The provider model allows the same calls to be used in an ADSI client application to access any type of directory with an ADSI provider. ADSI ships with four primary directory providers—a Microsoft Windows NT directory provider for Windows NT 4.0, a Novell NDS provider, a Novell NetWare Binding provider for NetWare versions before 4.*x*, and an LDAP provider—but you can write your own because the ADSI provider architecture is extensible. We will focus on the built-in LDAP provider in this chapter because it supports accessing and updating the Exchange Server directory.

WHAT IS ACTIVE DIRECTORY?

Active Directory is the directory technology integrated into Microsoft Windows 2000 Server products. It is the next generation of directory services offered by Microsoft. Because Active Directory is built on Exchange Server technology, you will already be familiar with some of its capabilities and APIs. By placing information in the Exchange Server directory, you can easily migrate to Active Directory by using the replication capabilities of the Exchange Active Directory Connector, which is included with Windows 2000 Server.

Microsoft's second goal for ADSI was to allow developers to use any COM-based development tool with the object library, including Visual Basic, Visual C++, Java, VBScript, and JavaScript. You can decide which tool is the best for creating your ADSI Exchange Server applications. In this chapter, I use VBScript and Microsoft Active Server Pages (ASP) to create a Web-based administrator program for Exchange Server. However, this application easily could have been written in Visual Basic or Java.

The third goal of ADSI was to provide you with a single directory API as a replacement for multiple directory APIs. With Exchange Server, you can use CDO or Directory API (DAPI) to access information stored in the Exchange Server directory. With the Windows NT directory, you can use the Win32 directory functions to access information stored in the Windows NT directory. However, depending on your

application, you can more easily write to a single set of interfaces and use different providers for these directory services rather than learn two or three different APIs. With ADSI, you need to learn only one API to use a multitude of directory services.

ADSI Object Library Architecture

The ADSI object library is a very approachable object library. Although it does not contain many objects, you can perform many functions with them. The only potentially difficult aspect of using the ADSI object library is understanding how to access the objects in your applications using distinguished pathnames. At first, these pathnames can be a little intimidating, but after you experiment with ADSI and its objects, you will understand how to exploit the power of these paths. Creating paths is discussed later in this chapter in the section "Creating Paths to Exchange Server Objects and Attributes."

In the ADSI architecture, every element in a directory service, such as the Exchange Server directory service, is represented by an ADSI object. The interfaces supported by the ADSI object are determined by the underlying functionality of the directory object. For example, a mailbox in Exchange Server, which does not contain directory objects under it, supports the *IADs* interface. In contrast, a *Recipients* container in the Exchange Server directory, which can hold objects such as mailboxes, distribution lists, and other *Recipients* containers under it, implements the *IADsContainer* interface. Since the requirements for a directory object that contains other directory objects are different from those for a single directory object, ADSI provides more methods and properties through extra interfaces for them.

IADs and *IADsContainer* Interfaces

The primary interfaces you will use when working with ADSI objects are the *IADs* interface and the *IADsContainer* interface. The *IADs* interface is required for all ADSI objects. It provides properties that describe the object—in essence, the metadata of the object—and methods that allow you to manage the actual directory information the object contains. ADSI stores this directory information in a property cache, which gives you a mechanism to batch changes or additions to a specific object in a temporary location and then burst this information to the directory service in one call. This property cache is useful because some programming languages such as Visual Basic do not provide native batching mechanisms, and without a property cache, every change you make to an ADSI object is put over the wire, decreasing performance of your application.

The property cache is useful, but only if you remember to use it! The most common mistake new ADSI developers make is not calling the specific ADSI method

SetInfo to flush the cache and submit to the directory service the modifications to the object. If you do not call this method and then exit your application, the changes you make will not be persisted in the directory. The second most common mistake new ADSI developers make is not calling the *GetInfo* method to refresh the cache after making changes. The *SetInfo* method does not automatically refresh the cache for you.

In addition to supporting the *GetInfo* and *SetInfo* methods, the *IADs* interface supports the *Get*, *GetEx*, *GetInfoEx*, *Put*, and *PutEx* methods. As you would expect, *Get* and *Put* do exactly what their names imply: *Get* retrieves a specific property from the directory, and *Put* saves the value for a specific property. The versions of these methods with the *Ex* suffix allow you to get or put a multivalued property. A multivalued property can contain multiple values of the same type. The best example of a multivalued property in the Exchange Server directory is the *Reports* property. Since one person can have many direct reports, the *Reports* property in the directory is a multivalued property—multiple direct report names can be stored in a single property for the directory object named reports. To access this property from ADSI, you must use the *GetEx* method. *GetInfoEx* is provided so that you can specify which properties to refresh in the property cache, preventing you from having to reload the entire cache from the underlying directory service.

The properties that the *IADs* interface implements are *Name*, *AdsPath*, *Class*, *Schema*, and *Parent*. (For our purposes, the *Schema* and *Parent* properties are not as important as the other three, so this discussion will focus on *Name*, *AdsPath*, and *Class*.) The *Name* property returns the relative name of the object. The *AdsPath* property returns the path to the object. In Exchange Server, this would be the LDAP query string that is used to access the object. The *Class* property is important because it returns the schema class name of the object. This property and its return value deserve a bit more attention in the next section.

IADsContainer Interface

As mentioned earlier in this chapter, if a directory object contains other objects, the directory object is considered a container. In ADSI, a container implements not only the *IADs* interface but also the *IADsContainer* interface. The *IADsContainer* interface provides you with more methods than those provided by the *IADs* interface so that you can traverse the child objects in the container as well as modify the container's properties. As you will see in the sample application in this chapter, you can use the For...Each construct in Visual Basic or VBScript to easily loop through all the child objects below a container object and retrieve individual properties from each child object. By traversing the individual objects under the container, you can easily build a hierarchical view of the information stored in the Exchange Server directory.

OTHER ADSI INTERFACES

Covering the other ADSI interfaces is beyond the scope of this book, but you should know that ADSI does provide a powerful feature set so that you can build not only directory applications that work with Exchange Server but also applications that work with other directory services. For example, ADSI defines an *IADsComputer* interface that lets you store information about a computer in a directory service. ADSI also defines interfaces such as *IADsPrintQueue* and *IADsPrintJob*, which enable you to list printers available in the directory service and store specifics about the actual print jobs taking place on those print queues. For more information on the other types of ADSI interfaces that you can use, refer to the "Getting Help with ADSI" section at the end of this chapter.

Exchange Server Object Classes

In Exchange Server, there are some important class names you should be aware of when developing your applications, the most common of which are container, groupOfNames, organizationalPerson, person, and remote-address:

- The container class identifies the object as a container for other objects in the directory.
- The groupOfNames class corresponds to distribution lists in the Exchange Server directory.
- The organizationalPerson class is used to represent recipients in the directory.
- The person class is an abstract class. It is used to represent any object that can receive mail, so other objects such as a distribution list inherit some attributes from the person class. Because the person class is an abstract class, you can never create an explicit object from it.
- The remote-address class corresponds to a custom recipient in the directory.

EXCHANGE SERVER SCHEMA

To write applications using ADSI for the Exchange Server directory, you must first understand the Exchange Server directory schema. This schema defines the available object classes as well as the relationship between these objects in the directory. This schema also contains the attributes for each object class.

To access the Exchange Server directory schema, you need to run the Exchange Administrator program in raw mode. To do this, run the file admin.exe with the /r switch. After the Exchange Administrator program starts, select Raw Directory from the View menu. You should see a container named *Schema* appear in the left pane. When you select the *Schema* container, you will see all the attributes and classes defined by the Exchange Server schema in the right pane, as shown in Figure 15-1. In the Exchange Administrator program, attributes have a folder icon marked with an *A* and classes have a folder icon marked with a *C*. To view and edit information about attributes and classes, select the object and choose Raw Properties from the File menu.

> **NOTE** Be careful when modifying the Exchange Server schema because you could cause unwanted behavior in your Exchange server. For example, if you change a schema property and set it incorrectly, Exchange Server will replicate the incorrect information to all your Exchange servers.

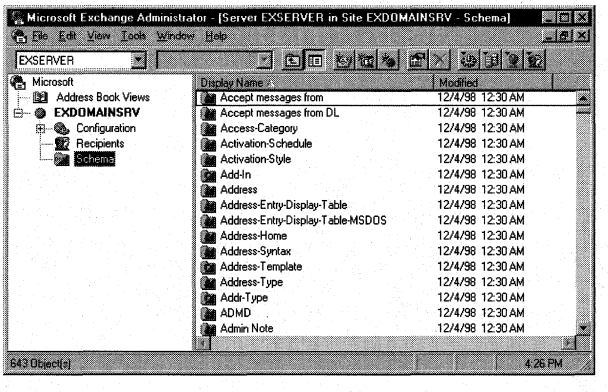


Figure 15-1. *The* Schema *container in the Exchange Administrator program. You can search for attributes and classes defined by the Exchange Server schema using this view.*

Three specific properties on the items contained in the schema are important when writing ADSI applications: the *Access-Category* property, the *Description* property, and the *Heuristics* property.

Access-Category Property

The *Access-Category* property specifies the rights needed by a user to modify an item on the Exchange server. This property contains an integer value:

- The value *0* specifies that only the system can modify an item.
- The value *1* allows users with the Exchange Modify Admin Attributes right to modify an item.

- The value *2* allows users who have the Exchange Modify User Attributes right to modify an item.
- The value *3* specifies that users who have the Exchange Modify Permissions right can change an item.

By default, users assigned as owners of a mailbox automatically have the Modify User Attributes right on their mailbox. When working with ADSI, however, check the *Access-Category* property for any attributes you want to modify before attempting to call your code. Exchange Server sets administrator access on some of the common properties that you might want to allow users to modify in your programs. The best example is the office attribute on a mailbox. By default, the office attribute has an *Access-Category* value of *1*, which specifies that only users with the Modify Admin Attributes permission on the mailbox can change it. Users by default do not have this right, so your ADSI code cannot modify the office attribute if it is running in the security context of the current user. You can modify the *Access-Category* property in the schema so that users can modify certain properties without having the Modify Admin Attributes right.

Description Property

The *Description* property specifies the LDAP name of the item. Sometimes in the Exchange Server directory, the LDAP name for an attribute or class is different from the Exchange Server directory name for that attribute or class. Because ADSI uses the LDAP provider when accessing the Exchange Server directory, you should always use the LDAP name of an attribute or class when working with ADSI.

Heuristics Property

The *Heuristics* property contains configuration information about the item. The value for this property is an integer, which represents a total of 5 bits. For example, a value of *3* stored in this property represents setting bit 0 to the value *1*, and setting bit 1 to the value *1*. Figure 15-2 shows how setting the bits for this property works.

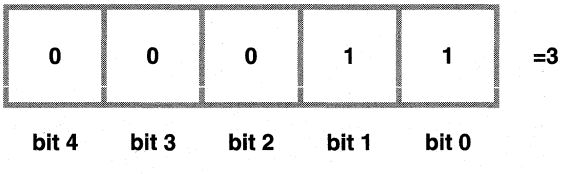


Figure 15-2. *Bits used with the* Heuristics *property.*

The following list describes the effects of changing bit values for the *Heuristics* property:

- Bit 0 in this property specifies whether to replicate the property between Exchange Server sites. Setting the property to *0* makes Exchange Server replicate the property. A setting of *1* prevents the property from being replicated.
- Bit 1 controls the visibility of an item to LDAP clients. Setting this bit to *0* prevents LDAP clients from seeing the item. Setting this bit to *1* allows both anonymous and authenticated LDAP clients to find and query the item.
- Bit 2 provides control over visibility of the item based on authentication of the client. If you set this bit to *0*, only anonymous clients can view the item; authenticated clients will not be able to view it. If you set this bit to *1*, only authenticated clients will be able to see the item.
- Bit 3 sets whether the item is an operational item. Setting this bit to *0* tells Exchange Server to not make the item an operational item. A setting of *1* specifies that the item is an operational item. If this bit is set to *1*, the item will not be visible through ADSI, even though you can set the properties on the item. To access the values in an operational item, you must use the OLE DB provider that is provided by ADSI and a Microsoft ActiveX Data Objects (ADO) query, which is described later in the chapter for the sample ADSI application.
- Bit 4 sets whether the item should be visible in the Exchange Administrator program user interface. A setting of *0* makes the item invisible, and a setting of *1* makes the item visible.

LOOKING FOR OTHER ADSI PROPERTIES

If you prefer not to browse the attributes and classes of the Exchange Server schema using the raw mode option in the Exchange Administrator program, the MSDN Library includes a section that describes the layout of the Exchange Server schema. This section can be found under the Platform SDK\Database and Messaging Services\Microsoft Exchange Server\Microsoft Exchange Server Programmer's Reference\Reference\Directory Schema portion of the MSDN Library.

CREATING PATHS TO EXCHANGE SERVER OBJECTS AND ATTRIBUTES

The first step in developing any ADSI application that accesses the Exchange Server directory is creating a valid instance of ADSI and passing this instance a valid path to the object you want to access. When writing Java, Visual Basic, VBScript, or VBA applications, the easiest way to create a valid instance of the ADSI library is to use the GetObject(AdsPath) syntax. The *AdsPath* parameter contains a valid ADSI path to a specific object. For example, to access a specific Exchange Server object named *ExServer*, a specific Organization in Exchange Server named *Microsoft*, and a specific site in the Organization named *ExSite*, you would pass in the following *AdsPath* to the *GetObject* method:

```
LDAP://ExServer/ou=ExSite, o=Microsoft
```

The *LDAP* at the beginning of the path specifies the ADSI provider to use. If you just want to set an object variable to a specific ADSI provider without attempting to open an object, you can also use the GetObject(ADSIProvider) syntax. For example,

```
oIADs = GetObject("LDAP:")
```

sets the *oIADs* variable to the LDAP provider. To specify a different provider, you would replace the LDAP string with *WinNT* or *NDS*. To access a specific recipient in a specific container on the ExServer Exchange Server, you would use the following *AdsPath*:

```
LDAP://ExServer/cn=RecipientName, cn=RecipientContainer,
ou=ExSite, o=Microsoft
```

As you can see from both examples, the syntax for creating a valid path follows this structure:

```
LDAP://Exchange NT Server Name/cn=Bottommost object,
cn=next level of object, ou=Exchange Site, o=Exchange Organization
```

Throughout the sample application in this chapter, you will see examples of how to use this syntax to query different parts of the Exchange Server directory.

ADSI APPLICATION

The best way to learn about using ADSI to program with the Exchange Server directory is to examine a sample application. I developed one that demonstrates how to create mailboxes, custom recipients, distribution lists, and recipient containers, and how to query for recipient directory information such as a user's name, address, and phone number. There is one caveat with this application—since it uses Dynamic HTML (DHTML) to mimic the Exchange Administrator program, the portion of the application

that queries the attributes of a specific user will not work with Netscape browsers and will work only in Microsoft Internet Explorer 4.0 or later.

Setting Up the ADSI Application

Before you can install the application, you must have a Windows NT 4.0 Server and a client with certain software installed. Table 15-1 describes the required software for setting up the application.

Minimum Software Requirements	*Installation Notes*
Exchange Server 5.5 Service Pack 1 with Outlook Web Access	Exchange Server SP3 is recommended.
Internet Information Server (IIS) 3.0 or later with Active Server Pages	Internet Information Services (IIS) 4.0 is recommended.
CDO library (cdo.dll) CDO Rendering library (cdohtml.dll)	Exchange Server 5.5 Service Pack 1 installs CDO library 1.21 and CDO Rendering library 1.21. Outlook installs CDO library 1.21.
ADSI 2.0	ADSI 2.5 is available as a free download from *http://www.microsoft.com/adsi*. Windows 2000 installs ADSI 2.5 by default.
ActiveX Data Objects	Internet Information Services 4.0 installs ADO 1.5. Visual Basic 6.0 installs ADO 2.0. For more information on ADO, consult *http:/www.microsoft.com/data/*.
For the client: Internet Explorer 4.0, Outlook 98	You can run the client software on the same machine or on a separate machine.

Table 15-1. *Installation requirements for the ADSI application.*

To install the ADSI application, first copy the ADSI folder from the companion CD to the Web server where you want to run the application. Start the IIS administration program. Create a virtual directory that points to the location where you copied the ADSI files, and name the virtual directory *adsi*. Make sure you enable the Execute permissions option for the virtual directory. You will be able to use the following URL to access your ADSI application: *http://*yourservername*/adsi*.

Included with the ADSI files on the CD is a dynamic link library (DLL) named AcctCrt.dll. Use the Regsvr32 utility to register it:

```
regsvr32 acctcrt.dll
```

The first page displayed in the ADSI application is the Logon page, as shown in Figure 15-3. Once a user enters logon information and verifies the dynamically generated Exchange Server information, the application presents a menu of available administrative options for the user, as shown in Figure 15-4.

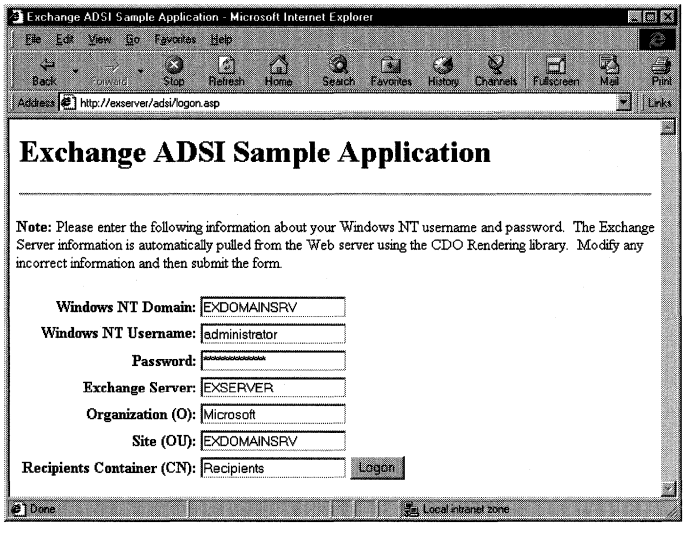


Figure 15-3. *The Logon page for the ADSI application. The Exchange Server name, organization, and site information are pulled dynamically using the CDO Rendering library.*

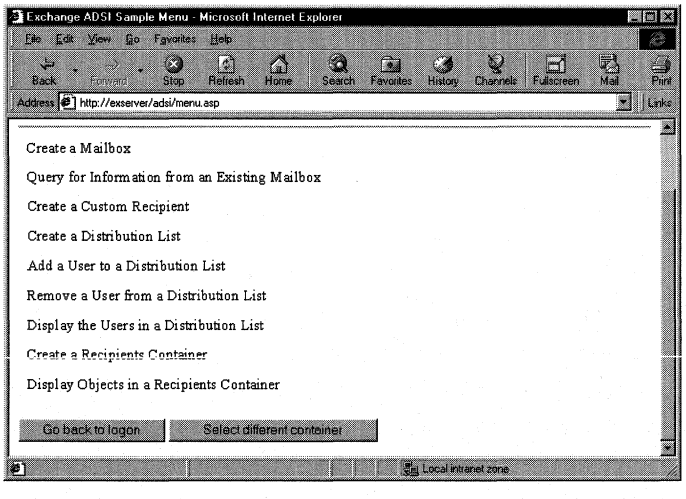


Figure 15-4. *The main menu of the ADSI application. Users can create or modify objects in the Exchange Server directory as long as they have the proper permissions on the object.*

Now let's step through the actual code that makes up these different menu items and see how to use the ADSI object library with an Exchange Server directory.

Logging On to ADSI

The most common operation in the code for the menu items is the object binding code for ADSI. This binding code is dispersed throughout all the code modules in the application, rather than being centralized and performed only once, to make it easier for you to browse the code and understand exactly what is happening.

To bind successfully to an object in the Exchange Server directory using ADSI, you must use the *OpenDSObject* method after using the *GetObject* method to set an object variable to the ADSI library. The *OpenDSObject* method takes four parameters:

- *AdsPath*. The path of the object you want to bind to. We saw how to create this path earlier in the chapter.
- *The Windows NT user name*. This is used to attempt authentication against the directory service.
- *The password for the Windows NT user name you specify*.
- *A flag that specifies the binding option to use*. You can use two possible flags: *&H00000001* specifies to use secure authentication, and *&H00000010* specifies to use encryption.

Depending on the provider used, these flags specifying the binding option might or might not be supported. On the LDAP provider, if you set both flags and pass in a user name and a password, ADSI will perform a simple bind over Secure Sockets Layer (SSL) sessions, which is a secure authentication over a secure channel. The sample application does not use either flag, so a *0* is passed in as the value for the final parameter to indicate that no encryption and no secure authentication should be used.

The following code example shows how to set an object variable to the LDAP provider and log on using the *OpenDSObject* method:

```
Set oIADs = GetObject("LDAP:")
oIADs.OpenDSObject(AdsPath, UserName, Password, 0)
```

Creating a Mailbox

Using ADSI, you can create mailboxes easily in the Exchange Server directory, but they are not fully functional. ADSI does not provide a way for you to specify a Windows NT account as the primary owner of the mailbox, nor does it allow you to change the permissions on the mailbox so that the primary owner has permission to

open it. You must perform these operations by using a separate program or the Exchange Administrator program. To specify a Windows NT account and to change user permissions on the mailbox in the sample application, I used a DLL named AcctCrt.dll. This COM component is discussed in detail in Chapter 16.

Figure 15-5 shows the interface in the ADSI application that prompts the user for information about the mailbox she wants to create. This information is needed by the ADSI code to set specific properties in the Exchange Server directory. The application also asks for Windows NT account information so that a corresponding Windows NT user account can be created and identified as the primary account for the Exchange Server mailbox. Be sure to enter information in all text boxes; otherwise, an error will be displayed.

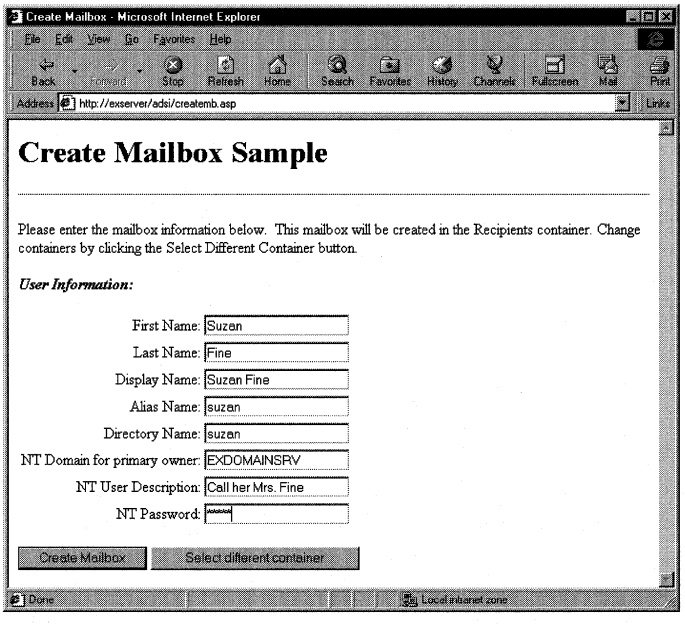


Figure 15-5. *The page where the user can enter information about the mailbox she wants to create using ADSI.*

After the mailbox information is submitted, the ASP page called by the application sets a reference to the ADSI library. The ASP page also retrieves the *Recipients* container, where it will create the mailbox and the private mailbox store so that it can parse out the correct domain name for the user's SMTP address. The application creates proxy addresses for the user just in case the Exchange server has a Lotus cc:Mail or Microsoft Mail connector installed. The application also sets other properties as shown in this code:

```
<%
bstrAT = Request.ServerVariables("AUTH_TYPE")
If InStr(1, "_BasicNTLM", bstrAT, vbTextCompare) < 2 Then
   Response.Buffer = TRUE
   Response.Status = ("401 Unauthorized")
   Response.AddHeader "WWW.Authenticate", "Basic"
   Response.End
end if

dim arrOtherAddresses(1)
'Make sure to set error checking
on error resume next
err.clear
'Retrieve the items from the previous form
fn = Session("FN")
ln = Session("LN")
dn = Session("DN")
al = Session("AL")
dir = Session("DIR")
NTDomain = Session("NTDomain")
NTDescrip = Session("NTDescrip")
NTPassword = Session("NTPassword")

'Create the Windows NT account.
'Use the new AcctCrt component.
set oNTContainer = CreateObject("MSExchange.AcctMgmt")
oNTContainer.NtAccountCreate NTDomain, al, NTPassword, "", ""

'Get the server ID and descriptor
call oNTContainer.GetSidFromName(NTDomain, al, SecurityID)
call oNTContainer.GenerateSecDescriptor(NTDomain, al, _
   SecurityDescriptor)

'Create the Exchange Server mailbox
'Get a reference to the ADSI library
Set oIADS = GetObject("LDAP:")
'Query to the Exchange server
bstr1 = "LDAP://" + Session("Server") + "/cn=" + Session("CN") + _
   ",ou=" + Session("OU") + ",o=" + Session("O")
bstr2 = Session("bstr2")
bstr3 = Session("bstr3")
'Query to the private MDB file on the Exchange server
bstrMDB = "LDAP://" + Session("Server") + _
   "/cn=Microsoft Private MDB,cn=" + Session("Server") + _
   ",cn=Servers ,cn=Configuration,ou=" + Session("OU") + ",o=" + _
   Session("O")
```

(continued)

```
Set oContainer = oIADS.OpenDSObject(bstr1, bstr2, bstr3, 0)
Set oMDB = oIADS.OpenDSObject(bstrMDB, bstr2, bstr3, 0)

'Create a new mailbox (class = organizationalPerson) with the
'correct directory name
Set oIADS = oContainer.Create("organizationalPerson", "cn=" + _
   CStr(dir))
oIADS.Put "NT-Security-Descriptor", (SecurityDescriptor)
oIADS.Put "Assoc-NT-Account", (SecurityID)

'Retrieve the information from the private .mdb
oMDB.GetInfo
'Retrieve the SMTP address for the private .mdb so that it can be parsed
'for the domain name
strSMTPAddr = oMDB.Get("mail")
'Get up to the @ sign
Pos = InStr(strSMTPAddr, "@")
'Parse out the domain name
SMTPExt = Mid(strSMTPAddr, Pos, Len(strSMTPAddr))
strUserSMTPAddr = replace(al, " ", "") + SMTPExt
'Create the domain name for the Message Transfer Agent of the user
strDNMTA = "cn=Microsoft MTA,cn=" + Session("Server") + _
   ",cn=Servers,cn=Configuration,ou=" + Session("OU") + ",o=" + _
   Session("O")

'Set the array of other addresses such as CCMAIL, MSMAIL,
'Profs, and so on
arrOtherAddresses(0) = "MS$" + Session("O") + "/" + _
   Session("OU") + "/" + al
arrOtherAddresses(1) = "CCMAIL$" + al + " at " + Session("OU")
'arrOtherAddresses(2) = "Your other addresses"

'Use the Put command to have ADSI write all of these values to
'the new mailbox.
'mailPreferenceOption must always be 0.
oIADS.Put "mailPreferenceOption", 0
oIADS.Put "givenName", CStr(fn)
oIADS.Put "sn", CStr(ln)
oIADS.Put "cn", CStr(dn)
oIADS.Put "uid", CStr(al)
oIADS.Put "Home-MTA", CStr(strDNMTA)
oIADS.Put "Home-MDB", "cn=Microsoft Private MDB,cn=" + _
   Session("Server") + ",cn=Servers,cn=Configuration,ou=" +
oIADS.Put "mail", CStr(strUserSMTPAddr)
oIADS.Put "MAPI-Recipient", True
oIADS.Put "MDB-Use-Defaults", True
oIADS.PutEx 2, "otherMailbox", (arrOtherAddresses)
oIADS.Put "rfc822Mailbox", CStr(strUserSMTPAddr)
```

```
oIADS.Put "textEncodedORaddress", CStr("c=US;a= ;p=" + _
   Session("O") + ";o=" + Session("OU") + ";s=" + ln + ";g=" + _
   fn + ";")
oIADS.SetInfo

if err.number = 0 then
   'Success!
%>
<SCRIPT LANGUAGE="JavaScript">
   alert("Successfully created mailbox and account!
      Please click OK to continue.");
   window.location="menu.asp";
</SCRIPT>
<%
else
   'Failure!
%>
<SCRIPT LANGUAGE="JavaScript">
   alert("Error! Error Number: <%=err.number %> Description:
      <%=err.Description%>");
   window.location="menu.asp";
</SCRIPT>
<%
end if
%>
```

As you can see in the code, the main properties set on the mailbox include *mailPrefenceOption*, *givenName* (first name), *sn* (last name), *cn* (display name), *uid* (alias name), *Home-MTA*, *Home-MDB*, *mail* (SMTP address), *MAPI-Recipient*, *MDB-Use-Defaults*, *otherMailbox* (other addresses for the mailbox such as CCMAIL and MSMAIL), *rfc822Mailbox* (SMTP address), and *textEncodedORaddress* (X.400 address).

The example shows how to use the ADSI method *PutEx* to enter a multivalue property into the Exchange Server directory. To create the value for a multivalue property, you must use an array in VBScript. When passing this array as an argument to the *PutEx* call, you must place parentheses around the array to de-reference it. Recall that the *Put* and *PutEx* methods will modify the copy of the attributes in the property cache but not in the actual directory service. For this reason, the last statement in the code calls *SetInfo* to take all the changes in the cache and commit them to the directory service.

Querying for Information from an Existing Mailbox

The ADSI application also shows you how to query for information from an existing Exchange Server mailbox. The user interface, shown in Figure 15-6, allows you to type in the first name of the user to find the mailbox.

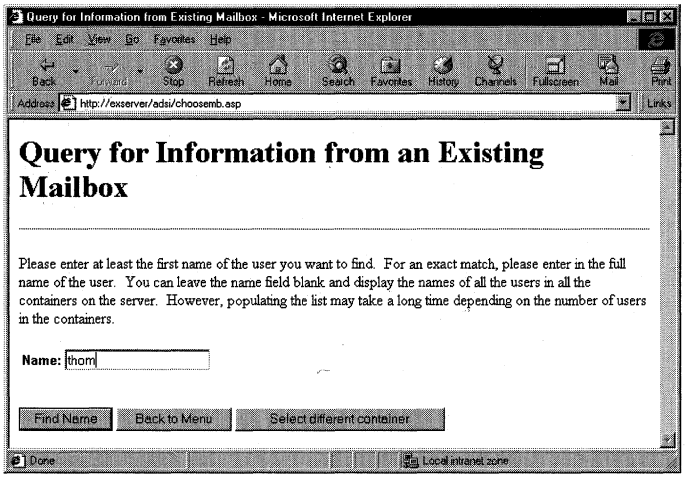


Figure 15-6. *The page allows you to type the first name of the user in the directory to locate the mailbox.*

After the user types in a name, the application uses ADO to query the directory service using LDAP. You might be wondering why ADO is used rather than the *OpenDSObject* method we saw in the code for creating a mailbox. The reason is that to use the *OpenDSObject* method, the user must know the exact name of the desired object in the directory. ADO is more forgiving. When the user is looking for an existing mailbox, the user is not typing in the exact name of the user he is looking for but rather some portion of the first name. Also, since many times users do not know the alias name of users they are querying, forcing users to type in aliases does not make sense. The code to create the ADO object and perform the query is shown here:

```
bstrSearchCriteria = Request.Form("UserName")
bstrServer = Session("Server")
'Create an ADO object
Set ADOconn = CreateObject("ADODB.Connection")
If Err.Number = 0 Then
   ADOconn.Provider = "ADSDSOObject"
   ADOconn.Open "ADs Provider"
   'Create a query using ADO to find all users across all containers
   bstrADOQueryString = "<LDAP://" + bstrServer + _
      ">;(&(objectClass=organizationalPerson)(cn=" + _
      bstrSearchCriteria + "*));cn,adspath;subtree"
   Set objRS = ADOconn.Execute(bstrADOQueryString)
   If Err.Number = 0 Then
      If objRS.RecordCount > 0 Then %>
         <p>Please select one of the following names from the
         list of names.</p>
         <p><em><strong>Returned Names:</strong></em></p>
         <SELECT NAME='MailboxPath'>
```

```
            <%
            'Builds the select control of the queried records
            While Not objRS.EOF
               bstrSelectStatement = bstrSelectStatement & _
                  "<OPTION VALUE='" & objRS.Fields(iCN).Value & _
                  "'>" & objRS.Fields(iADSPATH)
               objRS.MoveNext
            Wend
            Response.Write bstrSelectStatement & "</SELECT>"
         Else %>
            <B><I>No entries match your search criteria.
            Try again using a different value.</I></B>
         <%
         End If
      Else
         If Hex(Err.Number) = 80070044 Then
            Response.Write "<FONT FACE='Arial, Helvetica' " + _
               "SIZE=2>Error " + Hex(Err.Number) + _
               ":  Too many entries match your search " + _
               "criteria!</FONT>"
            Err.Clear
         Else
         %>
            <SCRIPT LANGUAGE="JavaScript">
            alert("Error Number: <%=Hex(Err.Number)%> \n
               Error Description: <%=Err.Description%>")
            history.back()
            </SCRIPT>
            <%
            Err.Clear
            End If
         End If
      Else
%>
   <SCRIPT LANGUAGE="JavaScript">
   alert("Error Number: <%=Hex(Err.Number)%> \nError Description:
      <%=Err.Description%>")
   history.back()
   </SCRIPT>
<%
   Err.Clear
End If
%>
```

This code creates an ADO Connection object and sets the *Provider* property to ADSDSOObject, which specifies the LDAP provider for ADSI. You can specify any string for the connection string argument to the *Open* method of the ADO Connection object. In this case, the application specifies *ADs Provider* as the argument. The

code then creates an LDAP query, which consists of four elements separated by semicolons. This is the format:

```
<LDAP://server/adsidn>;ldapfilter;attributescsv;scope.
```

The *server* argument specifies the name or the IP address of the server where the directory is hosted. The *adsidn* argument specifies the distinguished name in the directory where we want to start our query. You should pass in a correctly formed path, which we saw how to create earlier in the chapter. The filter parameter specifies the LDAP filter string to use. In this case, the LDAP filter states that the object class must be organizationalPerson and the name of the object must match the letters typed in by the user. The next argument, *attributescsv*, is a list of attribute names, separated by commas, that you want returned for each row in the recordset. In our example application, we want the name of the person and the AdsPath to the object that corresponds to that person returned so that we can place this information in the HTML form, as shown in Figure 15-7. The final argument, *scope*, informs the directory service how deeply in the hierarchy to search for the information being queried. The *scope* argument can be one of three values: *base*, *onelevel*, or *subtree*. Since we want to query for all mailboxes that match our specified criteria across all recipient containers in the directory, *subtree* is specified for this argument. The *subtree* argument causes the directory service to search for the information in every *subtree* under the starting object. The *base* argument searches only the currently specified object, and *onelevel* searches one level below the current object in the hierarchy.

If the query successfully returns records that match the filter, the code uses the standard ADO methods to scroll through the recordset and place the records in the HTML form.

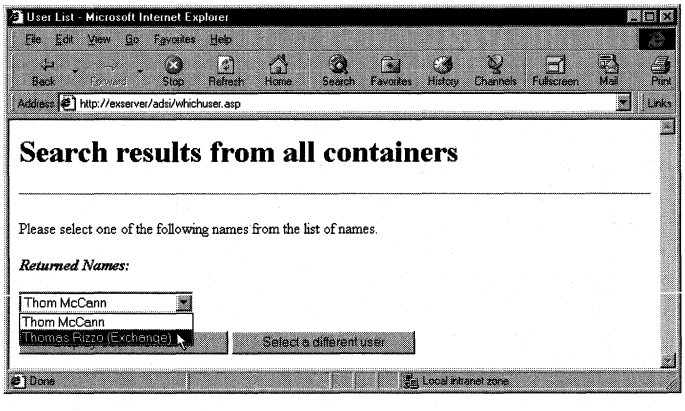


Figure 15-7. *After performing the query, the HTML form is populated with the corresponding recordsets so that a user can pick the person she is interested in finding more information about.*

After the user selects the name of the person she wants to find more information about in the HTML form, the application opens the directory object for this person and retrieves information such as the address, phone number, manager, and direct reports. This information is then represented using a DHTML tabbed dialog box in the browser, as shown in Figure 15-8.

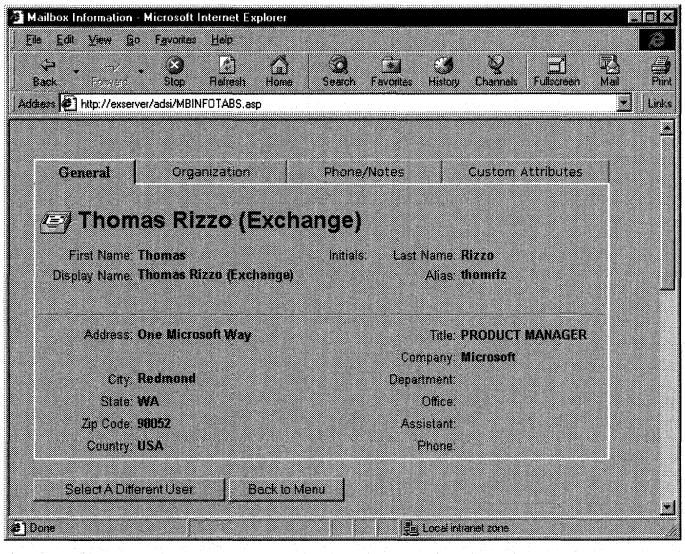


Figure 15-8. *The tabbed dialog box that shows the directory information for a specific user.*

The next section of code retrieves the user information from the directory. I intentionally left out some of the DHTML code from the listing to highlight how ADSI is used in the code. Also, only a portion of the ADSI code is listed here because the structure of the code throughout this part of the application is very similar. Only the specific properties retrieved from the directory using ADSI are different. The full code is included on the companion CD.

```
On Error Resume Next

if Request.QueryString("Path") = "" then
   bstrMailboxPath = Request.Form("MailboxPath")
else
   bstrMailboxPath = Request.Querystring("Path")
end if
bstrServer = Session("Server")
Set objIADs = GetObject(bstrMailboxPath)
strCustomAttributes = "LDAP://" & Session("Server") & _
   "/cn=Attribnum" & ",cn=" & "Microsoft DMD" & ",ou=" & _
   Session("ou") & ",o=" & Session("o")
```

(continued)

```
Function GetAttribName(AttribName)
   strADsPath = Replace(strCustomAttributes,"Attribnum", _
      AttribName,1,1)
   set objAttributeName = GetObject(strADsPath)
   strAttributeName = objAttributeName.Get("Admin-Display-Name")
   GetAttribName = strAttributeName & ":"
end Function

...

<TR>
<TD VALIGN=MIDDLE ALIGN=RIGHT>
<FONT FACE="Arial, Helvetica" SIZE=2>First Name:</TD>
<TD VALIGN=TOP><FONT FACE="Arial, Helvetica" SIZE=2>
<B><%=Server.HTMLEncode(objIADs.Get("givenName"))%></B></FONT></TD>

<TD VALIGN=MIDDLE ALIGN=RIGHT><FONT FACE="Arial, Helvetica" SIZE=2>
Initials:</TD>
<TD VALIGN=TOP><FONT FACE="Arial, Helvetica" SIZE=2>
<B><%=Server.HTMLEncode(objIADs.Get("initials"))%></B></FONT></TD>

<TD VALIGN=MIDDLE ALIGN=RIGHT><FONT FACE="Arial, Helvetica" SIZE=2>
Last Name:</TD>
<TD VALIGN=TOP><FONT FACE="Arial, Helvetica" SIZE=2>
<B><%=Server.HTMLEncode(objIADs.Get("sn"))%></B></FONT></TD>
</TR>
<TR>
<TD VALIGN=MIDDLE ALIGN=RIGHT><FONT FACE="Arial, Helvetica" SIZE=2>
Display Name:</TD>
<TD VALIGN=TOP><FONT FACE="Arial, Helvetica" SIZE=2>
<B><%=Server.HTMLEncode(objIADs.Get("cn"))%></B></FONT></TD>
<TD> </TD>
<TD> </TD>
<TD VALIGN=MIDDLE ALIGN=RIGHT><FONT FACE="Arial, Helvetica" SIZE=2>
Alias:</TD>
<TD VALIGN=TOP><FONT FACE="Arial, Helvetica" SIZE=2>
<B><%=Server.HTMLEncode(objIADs.Get("uid"))%></B></FONT></TD>
</TR>

<TR>
<TD WIDTH="100%" COLSPAN="10"> <HR>
</TD>
</TR>

<TR>
<TD VALIGN=TOP ALIGN=RIGHT><FONT FACE="Arial, Helvetica" SIZE=2>
Address:</FONT></TD>
<TD VALIGN=TOP><FONT FACE="Arial, Helvetica" SIZE=2>
<B><%=Server.HTMLEncode(objIADs.Get("postalAddress"))%></B>
</FONT></TD>
```

```
<TD> </TD>
<TD> </TD>

<TD ALIGN=RIGHT><FONT FACE="Arial, Helvetica" SIZE=2>
Title:</FONT></TD>
<TD ALIGN=LEFT><FONT FACE="Arial, Helvetica" SIZE=2>
<B><%=Server.HTMLEncode(objIADs.Get("title"))%></B></FONT></TD>
</TR>

<TR>
<TD> </TD>
<TD> </TD>
<TD> </TD>
<TD> </TD>

<TD ALIGN=RIGHT><FONT FACE="Arial, Helvetica" SIZE=2>
Company:</FONT></TD>
<TD><FONT FACE="Arial, Helvetica" SIZE=2>
<B><%=Server.HTMLEncode(objIADs.Get("Company"))%></B></FONT></TD>
</TR>

<TR><TD ALIGN=RIGHT><FONT FACE="Arial, Helvetica" SIZE=2>
City:</FONT></TD>
<TD VALIGN=TOP><FONT FACE="Arial, Helvetica" SIZE=2>
<B><%=Server.HTMLEncode(objIADs.Get("l"))%></B></FONT></TD>

<TD> </TD>
<TD> </TD>

<TD ALIGN=RIGHT><FONT FACE="Arial, Helvetica" SIZE=2>
Department:</FONT></TD>
<TD><FONT FACE="Arial, Helvetica" SIZE=2>
<B><%=Server.HTMLEncode(objIADs.Get("department"))%></B></FONT></TD>
</TR>

<TR>
<TD ALIGN=RIGHT><FONT FACE="Arial, Helvetica" SIZE=2>
State:</FONT></TD>
<TD><FONT FACE="Arial, Helvetica" SIZE=2>
<B><%=Server.HTMLEncode(objIADs.Get("st"))%></B></FONT></TD>

<TD> </TD>
<TD> </TD>

<TD ALIGN=RIGHT><FONT FACE="Arial, Helvetica" SIZE=2>
Office:</FONT></TD>
<TD><FONT FACE="Arial, Helvetica" SIZE=2>
<B><%=Server.HTMLEncode(objIADs.Get("physicalDeliveryOfficeName"))%>
</B></FONT></TD>
</TR>
```

(continued)

```
<TR>
<TD ALIGN=RIGHT><FONT FACE="Arial, Helvetica" SIZE=2>
Zip Code:</FONT></TD>
<TD><FONT FACE="Arial, Helvetica" SIZE=2>
<B><%=Server.HTMLEncode(objIADs.Get("postalCode"))%></B>
</FONT></TD>

<TD> </TD>
<TD> </TD>

<TD ALIGN=RIGHT><FONT FACE="Arial, Helvetica" SIZE=2>
Assistant:</FONT></TD>
<TD><FONT FACE="Arial, Helvetica" SIZE=2>
<B><%=Server.HTMLEncode(objIADs.Get("secretary"))%></B></FONT></TD>
</TR>

<TR><TD ALIGN=RIGHT><FONT FACE="Arial, Helvetica" SIZE=2>
Country:</FONT></TD>
<TD><FONT FACE="Arial, Helvetica" SIZE=2>
<B><%=Server.HTMLEncode(objIADs.Get("co"))%></B></FONT></TD>
<TD> </TD>
<TD> </TD>

<TD ALIGN=RIGHT><FONT FACE="Arial, Helvetica" SIZE=2>
Phone:</FONT></TD>
<TD><FONT FACE="Arial, Helvetica" SIZE=2><B>
<%=Server.HTMLEncode(objIADs.Get("telephoneNumber"))%>
</B></FONT></TD>
</TR>
</TABLE>
</DIV>

<DIV CLASS=conts ID=t2Contents>

<!--  Draw out the tab for Organization -->
<TABLE WIDTH=600>
<TR>
<TD WIDTH=100% COLSPAN="5"><IMG SRC="mailbox.jpg" ALIGN="middle">
 <FONT FACE="Arial, Helvetica" SIZE=5>
<B><%=Server.HTMLEncode(objIADs.Get("cn"))%></B></FONT></TD>
</TR>

<TR> </TR></TABLE>
<TABLE BORDER=0>
<TR>
<TD ALIGN=LEFT NoWrap><FONT FACE="Arial, Helvetica" SIZE=2>
Manager Name:</FONT></TD>
```

```
<TR> </TR>
<%
   strManager = objIADs.Get("manager")
   strManagerPath = "LDAP://" & Session("Server") & "/" & strManager
   set oIADsManager = GetObject(strManagerPath)
   strManagercn = Server.HTMLEncode(oIADsManager.Get("cn"))
%>
<TR>
<TD><FONT FACE="Arial, Helvetica" SIZE=2><B>
<A Href='MBINFOTABS.ASP?Path=<%=strManagerPath%>'>
<%=strManagercn%></a></B></FONT></TD>
</TR>
<TR><TD> </TD></TR>
<TR><TD ALIGN=LEFT><FONT FACE="Arial, Helvetica" SIZE=2>
Direct Reports:</FONT></TD></TR>

<% err.clear
   strReports = objIADs.GetEx("Reports")
      for i = LBound(strReports) to UBound(strReports)
         'Get each Directory Service object to return the friendly name
         strDirectPath = "LDAP://" & Session("Server") & _
            "/" & strReports(i)
         set oIADsReports = GetObject(strDirectPath)
         strReportscn = _
            Server.HTMLEncode(oIADsReports.Get("cn"))
%>
         <TR><TD><FONT FACE="Arial, Helvetica" SIZE=2>
         <B><img src="mailboxs.jpg" ALIGN="Middle"> 
         <A Href='MBINFOTABS.ASP?Path=<%=strDirectPath%>'>
         <%=strReportscn%></a>
         </B></TD></TR>
<%
      next
%>
</TR></TABLE>
</DIV>
<DIV CLASS=conts ID=t4Contents>
<!--  Draw out the tab for Custom Attributes -->
<Table width=600>
<TR>
<TD width=100% colspan="5"><img src="mailbox.jpg" align="middle">
 <FONT FACE="Arial, Helvetica" SIZE=5>
<B><%=Server.HTMLEncode(objIADs.Get("cn"))%></B></FONT></TD>
</TR>

<TR> </TR>
</TABLE>
<Table>
```

(continued)

```
<TR><TD ALIGN=RIGHT NoWrap><FONT FACE="Arial, Helvetica" SIZE=2>
<%=GetAttribName("Extension-Attribute-1")%></FONT></TD>
<TD><FONT FACE="Arial, Helvetica" SIZE=2>
<B><%=Server.HTMLEncode(objIADs.Get("Extension-Attribute-1"))%>
</B></FONT></TD>
</TR>

<TR><TD ALIGN=RIGHT NoWrap><FONT FACE="Arial, Helvetica" SIZE=2>
<%=GetAttribName("Extension-Attribute-2")%></FONT></TD>
<TD><FONT FACE="Arial, Helvetica" SIZE=2>
<B><%=Server.HTMLEncode(objIADs.Get("Extension-Attribute-2"))%>
</B></FONT></TD>
</TR>

<TR><TD ALIGN=RIGHT NoWrap><FONT FACE="Arial, Helvetica" SIZE=2>
<%=GetAttribName("Extension-Attribute-3")%></FONT></TD>
<TD><FONT FACE="Arial, Helvetica" SIZE=2>
<B><%=Server.HTMLEncode(objIADs.Get("Extension-Attribute-3"))%></B>
</FONT></TD>
</TR>

<TR><TD ALIGN=RIGHT NoWrap><FONT FACE="Arial, Helvetica" SIZE=2>
<%=GetAttribName("Extension-Attribute-4")%></FONT></TD>
<TD><FONT FACE="Arial, Helvetica" SIZE=2>
<B><%=Server.HTMLEncode(objIADs.Get("Extension-Attribute-4"))%>
</B></FONT></TD>
</TR>

<TR><TD ALIGN=RIGHT NoWrap><FONT FACE="Arial, Helvetica" SIZE=2>
<%=GetAttribName("Extension-Attribute-5")%></FONT></TD>
<TD><FONT FACE="Arial, Helvetica" SIZE=2><B>
<%=Server.HTMLEncode(objIADs.Get("Extension-Attribute-5"))%>
</B></FONT></TD>
</TR>

<TR><TD ALIGN=RIGHT NoWrap><FONT FACE="Arial, Helvetica" SIZE=2>
<%=GetAttribName("Extension-Attribute-6")%></FONT></TD>
<TD><FONT FACE="Arial, Helvetica" SIZE=2>
<B><%=Server.HTMLEncode(objIADs.Get("Extension-Attribute-6"))%>
</B></FONT></TD>
</TR>

<TR><TD ALIGN=RIGHT NoWrap><FONT FACE="Arial, Helvetica" SIZE=2>
<%=GetAttribName("Extension-Attribute-7")%></FONT></TD>
<TD><FONT FACE="Arial, Helvetica" SIZE=2><B>
<%=Server.HTMLEncode(objIADs.Get("Extension-Attribute-7"))%>
</B></FONT></TD>
</TR>

<TR><TD ALIGN=RIGHT NoWrap><FONT FACE="Arial, Helvetica" SIZE=2>
```

```
<%=GetAttribName("Extension-Attribute-8")%></FONT></TD>
<TD><FONT FACE="Arial, Helvetica" SIZE=2><B>
<%=Server.HTMLEncode(objIADs.Get("Extension-Attribute-8"))%>
</B></FONT></TD>
</TR>

<TR><TD ALIGN=RIGHT NoWrap><FONT FACE="Arial, Helvetica" SIZE=2>
<%=GetAttribName("Extension-Attribute-9")%></FONT></TD>
<TD><FONT FACE="Arial, Helvetica" SIZE=2>
<B><%=Server.HTMLEncode(objIADs.Get("Extension-Attribute-9"))%>
</B></FONT></TD>
</TR>
</TABLE>
</FORM>
</DIV>
```

The mailbox that the user wants to query is passed to the ASP page. Using the *GetObject* method, the code opens that mailbox in the Exchange Server directory and sets an object variable, *objIADs*, to that mailbox. Throughout much of the remaining code, the *Get* method of the objIADs object is used to retrieve specific attributes on the mailbox.

The most interesting pieces of code besides the code for retrieving attributes include those that retrieve the user's manager and direct reports from the directory. In the application, the manager's name is displayed as a hyperlink on the Organization tab so that users can quickly look up the manager's directory information. The direct reports of the current user are also displayed as hyperlinks on the Organization tab so that users can look at the direct report's directory information as well. Figure 15-9 shows a sample of these hyperlinks.

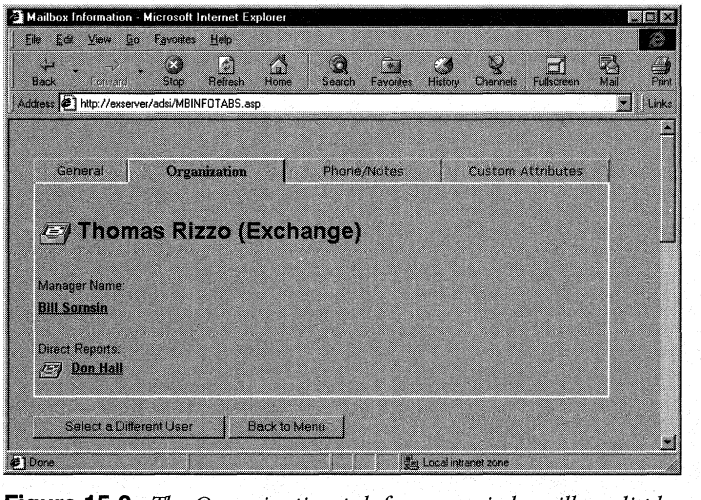


Figure 15-9. *The Organization tab for a queried mailbox displays the manger and direct reports as hyperlinks.*

The following code implements the hyperlink functionality:

```
<Table border=0>
<TR>
<TD ALIGN=LEFT NoWrap><FONT FACE="Arial, Helvetica" SIZE=2>
Manager Name:</FONT></TD>
<TR> </TR>
<%
   strManager = objIADs.Get("manager")
   strManagerPath = "LDAP://" & Session("Server") & "/" & strManager
   set oIADsManager = GetObject(strManagerPath)
   strManagercn = Server.HTMLEncode(oIADsManager.Get("cn"))
%>
<TR>
<TD><FONT FACE="Arial, Helvetica" SIZE=2><B>
<A Href='MBINFOTABS.ASP?Path=<%=strManagerPath%>'>
<%=strManagercn%></a></B></FONT></TD>
</TR>
<TR><TD> </TD></TR>
<TR><TD ALIGN=LEFT><FONT FACE="Arial, Helvetica" SIZE=2>
Direct Reports:</FONT></TD></TR>
<%
err.clear
strReports = objIADs.GetEx("Reports")
for i = LBound(strReports) to UBound(strReports)
   'Get each DS object to return the friendly name
   strDirectPath = "LDAP://" & Session("Server") & "/" & _
      strReports(i)
   set oIADsReports = GetObject(strDirectPath)
   strReportscn = Server.HTMLEncode(oIADsReports.Get("cn"))
%>
   <TR><TD><FONT FACE="Arial, Helvetica" SIZE=2>
   <B><img src="mailboxs.jpg" ALIGN="Middle"> 
   <A Href='MBINFOTABS.ASP?Path=<%=strDirectPath%>'>
   <%=strReportscn%></a>
   </B></FONT></TD></TR>
<%
   next
%>
</TR></TABLE>
```

When you retrieve the manager property from the Exchange Server directory, the directory returns the distinguished name of the manager. To retrieve the display name of the manager, the code uses the distinguished name to create a full AdsPath to the directory object that corresponds to the manager. Then the code opens that object and retrieves the display name of the manager.

To retrieve the direct reports, the code uses the *GetEx* method in ADSI. (Recall that the reports attribute is a multivalued property. You must use *GetEx* when retrieving multivalued properties from the Exchange Server directory.) The *GetEx* code returns an array of distinguished names for all direct reports of the current user. The code scrolls through each direct report in the array, and it displays as a hyperlink an image and the full name of each direct report.

The next code snippet includes some interesting code that retrieves the custom attribute names and the values for these custom attributes. The Exchange Server directory contains 15 customizable attributes that developers or administrators can use to specify custom properties for each entry in the directory. Because you can customize the attribute names so that they correspond to the value types you store in the attribute, such as cost center or social security number, the application queries the Exchange Server directory for the names of the custom attributes. The application also queries the directory for the actual values in the attributes. All of this functionality is implemented in the following section of code:

```
strCustomAttributes = "LDAP://" & Session("Server") & _
   "/cn=Attribnum" & ",cn=" & "Microsoft DMD" & ",ou=" & _
   Session("ou") & ",o=" & Session("o")

Function GetAttribName(AttribName)
   strADsPath = Replace(strCustomAttributes,"Attribnum", _
      AttribName,1,1)
   set objAttributeName = GetObject(strADsPath)
   strAttributeName = objAttributeName.Get("Admin-Display-Name")
   GetAttribName = strAttributeName & ":"
end Function

...

<Table>
<TR><TD ALIGN=RIGHT NoWrap><FONT FACE="Arial, Helvetica" SIZE=2>
<%=GetAttribName("Extension-Attribute-1")%></FONT></TD>
<TD><FONT FACE="Arial, Helvetica" SIZE=2>
<B><%=Server.HTMLEncode(objIADs.Get("Extension-Attribute-1"))%>
</B></FONT></TD>
</TR>

<TR><TD ALIGN=RIGHT NoWrap><FONT FACE="Arial, Helvetica" SIZE=2>
<%=GetAttribName("Extension-Attribute-2")%></FONT></TD>
<TD><FONT FACE="Arial, Helvetica" SIZE=2>
<B><%=Server.HTMLEncode(objIADs.Get("Extension-Attribute-2"))%>
</B></FONT></TD>
</TR>
```

(continued)

```
<TR><TD ALIGN=RIGHT NoWrap><FONT FACE="Arial, Helvetica" SIZE=2>
<%=GetAttribName("Extension-Attribute-3")%></FONT></TD>
<TD><FONT FACE="Arial, Helvetica" SIZE=2>
<B><%=Server.HTMLEncode(objIADs.Get("Extension-Attribute-3"))%>
</B></FONT></TD>
</TR>

<TR><TD ALIGN=RIGHT NoWrap><FONT FACE="Arial, Helvetica" SIZE=2>
<%=GetAttribName("Extension-Attribute-4")%></FONT></TD>
<TD><FONT FACE="Arial, Helvetica" SIZE=2>
<B><%=Server.HTMLEncode(objIADs.Get("Extension-Attribute-4"))%>
</B></FONT></TD>
</TR>

<TR><TD ALIGN=RIGHT NoWrap><FONT FACE="Arial, Helvetica" SIZE=2>
<%=GetAttribName("Extension-Attribute-5")%></FONT></TD>
<TD><FONT FACE="Arial, Helvetica" SIZE=2>
<B><%=Server.HTMLEncode(objIADs.Get("Extension-Attribute-5"))%>
</B></FONT></TD>
</TR>

<TR><TD ALIGN=RIGHT NoWrap><FONT FACE="Arial, Helvetica" SIZE=2>
<%=GetAttribName("Extension-Attribute-6")%></FONT></TD>
<TD><FONT FACE="Arial, Helvetica" SIZE=2>
<B><%=Server.HTMLEncode(objIADs.Get("Extension-Attribute-6"))%>
</B></FONT></TD>
</TR>

<TR><TD ALIGN=RIGHT NoWrap><FONT FACE="Arial, Helvetica" SIZE=2>
<%=GetAttribName("Extension-Attribute-7")%></FONT></TD>
<TD><FONT FACE="Arial, Helvetica" SIZE=2>
<B><%=Server.HTMLEncode(objIADs.Get("Extension-Attribute-7"))%>
</B></FONT></TD>
</TR>

<TR><TD ALIGN=RIGHT NoWrap><FONT FACE="Arial, Helvetica" SIZE=2>
<%=GetAttribName("Extension-Attribute-8")%></FONT></TD>
<TD><FONT FACE="Arial, Helvetica" SIZE=2>
<B><%=Server.HTMLEncode(objIADs.Get("Extension-Attribute-8"))%>
</B></FONT></TD>
</TR>

<TR><TD ALIGN=RIGHT NoWrap><FONT FACE="Arial, Helvetica" SIZE=2>
<%=GetAttribName("Extension-Attribute-9")%></FONT></TD>
<TD><FONT FACE="Arial, Helvetica" SIZE=2>
<B><%=Server.HTMLEncode(objIADs.Get("Extension-Attribute-9"))%>
</B></FONT></TD>
</TR>
</TABLE>
```

Notice that in the *strCustomAttributes* string, the container *Microsoft DMD* is specified. This container corresponds to the actual schema definitions for the Exchange Server directory. To retrieve the names of any custom attributes, we must query the schema. To make the querying easier, I created a function that takes a string that specifies the name of the custom attribute you want to query. The function in turn grabs that attribute in the schema and figures out the corresponding custom name of the attribute—for example, cost center. The code then uses the standard *Get* method on the user's mailbox to retrieve the attribute value for that particular user and attribute, as shown in Figure 15-10.

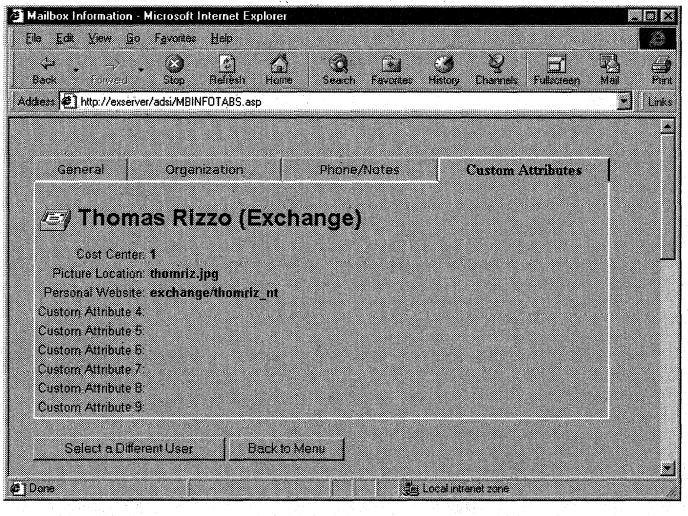


Figure 15-10. *The Custom Attributes tab that shows the custom attributes for the queried mailbox and the custom names of those attributes from the Exchange Server schema.*

Creating a Custom Recipient

The code for creating a custom recipient in the Exchange Server directory is similar to the code for creating a mailbox, as you would expect. The main differences are these:

- You must specify remote-address as the object class for a custom recipient rather than specify organizationalPerson, which is the object class for a mailbox.
- You need to set different properties when creating a custom recipient. For example, you must set the *target-address* property of the custom recipient, which specifies the actual address of the recipient.

The following code creates a custom recipient using ADSI:

```
Dim arrOtherAddresses(1)

on error resume next
err.clear
smtp = Request.Form("SMTP")
fn = Request.Form("FN")
ln = Request.Form("LN")
dn = Request.Form("DN")
al = Request.Form("AL")
dir = Request.Form("DIR")

Set oIADs = GetObject("LDAP:")
bstr1 = "LDAP://" + Session("Server") + "/cn=" + Session("CN") + _
   ",ou=" + Session("OU") + ",o=" + Session("O")
bstr2 = Session("bstr2")
bstr3 = Session("bstr3")

Set oContainer = oIADs.OpenDSObject(bstr1, bstr2, bstr3, 0)
Set oIADs = oContainer.Create("remote-address", "cn=" + CStr(dir))

arrOtherAddresses(0) = "MS$" + Session("O") + "/" + _
   Session("OU") + "/" + al
arrOtherAddresses(1) = "CCMAIL$" + al + " at " + Session("OU")

oIADs.Put "target-address", "SMTP:" + CStr(smtp)
oIADs.Put "givenName", CStr(fn)
oIADs.Put "sn", CStr(ln)
oIADs.Put "cn", CStr(dn)
oIADs.Put "uid", CStr(al)
oIADs.Put "MAPI-Recipient", False
oIADs.Put "mail", CStr(smtp)
oIADs.PutEx 2, "otherMailbox", (arrOtherAddresses)
oIADs.Put "rfc822Mailbox", CStr(smtp)
oIADs.Put "textEncodedORaddress", CStr("c=US;a= ;p=" + _
   Session("O") + ";o=" + Session("OU") + ";s=" + ln + _
   ";g=" + fn + ";")
oIADs.SetInfo
```

Creating a Distribution List

Creating a distribution list, like creating a custom recipient, is again very similar to creating a mailbox. The object class for a distribution list is groupOfNames, and the properties you need to set for the distribution list are a little different from the properties you set for a mailbox. For example, for a distribution list, you can set the *report-to-owner* and the *report-to-originator* properties, which specify whether reports

should be sent to the distribution list owner or to the message originator, respectively. You can also set the distribution list *owner* property, as shown in the next snippet of code, by placing the distinguished name of a user into the *owner* property. This code creates a distribution list:

```
<%
dim arrOtherAddress(1)
on error resume next
err.clear
cn = Request.Form("cn")
uid = Request.Form("uid")

owner = Request.Form("owner")
Set oIADs = GetObject("LDAP:")
bstr1 = "LDAP://" + Session("Server") + "/cn=" + Session("CN") + _
   ",ou=" + Session("OU") + ",o=" + Session("O")
bstr2 = "cn=" + Session("UserName") + ", cn=" + Session("Domain")
bstr3 = Session("Password")
bstrMDB = "LDAP://" + Session("Server") + _
   "/cn=Microsoft Private MDB,cn=" + Session("Server") + _
   ",cn=Servers ,cn=Configuration,ou=" + Session("OU") + _
   ",o=" + Session("O")
bstrOwner = "LDAP://" + Session("Server") + "/cn=" + owner + _
   ",cn=" + Session("CN") + ",ou=" + Session("OU") + _
   ",o=" + Session("O")
Set oContainer = oIADs.OpenDSObject(bstr1, bstr2, bstr3, 0)
Set oObject = oIADs.OpenDSObject(bstrMDB, bstr2, bstr3, 0)
set oOwner = oIADs.OpenDSObject(bstrOwner,bstr2,bstr3,0)
Set oIADs = oContainer.Create("groupOfNames", "cn=" + uid)
oIADs.Put "cn", Cstr(cn)
oIADs.Put "uid", CStr(uid)
oIADs.Put "owner", oOwner.distinguishedName
oObject.GetInfo
Mail = oObject.Get("mail")
Pos = InStr(Mail, "@")
SMTPExt = Mid(Mail, Pos, Len(Mail))
iaddr = replace(uid, " ", "") + SMTPExt
arrOtherAddress(0) = CStr("MS$" + Session("O") + "/" + _
   Session("OU") + "/" + uid)
arrOtherAddress(1) = Cstr("CCMAIL$" + uid + " at " + _
   Session("OU"))
oIADs.Put "distinguishedName", CStr("cn=" + uid + ",cn=" + _
   Session("CN") + ",ou=" + Session("OU") + ",o=" + Session("O"))
oIADs.Put "mail", CStr(iaddr)
oIADs.PutEx 2, "otherMailbox", (arrOtherAddress)
oIADs.Put "Report-To-Originator", True
oIADs.Put "Report-to-Owner", False
```

(continued)

```
oIADs.Put "Replication-Sensitivity", CInt(20)
oIADs.Put "rfc822Mailbox", CStr(iaddr)
oIADs.Put "textEncodedORaddress", CStr("c=US;a= ;p=" + _
   Session("O") + ";o=" + Session("OU") + ";s=" + uid + ";")
oIADs.SetInfo
if err.number = 0 then
   'Success!
%>
   <SCRIPT LANGUAGE="JavaScript">
      alert("Distribution List successfully created.
      Please click OK to continue.");
      window.location="menu.asp";
   </SCRIPT>
<%
else
   'Failure!
%>
   <SCRIPT LANGUAGE="JavaScript">
      alert("Error! Error Number: <%=err.number%>
      Description: <%=err.Description%>");
      window.location="menu.asp";
   </SCRIPT>
<%
end if
%>
```

Adding and Removing Users from a Distribution List

ADSI provides another interface that you can take advantage of when working with directory objects that manage group memberships such as a distribution list: the *IADsGroup* interface. It provides methods such as *Add* and *Remove* that make it easy to add and remove members from the group. All you need to specify to these methods is the *AdsPath* that corresponds to the object you will either add or remove. To remove a user from a distribution list, you would just replace the *Add* method call with the *Remove* method.

This interface also provides a *Members* method that returns a collection of the current members of the group. The following code checks to see whether a user is already a member of a distribution list and, if the user is not, the code adds the user to the list:

```
<%
Set oIADs = GetObject("LDAP:")
bstr1 = Session("strDLName")
bstr2 = Session("bstr2")
bstr3 = Session("bstr3")
```

```
'Open the Exchange Server DS object
Set objDL = oIADs.OpenDSObject(bstr1, bstr2, bstr3, 0)
strAlias = Request.Form("USERSELECT")
bstr1 = strAlias
bstr2 = Session("bstr2")
bstr3 = Session("bstr3")

'Open the object in the Exchange Server DS that corresponds to
'the user
set objUser = oIADs.OpenDSObject(bstr1, bstr2, bstr3, 0)
booluserdoesnotexist=true
for each item in objDL.Members
   if item.ADSPath = strAlias then
      booluserdoesnotexist = false
   end if
next
if booluserdoesnotexist then
   'User does not exist in the Distribution List already, so add the user
   objDL.Add objUser.ADSPath
   if err.number = 0 then
%>
   <SCRIPT LANGUAGE="JavaScript">
      alert("Successfully added user to DL.
         Please click OK to continue.");
      window.location="menu.asp";
   </SCRIPT>
<%
   else
   'Failure!
%>
   <SCRIPT LANGUAGE="JavaScript">
      alert("Error! Error Number: <%=err.number %>
         Description: <%=err.Description%>");
      window.location="menu.asp";
   </SCRIPT>
<%
   end if
else
   'User does exist!
%>
   <SCRIPT LANGUAGE="JavaScript">
      alert("User already is a member of this DL.
         Please click OK to continue.");
      window.location="menu.asp";
   </SCRIPT>
<%
end if
%>
```

Displaying the Users in a Distribution List

The sample application allows you to display the users contained in a distribution list in an HTML table, as shown in Figure 15-11. This table is generated by using a For...Each construct to scroll through the collection returned by the *Members* method of the *IADsGroup* interface. After retrieving the object that corresponds to each member, the code checks the object class and displays the correct identifier for each member, such as mailbox, distribution list, or custom recipient. Remember that distribution lists can hold different types of objects. The code that displays users in a distribution list is shown here:

```
<HTML>
<HEAD>
<TITLE>Display users in Distribution List</TITLE>
</HEAD>
<BODY>
<h1>The members of this Distribution List are:</font></h1>
<hr>
<form METHOD="POST" NAME="INFO" ACTION="">
<input TYPE="button" VALUE="Back to Main Menu"
   OnClick='window.location="menu.asp";'>
   <input TYPE="button" VALUE="Select different container"
   OnClick='window.location="logon.asp?diffcont=1";'>
</FORM>
<P>
<TABLE BORDER=1 bgcolor="#79AA86">
<%
dim oIADs
dim MyContainer
dim objRecipients
dim item

on error resume next
err.clear
strDLName = Request.Form ("DLSELECT")
Set oIADs = GetObject("LDAP:")
bstr1 = strDLName
bstr2 = Session("bstr2")
bstr3 = Session("bstr3")
Set objDL = oIADs.OpenDSObject(bstr1, bstr2, bstr3, 0)
Response.Write "<TR><TD><B>Class</TD><TD><B>Display Name</TD>
<TD><B>Alias</TD><TD><B>Directory Name</TD></TR>"
for each item in objDL.Members
   set objitem = oIADs.OpenDSObject(item.ADSPath, bstr2, bstr3, 0)
   select case item.class
      case "organizationalPerson"
         Response.Write "<TD>MailBox</TD><TD>" & _
```

```
            objitem.get("cn") & "</TD><TD>" &objitem.get("uid") & _
            "</TD><TD>" & item.name & "</TD></TR>"
         case "Remote-Address"
            Response.Write "<TD>Custom Recipient</TD><TD>" & _
            objitem.get("cn") & "</TD><TD>" & objitem.get("uid") & _
            "</TD><TD>" & item.name & "</TD></TR>"
         case "groupOfNames"
            Response.Write "<TD>Distribution List</TD><TD>" & _
            objitem.get("cn") & "</TD><TD>" & objitem.get("uid") & _
            "</TD><TD>" & item.name & "</TD></TR>"
         case else
            Response.Write "<TD>" & item.class & "</TD><TD>" & _
            objitem.get("cn") & "</TD><TD>" & objitem.get("uid") & _
            "</TD><TD>" & item.name & "</TD></TR>"
      end select
   next
   %>
   </TABLE>
   </FONT>
```

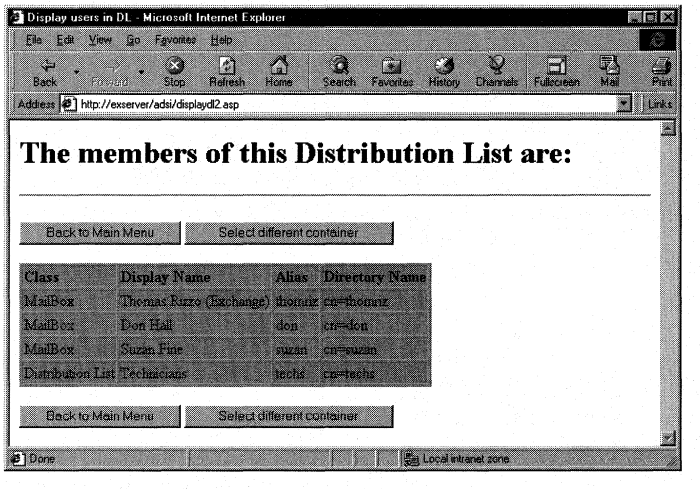


Figure 15-11. *An HTML table of the users in a specific distribution list. The table is generated by parsing the return value of the* Members *method of the* IADsGroup *interface.*

Creating a *Recipients* Container

The code that creates a *Recipients* container is probably the easiest to write in ADSI because you don't need to set many properties on it. Just remember to specify Container as the object class when creating the new *Recipients* container. The only gotcha when creating a *Recipients* container is that if you want the container to appear in the address book, you must set the Container-Info attribute to *-2147483647* or

&H80000001. The following code creates a *Recipients* container based on the values specified by the user:

```
<%
on error resume next
err.clear
strDisplayName = Request.Form("display")
strDirectoryName = Request.Form("dir")
Set oIADs = GetObject("LDAP:")
bstr1 = "LDAP://" + Session("Server") + "/ou=" + Session("OU") + _
   ",o=" + Session("O")
bstr2 = Session("bstr2")
bstr3 = Session("bstr3")
Set oContainer = oIADs.OpenDSObject(bstr1, bstr2, bstr3, 0)
Set oIADs = oContainer.Create("Container", "cn=" + strDirectoryName)
oIADs.Put "Container-Info", -2147483647
oIADs.Put "Admin-Display-Name", Cstr(strDisplayName)
oIADs.Put "rdn", CStr(strDirectoryName)
oIADs.SetInfo
if err.number = 0 then
   'Success!
%>
   <SCRIPT LANGUAGE="JavaScript">
      alert("Recipients Container successfully created.
         Please click OK to continue.");
      window.location="menu.asp";
   </SCRIPT>
<%
else
   'Failure!
%>
   <SCRIPT LANGUAGE="JavaScript">
      alert("Error! Error Number: <%=err.number %> Description:
         <%=err.Description%>");
      window.location="menu.asp";
   </SCRIPT>
<%
end if
%>
```

Displaying the Objects in a *Recipients* Container

The final code we'll look at in this section is similar to the code that displays the members of a distribution list. This code, however, displays the objects contained in a specific *Recipients* container in the directory. The first part of the code displays the list of available *Recipients* containers in the directory by scrolling through the available objects below the Organizational Unit (OU), or Exchange Server site, and then

parses out only the objects whose class name is Container. Since the configuration portion of Exchange Server, which contains settings for connectors, protocol settings, and monitor settings, also has an object class of Container, and you do not want users trying to display all the objects in the *Configuration* container, the code skips this *Configuration* container. The code then displays all the remaining *Recipients* containers so that the user can select the container whose contents they want to display.

```
on error resume next
err.clear
Set oIADs = GetObject("LDAP:")
bstr1 = "LDAP://" + Session("Server") + "/ou=" + Session("OU") + _
   ",o=" + Session("O")
bstr2 = Session("bstr2")
bstr3 = Session("bstr3")
Set oRecips = oIADs.OpenDSObject(bstr1, bstr2, bstr3, 0)
for each child in oRecips
   select case child.class
      case "Container"
         'Block out the Configuration container
         if instr(child.name,"Configuration") = 0 then
            Response.Write "<OPTION VALUE='" & child.Name & _
               "'>" & Replace(child.name,"cn=", "")
         end if
   end select
next
```

Next the code scrolls through all the objects in the selected container and displays the class, display name, alias, and directory name of the object by using a For Each ...Next loop. The result of this code is shown in Figure 15-12.

```
<%
on error resume next
err.clear
Set oIADs = GetObject("LDAP:")
bstr1 = "LDAP://" + Session("Server") + "/" + Request.Form("cn") + _
   ",ou=" + Session("OU") + ",o=" + Session("O")
bstr2 = Session("bstr2")
bstr3 = Session("bstr3")
Set oRecipObjects = oIADs.OpenDSObject(bstr1, bstr2, bstr3, 0)
Response.Write "<TR><TD><B>Type</TD><TD><B>Display Name</TD>
<TD><B>Alias</TD><TD><B>Directory Name</TD></TR>"
for each item in oRecipObjects
   set objitem =  oIADs.OpenDSObject(item.ADSPath, bstr2, bstr3, 0)
   select case item.class
      case "organizationalPerson"
         Response.Write "<TD>MailBox</TD><TD>" + _
```

(continued)

```
            objitem.get("cn") + "</TD><TD>" + objitem.get("uid") + _
            "</TD><TD>" + item.name + "</TD></TR>"
         case "Remote-Address"
            Response.Write "<TD>Custom Recipient</TD><TD>" + _
            objitem.get("cn") + "</TD><TD>" + objitem.get("uid") + _
            "</TD><TD>" + item.name + "</TD></TR>"
         case "groupOfNames"
            Response.Write "<TD>Distribution List</TD><TD>" + _
            objitem.get("cn") + "</TD><TD>" + objitem.get("uid") + _
            "</TD><TD>" + item.name + "</TD></TR>"
         case "Container"
            Response.Write "<TD>Container</TD><TD>" + _
            objitem.get("rdn") + "</TD><TD>" + + "</TD><TD>" + _
            item.name + "</TD></TR>"
         case else
            Response.Write "<TD>" + item.class + "</TD><TD>" + _
            objitem.get("cn") + "</TD><TD>" + objitem.get("uid") + _
            "</TD><TD>" + item.name + "</TD></TR>"
      end select
   next
   %>
```

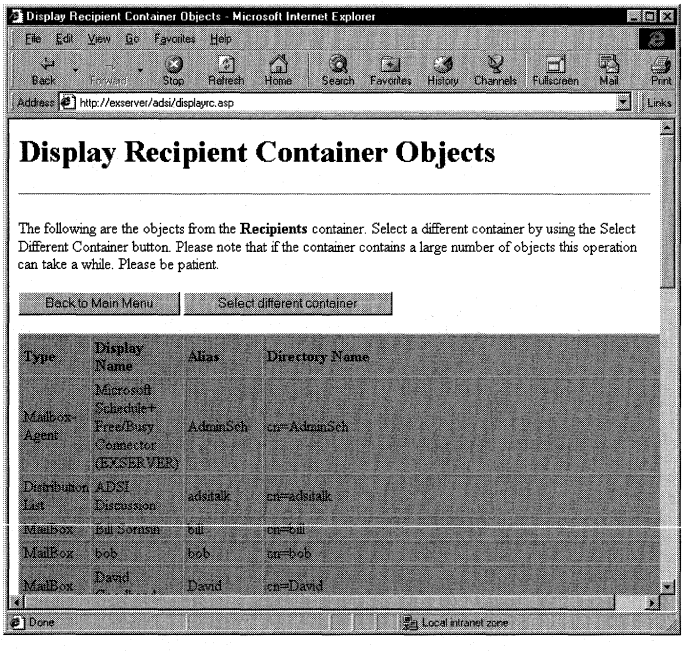


Figure 15-12. *The HTML table, which is dynamically generated from the object in a* Recipients *container.*

RAISING THE NUMBER OF RESULTS RETURNED FOR LDAP QUERIES

By default, Exchange Server will return only 100 results, so you might want to raise the number of results the Exchange server returns for LDAP queries. To do this, launch the Exchange Administrator program, and under the Configuration object for your site, select the Protocols icon. Double-click the LDAP (Directory) Site Defaults icon in the right pane. In the displayed Properties dialog box, click on the Search tab. Increase the number in the text box named Maximum Number Of Search Results Returned to the number of results you want to return from an LDAP query. If you do not raise this number and your LDAP query has more than the default 100 results, Exchange Server will not return any results.

ACTIVE DIRECTORY AND ADSI

To show how you can use your ADSI skills against the Exchange Server 5.5 directory and Active Directory, I've created a simple Web application that draws an organizational chart from both these directories. As you'll see in the code, you can easily query both directories. You'll also see that programming both directories using ADSI is similar. The only differences are the path you need to provide for LDAP binding and some of the property names. Obviously, more differences arise when you start performing more complex operations, such as setting securities and creating users.

Figure 15-13 shows the Logon page for the Web-based application. On this page, the user can select which directory he wants to connect to, thereby executing an ASP page that sets the correct credentials for the directory and then calls to the organizational chart ASP program. The JavaScript in the Web page displays or hides text boxes that ask for the alias or full user name.

> **NOTE** There is a difference between Exchange Server and Active Directory in the form of LDAP paths they expect. Exchange Server expects the alias of the user (for example, Thomriz), while Active Directory expects the name of the user (for example, Thomas Rizzo). Watch out for this in your applications.

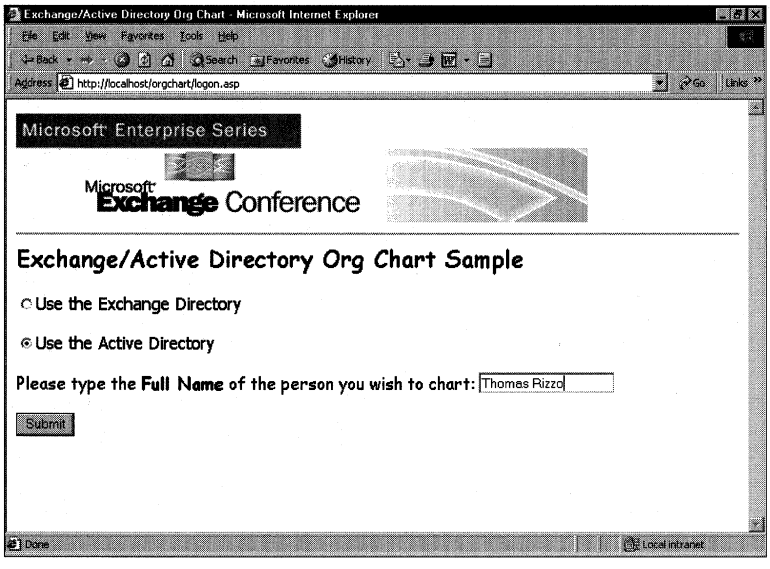


Figure 15-13. *The Logon page for the organizational chart ADSI sample.*

Figure 15-14 shows how to pull organizational information from the Exchange Server directory. The Web page uses a Java applet that displays the organizational chart and provides hyperlinks so that you can click on the hyperlink for another user to see that user's organizational information.

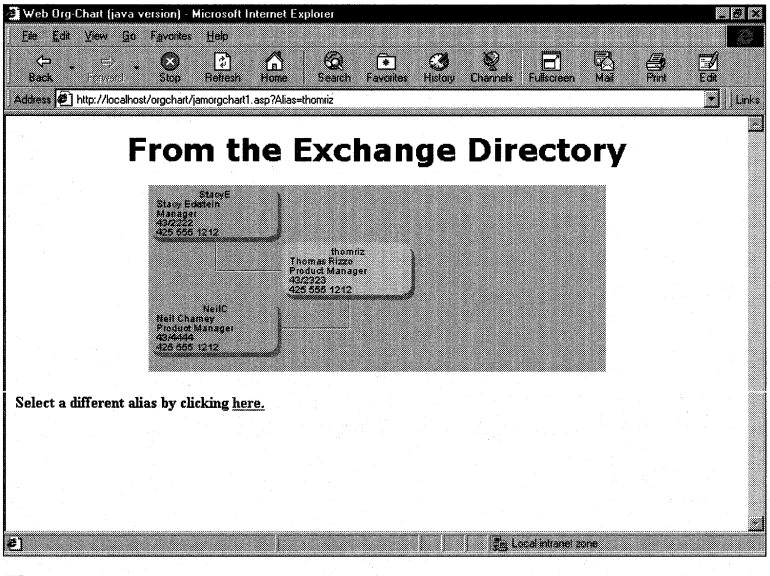


Figure 15-14. *The organizational information from an Exchange Server 5.5 directory.*

The code to perform this functionality is straightforward. It first connects to the Exchange server—specifically the object that corresponds to the alias passed in. Then, the code retrieves the user information, such as office location, phone number, title, and alias name. Next, the code tries to retrieve the same information for the user's manager by querying the manager property. The code then attempts to open the manager's object from the directory and retrieve the same information from that object. Finally, the code performs the same functions for the specified user's direct reports and passes this information into the Java applet. I've left some of the debugging statements in the code; you can uncomment them and see exactly what's happening at your leisure.

```
if Session("Type") = "EXCHANGE" then
   'Get if from Exchange!

   Set oIADs = GetObject("LDAP:")
   bstr1 = "LDAP://" + Session("Server") + "/cn=" + strUser + ",cn="


+ Session("cn") + ",ou=" + Session("OU") + ",o=" + Session("O")
   'bstr2 = Session("bstr2")
   'bstr3 = Session("bstr3")
   'Response.Write bstr1
   'Response.Write "bstr2=" & bstr2
   'Response.Write "bstr3=" & Session("bstr3")
   'Attempt an anonymous bind; if this doesn't work, you'll need
   'to uncomment the line following the one below
   Set objIADs = oIADs.OpenDSObject(bstr1, "","",0)
   'Set objIADs = oIADs.OpenDSObject(bstr1,bstr2,bstr3,0)

   'Get the current person's information
   strOffice = objIADs.Get("physicalDeliveryOfficeName")
   strTelephone = objIADs.Get("telephoneNumber")
   strTitle = objIADs.Get("title")
   strcn = objIADs.Get("cn")
   strrdn = objIADs.Get("rdn")

   strManager = objIADs.Get("manager")
   'Response.Write strManager
   strManagerPath = "LDAP://" & Session("Server") & "/" & strManager

   set oIADsManager = oIADS.OpenDSObject(strManagerPath,"","",0)
   'Anonymous bind; uncomment the line below if it doesn't work
   set oIADsManager =


oIADS.OpenDSObject(strManagerPath,bstr2,bstr3,0)
   strManagercn = oIADsManager.Get("cn")
```

(continued)

```
   strManagerrdn = oIADsManager.Get("rdn")
   'Response.Write "mcn=" & strManagercn
   'Response.Write "crdn=" & strManagerrdn
   strManagerOffice = oIADsManager.Get("physicalDeliveryOfficeName")
   strManagerTelephone = oIADsManager.Get("telephoneNumber")
   strManagerTitle = oIADsManager.Get("title")

   strReports = objIADs.GetEx("Reports")
   'Response.Write "Upper: " & UBound(strReports)
   'Response.Write "Lower: " & lbound(strReports)
      %>
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 3.2//EN">
<HTML>
<HEAD>
<Title>Web Org-Chart (java version)</Title>
</HEAD>
<center><font face=Verdana size=6><B>From the Exchange


Directory</B></center></font>


<p align="CENTER">

<applet
   codebase="java"
   code=JOrgChart.class
   id=JOrgChart
   width=480
   <% if UBound(strReports) = LBound(strReports) then
      iHeight = 0
      else
      iHeight = UBound(strReports)
      end if
   %>
   height=<%=(Int(((iHeight)/2)+3)*65)%>>
   <param name=HostName value="<% =ROOTURL %>?Alias=">
   <param name=Root value="<%=strManagerrdn%>?
            <%=strManagercn%>?
            <%=EmptyToNA(strManagerTitle)%>?
            <%=EmptyToNA(strManagerOffice)%>?
            <%=EmptyToNA(strManagerTelephone)%>">
   <param name=L1Node


value="<%=strrdn%>?<%=strcn%>?<%=strTitle%>?<%=strOffice%>?<%=strTelephon


e%>">
<%
```

```
   on error resume next

      for i = LBound(strReports) to UBound(strReports)
         strLogonName = _
            left(strReports(i),(instr(1,strReports(i),",")-1))

           'Get each DS object to return the friendly name
           strDirectPath = "LDAP://" & Session("Server") & _
              "/" & strReports(i)

           set oIADsReports =

oIADs.OpenDSObject(strDirectPath, "","",0)
         'Anonymous bind; uncomment if it doesn't work
         set oIADsReports = _
            oIADs.OpenDSObject(strDirectPath, bstr2,bstr3,0)


         strReportscn = oIADsReports.Get("cn")
         'Response.Write "cn" & strReportscn
         strReportsOffice =


oIADsReports.Get("physicalDeliveryOfficeName")
         strReportsTelephone = _
            oIADsReports.Get("telephoneNumber")


         strReportsTitle = oIADsReports.Get("title")
         strReportsrdn = oIADsReports.Get("rdn")

%>
   <param name=L2Node<%=i%> value="<%=strReportsrdn%>?
               <%=strReportscn%>?
               <%=EmptyToNA(strReportsTitle)%>?
               <%=EmptyToNA(strReportsOffice)%>?


<%=EmptyToNA(strReportsTelephone)%>">
<%
   Next

%>

</applet>
</p>
```

(continued)

```
<P>
<B>Select a different alias by clicking <a href="logon.asp">here.</B>

</BODY>
```

Figure 15-15 shows how to query similar information from Active Directory using LDAP, and the code for this query follows. Notice in the code that the LDAP path is a bit different from the one specified in the earlier code. Instead of specifying O (organization) and OU (organizational unit), you specify DC (domain controllers). If you want to simply grab information from Active Directory, you'll find your Exchange Server directory ADSI skills quite useful.

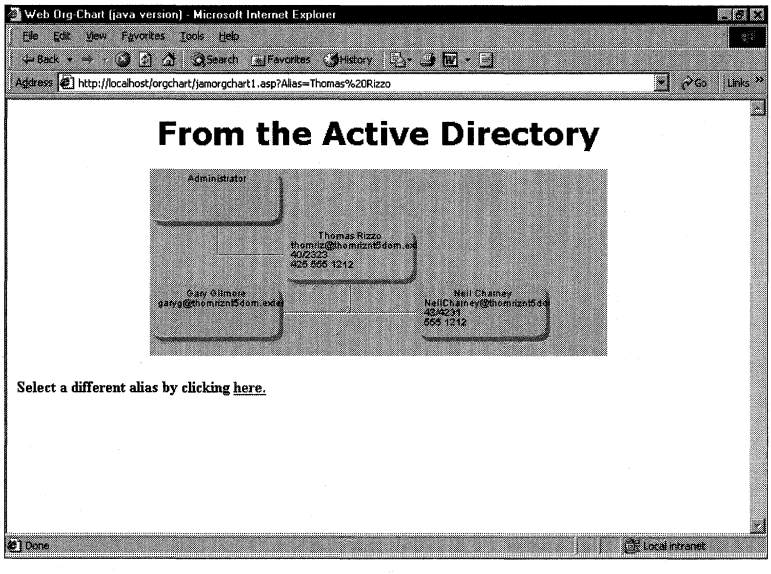


Figure 15-15. *Querying information from Active Directory.*

```
<% else %>

<%

'Get if from Active Directory

Set oIADs = GetObject("LDAP:")
   bstr1 = "LDAP://" + Session("Server") + "/cn=" + strUser + ",cn="


+ Session("cn") + ",dc=" + Session("Domain") + ",dc=" + Session("DC") +


",dc=" + Session("OU") + ",dc=" + Session("o")
   'bstr1 = "LDAP://" + Session("Server") + "/" +
```

```
"cn=thomriz,cn=Users,dc=ADCDEMO,DC=extest,DC=Microsoft,DC=com"
   'bstr1 = "LDAP://" + Session("Server") + "/" + Request.Form("cn")


+ ",ou=" + Session("OU") + ",o=" + Session("O")
   bstr2 = Session("bstr2")
   bstr3 = Session("bstr3")
   'Response.Write bstr1
   'Response.Write "bstr2=" & bstr2
   'Response.Write "bstr3=" & Session("bstr3")
   Set objIADs = oIADs.OpenDSObject(bstr1, bstr2, bstr3, 0)

   'Response.Write objIADs.Get("manager")
   'Get the current person's information
   strDisplayName = objIADs.Get("displayName")
   strOffice = objIADs.Get("physicalDeliveryOfficeName")
   strTelephone = objIADs.Get("telephoneNumber")
   strTitle = objIADs.Get("title")
   strcn = objIADs.Get("cn")
   'Try to retrieve the Mail property
   strmail = ""
   strmail = objIADs.Get("mail")
   if strmail = "" then
      'Try getting mailNickname
      strmail = objIADs.Get("mailNickname")
   end if
   'Response.Write "strcn= " & strcn
   strManager = objIADs.Get("manager")
   'Response.Write strManager
   strManagerPath = "LDAP://" & Session("Server") & "/" & strManager
   'Response.Write strManagerPath
   set oIADsManager =


oIADS.OpenDSObject(strManagerPath,bstr2,bstr3,0)
   strManagercn = oIADsManager.Get("cn")
   'Try to retrieve the Mail property
   strManagerMail = ""
   strManagerMail = oIADsManager.Get("mail")
   if strManagerMail = "" then
      strManagerMail = oIADsManager.Get("mailNickname")
   end if
   'Response.Write "mcn=" & strManagercn
   strManagerOffice = oIADsManager.Get("physicalDeliveryOfficeName")
   strManagerTelephone = oIADsManager.Get("telephoneNumber")
   strManagerTitle = oIADsManager.Get("title")
   strManagerDisplayName = oIADsManager.Get("displayName")
```

(continued)

```
   strReports = objIADs.GetEx("directReports")
      %>
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 3.2//EN">
<HTML>
<HEAD>
<Title>Web Org-Chart (java version)</Title>
</HEAD>
<center><font face=Verdana size=6><B>From Active


Directory</B></center></font>
<p align="CENTER">

<applet
   codebase="java"
   code=JOrgChart.class
   id=JOrgChart
   width=480
   <% if UBound(strReports) = LBound(strReports) then
      iHeight = 0
      else
      iHeight = UBound(strReports)
      end if
   %>
   height=<%=(Int(iHeight/2)+3)*65%>>
   <param name=HostName value="<% =ROOTURL %>?Alias=">
   <param name=Root value="<%=strManagerDisplayName%>?
            <%=strManagerMail%>?
            <%=EmptyToNA(strManagerTitle)%>?
            <%=EmptyToNA(strManagerOffice)%>?
            <%=EmptyToNA(strManagerTelephone)%>">
   <param name=L1Node


value="<%=strDisplayName%>?<%=strMail%>?<%=EmptyToNA(strTitle)%>?<%=Empty


ToNA(strOffice)%>?<%=EmptyToNA(strTelephone)%>">
<%

   for i = LBound(strReports) to UBound(strReports)

      strLogonName =


left(strReports(i),(instr(1,strReports(i),",")-1))

         'Get each DS object to return the friendly name
         strDirectPath = "LDAP://" & Session("Server") &
```

```
"/" & strReports(i)
         set oIADsReports =


oIADs.OpenDSObject(strDirectPath, bstr2,bstr3,0)
         strReportscn = oIADsReports.Get("cn")
         'Try to get the mail address
         strReportsMail = ""
         strReportsMail = oIADsReports.Get("mail")
         if strReportsMail = "" then
            strReportsMail =


oIADsReports.Get("mailNickname")
         end if
         strReportsOffice =


oIADsReports.Get("physicalDeliveryOfficeName")
         strReportsTelephone =


oIADsReports.Get("telephoneNumber")
         strReportsTitle = oIADsReports.Get("title")
         strReportsDisplayName =


oIADsReports.Get("displayName")

%>
   <param name=L2Node<%=i%> value="<%=strReportsDisplayName%>?
               <%=strReportsMail%>?
               <%=EmptyToNA(strReportsTitle)%>?
               <%=EmptyToNA(strReportsOffice)%>?



<%=EmptyToNA(strReportsTelephone)%>">
<%
   Next
%>

</applet>
</p>
<P>
<B>Select a different alias by clicking <a href="logon.asp">here.</B>
</BODY>
```

GETTING HELP WITH ADSI

This chapter provides you with examples of the most common features that you will want to program using ADSI with the Exchange Server directory, but you can program a lot more functionality. If you want to learn more about ADSI and the Exchange Server directory, my recommendation is to review the ADSI information in the MSDN Library under the section Platform SDK\Networking and Distributed Services\Active Directory\Active Directory Services Interfaces (ADSI). Also, to make it easier to visualize the relationships among the objects in the Exchange Server hierarchy, I would recommend using the program named Active Directory Browser (adsvw.exe), which is available on the companion CD. This program draws out the hierarchy of any directory service using ADSI and allows you to query and browse the objects and properties contained in a specific directory as shown in Figure 15-16. This program is an invaluable tool to help you discover the available objects and attributes contained in the Exchange Server directory.

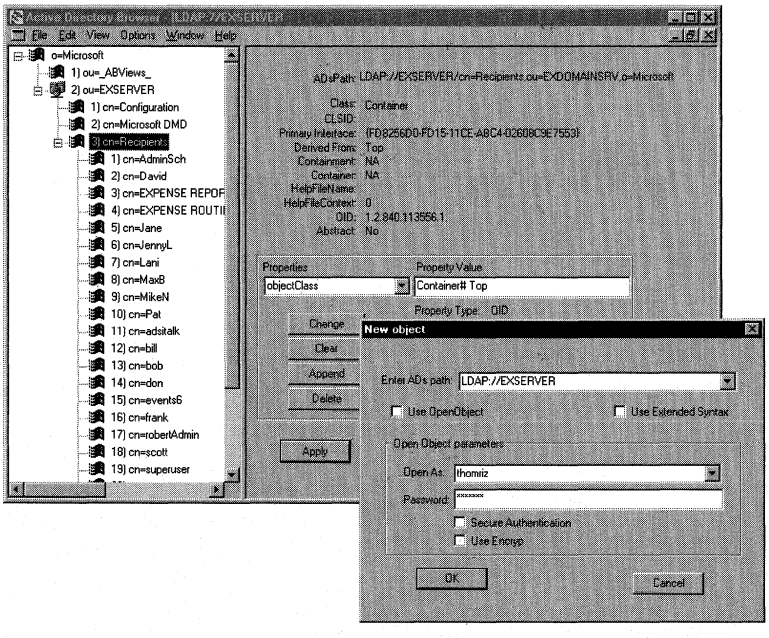


Figure 15-16. *The Active Directory Browser program. This program displays graphically the relationships and attributes of the objects contained in any directory that you can connect to using ADSI.*

LDP

As part of the Windows 2000 Resource Kit, LDP provides a graphical, low-level interface for LDAP operations such as Bind, Search, Modify, and Delete. You can use LDP with

any LDAP-compliant directory, such as Active Directory or the Exchange Server 5.5 directory. Figure 15-17 and Figure 15-18 show the LDP tool working with an Exchange Server 5.5 directory.

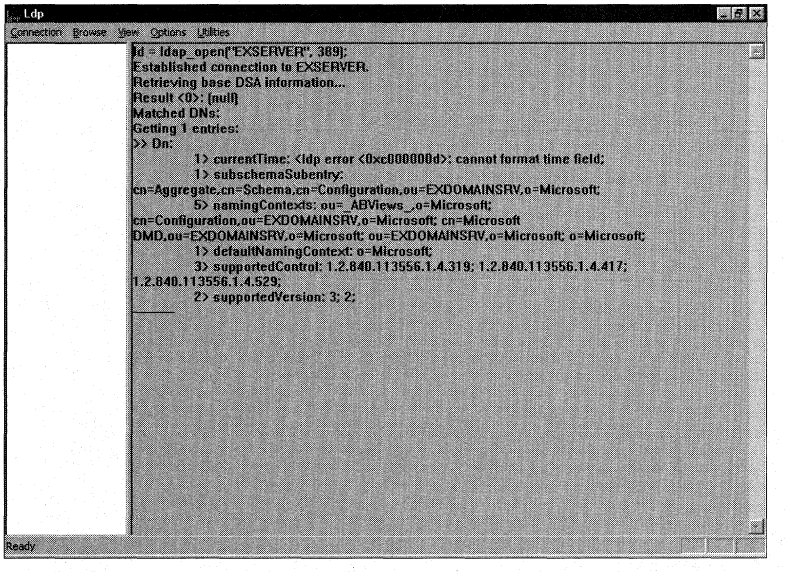


Figure 15-17. *LDP at work, connecting to an Exchange Server 5.5 directory. This tool is more low-level than the Active Directory Browser.*

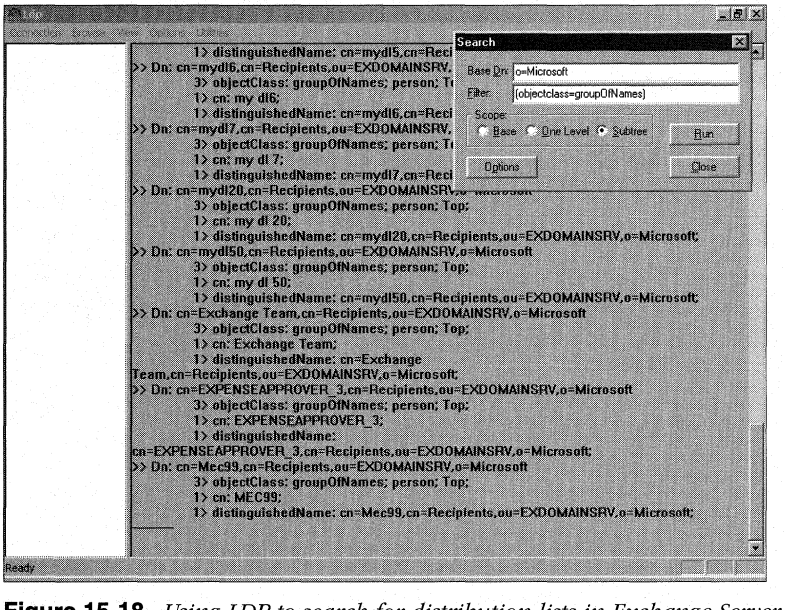


Figure 15-18. *Using LDP to search for distribution lists in Exchange Server 5.5.*

ADSI Edit

ADSI Edit is a Microsoft Management Console (MMC) snap-in that allows you low-level access to Active Directory by using ADSI interfaces. You can use this tool, shown in Figure 15-19, to browse and modify information in Active Directory.

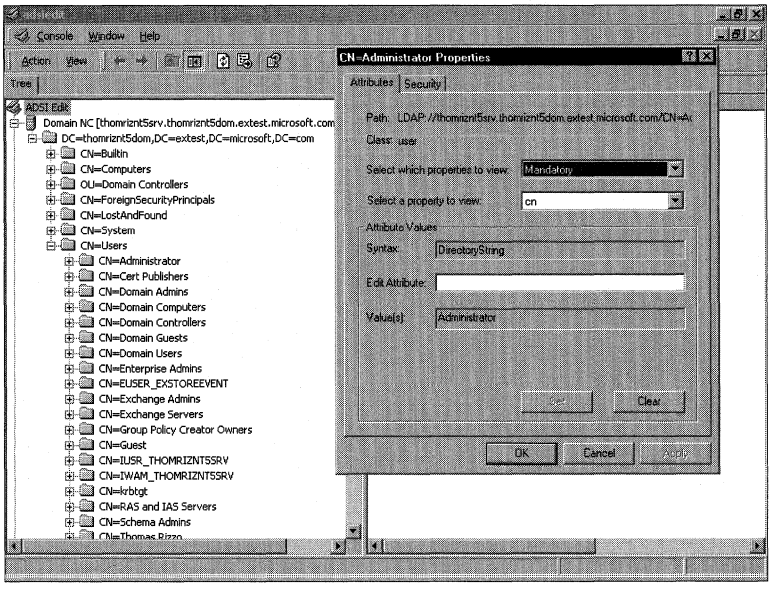


Figure 15-19. *ADSI Edit working with Windows 2000 Server.*

What About ADSI 2.5?

All the applications discussed in this chapter work with ADSI 2.5. While ADSI 2.5 greatly improves the capabilities of using ADSI with Windows 2000 Server, no great enhancements exist for using ADSI with Exchange 5.5 servers. I've included the redistributable version of the ADSI 2.5 object library as well as the programmer's guide on the companion CD so that you can get more information about ADSI 2.5. For those of you running Windows 2000, don't install the redistributable version of ADSI 2.5 since Windows 2000 comes with ADSI 2.5 installed.

Chapter 16

Enhancing Your Exchange Server Applications with COM Components

Throughout this book, we've looked at ways to develop applications that take advantage of the collaborative functionality of both Microsoft Exchange Server and Microsoft Outlook. And by now you're probably wondering how to work around the functionality *not* provided by the Microsoft Collaboration Data Objects (CDO) and Outlook object models discussed in this book. For example, how do you programmatically change permissions on folders from a Web page when users don't have Outlook on their machines? And how do you programmatically create, delete, and edit rules on the server so that your applications don't require the user to set rules through Outlook? Perhaps you can save time by using rules rather than the Event Scripting Agent for simple tasks like automatically forwarding new messages or returning unwanted messages to the sender. Or maybe you need the ability to create, delete, and modify Microsoft Windows NT accounts and programmatically set the security descriptors for newly created Active Directory Services Interfaces (ADSI) Exchange Server

accounts. CDO and the Outlook object model do not automatically provide these capabilities, but Microsoft has released three COM components that do: AcctCrt, ACL, and Rules. Because Exchange Server is extensible, you can use these components in your applications. You can pretty much guess the functionality of these components by their names:

- AcctCrt is an account creation component that was used in the ADSI application in Chapter 15. It allows you to create and associate Windows NT accounts with Exchange Server mailboxes so that you can use ADSI to create functional mailboxes in Exchange Server.
- ACL is a component that allows you to programmatically query, create, or change permissions on folders for users or distribution lists.
- Rules allows you to programmatically create, change, or delete rules on your folders.

These components ship as part of the Microsoft Platform Software Development Kit, but to make obtaining and using these components easier, I've included them on the companion CD in a folder named COM Components. I have also included sample applications from the Platform SDK that use the components. This chapter describes all three components and then shows a sample application named Project that uses two of them—Rules and ACL. Because AcctCrt was used in Chapter 15, we'll just look at some simple samples.

ACCTCRT COMPONENT

The AcctCrt component provides services for creating mailboxes in Exchange Server version 5.5 using ADSI. ADSI does not provide a mechanism to associate Windows NT accounts with Exchange Server mailboxes, but the AcctCrt COM component does. It also allows you to programmatically create and delete Windows NT accounts in your Windows NT domain. The AcctCrt component is very straightforward—it supports only six methods: *ChangeOwnerofSecDescriptor*, *NTAccountCreate*, *NTAccountDelete*, *GetSidFromName*, *GetNameFromSid*, and *GenerateSecDescriptor*. Let's take a look at how to create an instance of the AcctCrt component and use the methods it supports.

Creating an Instance of the AcctCrt Component

Creating an instance of the AcctCrt component is actually very easy. The ProgID for the component is MSExchange.AcctMgmt. The following line of code shows how to create an instance of the component and store it in a variable named *mntAcct*:

```
Set mntAcct = CreateObject("MSExchange.AcctMgmt")
```

Once you have an instance of the component, you can call its available methods.

Creating a Windows NT Account by Using the AcctCrt Component

The AcctCrt component contains a method named *NTAccountCreate* that allows you to programmatically add a new account to your Windows NT domain as long as you have the proper permissions in that domain. This method takes five arguments: *Domain*, *Login*, *Password*, *UserComment*, and *LocalGroup*. If you don't specify the domain name, AcctCrt defaults to using the local machine domain. If you don't specify the local group, the component automatically adds the user to the domain user group. The following code example shows you how to create a Windows NT account using *NTAccountCreate*:

```
Set mntAcct = CreateObject("MSExchange.AcctMgmt")
'NTAccountCreate takes Domain,Login,Password,UserComment,LocalGroup
MntAcct.NTAccountCreate "", "Test User", "password", "", ""
```

> **NOTE** You need the proper permissions to create the Windows NT account using *NTAccountCreate*. If you want to use the AcctCrt component from an Active Server Pages (ASP) page, you must authenticate the user by challenging the user's credentials in the browser; otherwise, ASP will use the anonymous Microsoft Internet Information Services (IIS) account to attempt creating the Windows NT user account. This attempt will most likely fail.

Deleting a Windows NT Account by Using the AcctCrt Component

Deleting a Windows NT Account using the AcctCrt component is as easy as creating a Windows NT account. To delete a Windows NT account, you use the *NTAccountDelete* method, which takes two arguments: *Domain* and *UserLogin*. If you don't specify the domain parameter, the component will use the local machine domain. Here's how you delete the account we added in the previous example code:

```
MntAcct.NTAccountDelete "","Test User"
```

Associating Windows NT Accounts with Exchange Server Mailboxes

Now that you've seen how to create and delete a Windows NT account, you need to learn how to associate a new Windows NT account with a mailbox and change a Windows NT account associated with a mailbox (in cases where you delete the Windows NT account). The AcctCrt component provides these capabilities through its remaining four methods. The following subroutine shows you all four of these methods.

```
Public Sub ManageSids(oldSid, oldDescriptor, NTDomain, _
NTAccountName, NewSid, NewDescriptor)
'Check to see if modifying existing SID
If IsEmpty(oldSid) then
   'Generate new SID
   mntAcct.GenerateSecDescriptor NTDomain, NTAccountName, _
      NewSecDescriptor
   'NewSecDescriptor now contains the new security descriptor.
   'We can then use the new security descriptor for our
   'mailbox in ADSI.
else
   mntAcct.GetNameFromSid NTDomain, (oldSid), oldNTDomain, _
      oldNTAccountName
   mntAcct.ChangeOwnerofSecDescriptor oldNTDomain, _
      oldNTAccountName, NewNTDomain, NewNTAccountName, _
      (oldSid), newSecDescriptor
   'Just to show how to use it
   mntAcct.GetSidFromName NewNTDomain, NewNTAccountName, testSid
end if
```

The *ManageSids* subroutine takes a number of parameters. If you pass in a security descriptor for the *oldSid* variable, the subroutine expects you to also pass in a domain name and an account name representing the new account you want to assign the security identifier (SID) to. The subroutine then modifies the security descriptor to reflect the new account and domain. It does this by retrieving the name of the old Windows NT domain and account for the security descriptor using the *GetNameFromSid* method. This method takes the domain name and a current SID as its parameters. (Be sure to enclose the variable for your SID in parentheses so that the value is passed by reference to the method. If you don't do this, you will receive an error.) The final two parameters are variables that the method fills in for you. They contain the Windows NT domain and the user name that the SID corresponds to.

SECURITY DESCRIPTORS AND SECURITY IDENTIFIERS

Understanding the difference between a security descriptor and a security identifier might be a little confusing. A security descriptor is a structure that contains the security information about an object, such as the owner and primary group, and users who have permissions to access the object. A SID is a structure that uniquely identifies a user or a group in Windows NT. Exchange Server requires the security descriptor to be placed in the NT-Security-Descriptor attribute and the SID to be placed in the Assoc-NT-Account attribute.

The *ManageSids* subroutine needs to change the ownership of the security descriptor to the new Windows NT domain and account passed in by the user by using the *ChangeOwnerOfSecDescriptor* method. This method takes six parameters. The first five are values that you pass in, such as the Windows NT domain and account, which is the current account for the security descriptor; the new Windows NT domain and account you want to change the descriptor to; and the descriptor you want to modify, enclosed in parentheses. The sixth parameter is a variable where the new security descriptor is returned. You can then take the new security descriptor and use it to update permissions on the Exchange Server mailbox to reflect a new user using ADSI.

I included the *GetSidFromName* method in the subroutine to show you how to use it. It retrieves the SID for a Windows NT account if you know only the name and domain of the account. Use *GetSidFromName* when you want to quickly find an account and retrieve its SID so that you can place it into an Exchange Server mailbox to assign ownership for the mailbox. *GetSidFromName* takes three parameters, the first two being the Windows NT domain and the account name that you want to find the SID for. Assuming the method could find the account, the third parameter is a variable that the method fills in with the value of the SID.

If you do not pass in the SID to the *ManageSids* subroutine, the subroutine assumes that you want to generate a new security descriptor for the Windows NT domain and account name that you passed to the parameter. The subroutine generates this new security descriptor by using the method *GenerateSecDescriptor*. The *GenerateSecDescriptor* method takes the Windows NT domain, the user name that you want to generate a security descriptor for, and a return variable for the new security descriptor. You can use this new security descriptor in your ADSI code for mailboxes you create.

RULES COMPONENT

Exchange Server includes excellent facilities to run rules on the server, which fire depending on the conditions you set. If you've looked at the Rules Wizard in Outlook, you have a sense of the complex rules you can create on your Exchange server. In the past, rules could only be created programmatically with C/C++, but with the Rules COM component, Microsoft Visual Basic developers can create complex rules for their application folders in Exchange Server. The Rules COM component provides an extensive object library, which we'll look at later.

Storing Rules

Before discussing the Rules component, we first must take a look at how rules are stored in the Exchange Server system. The Exchange Server system stores rules as

hidden messages inside folders. To find rules in the CDO HiddenMessages collection for a folder, search for the message class IPM.Rule.Message. This message class specifies that the item is a rule item, and the properties on the item contain the various conditions for the rule. The easiest way to see the different rules is to use the MDB Viewer Test Application (MDBVUE) tool included on the companion CD. This tool allows you to see hidden messages in folders as well as retrieve the properties of all items in the Exchange Server store. Figure 16-1 shows a screen from the MDB Viewer Test Application.

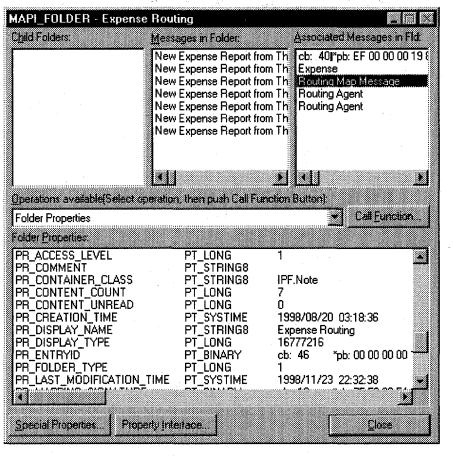


Figure 16-1. *The MDB Viewer Test Application provides functionality to investigate the objects stored in your Exchange Server system. This tool can even be used to investigate the properties on rule items stored as hidden messages in your folders.*

Creating an Instance of the Rules Component

To create an instance of this component, all you need to do is pass the ProgID for the component, MSExchange.Rules, to the *CreateObject* method, as shown in the following line of code. The Rules component contains other instantiable objects, which you will see later in this chapter, that correspond to conditions you can set for the component.

```
Set myRules = CreateObject("MSExchange.Rules")
```

Using the Rules Component

The easiest way to show you how to use the Rules component is to step through some snippets of code. These snippets show you many of the objects and methods that constitute the component. The first sample we'll review, written in Visual Basic, illustrates the major issues you'll confront when using the Rules component to create rules that compare a single property to a specified value. The sample creates a

new rule, named *Imp Rule*, that looks for low-importance messages sent to the Inbox. When the rule finds a low-importance message in the Inbox, it moves it to a subfolder of the Inbox named *To Me*.

```
Dim oSession As MAPI.Session
Set oSession = CreateObject("MAPI.Session")
oSession.Logon
Set oRules = CreateObject("MSExchange.Rules")

'This can also be Public Folders.
'You need Owner permissions on the folder to create
'and enable a rule.
oRules.Folder = oSession.Inbox

'Set the property value for a condition in the rule
'to the importance of the message
Set ImpProp = CreateObject("MSExchange.PropertyValue")
ImpProp.Tag = CdoPR_IMPORTANCE
ImpProp.Value = 0

'Set a condition so that this rule
'looks for all nonimportant messages
Set ImpCond = CreateObject("MSExchange.PropertyCondition")
ImpCond.Value = ImpProp
ImpCond.PropertyTag = CdoPR_IMPORTANCE
ImpCond.Operator = 7

Set oFolder = oSession.Inbox.Folders("To Me")

'Set the action for the rule
Set oAction = CreateObject("MSExchange.Action")
oAction.ActionType = 1
oAction.Arg = oFolder

'Create the actual rule
Set oRule = CreateObject("MSExchange.Rule")
oRule.Name = "Imp Rule"
oRule.Actions.Add 1, oAction
oRule.Condition = ImpCond

'Add it to the Rules collection
oRules.Add , oRule
oRules.Update

oSession.Logoff
```

When using the Rules component, you need to make sure you have a valid CDO session with the server so that you can pass in a CDO Folder object to the *Folder*

property on the Rules component, which tells the Rules component which folder to create your rules in.

Your next step is to set up conditions that must be met by the incoming message to make the rule fire. The Rules component supports many similar types of conditions. We'll cover the major ones you'll use: Bitmask, Comment, CompareProps, Content, Exists, Logical, Property, Size, and Sub. We'll look in detail at Bitmask, Content, Logical, and Property.

In the preceding code example, you saw a property condition in action. To create a PropertyCondition object, the code calls the *CreateObject* method with the ProgID for a property condition. After you create the condition object, you have to set its properties. For the PropertyCondition object, you need to set the *Value*, *PropertyTag*, and *Operator* properties.

For the *Value* property, you have to pass a PropertyValue object that contains the value you want to compare to the desired property. For this reason, before a PropertyCondition object is created, a PropertyValue object is created, with its *Tag* property set to the CDO property identifier that we're interested in, *CdoPR_IMPORTANCE*. The *Value* property for the object is set to the value we want satisfied by the condition. In this case, the *Value* property is set to *0*, which specifies a low-importance message in CDO.

> **NOTE** When I tested the code for the Rules component on different machines, I sometimes had trouble getting the importance and sensitivity properties to be identified. If you have trouble getting your rule to fire, try testing your rule with different message properties.

Now that we have a valid PropertyValue object, we can pass it to the *Value* property for the PropertyCondition object. Then we need to set the *PropertyTag* property on our condition. This property should contain the same property identifier as the PropertyValue object.

Next, the *Operator* property must be set. The *Operator* property can have seven possible values, described in Table 16-1.

Name	*Value*	*Description*
REL_GE	*1*	Greater than or equal to
REL_GT	*2*	Greater than
REL_LE	*3*	Less than or equal to
REL_LT	*4*	Less than
REL_NE	*5*	Not equal to
REL_RE	*6*	Like
REL_EQ	*7*	Equal to

Table 16-1. *Values for the* Operator *property.*

We want the importance level for the *Operator* property to equal the value specified earlier for the PropertyValue object. The code sets the *Operator* value to *7*, which is the EQUALTO operator.

The code's next step is to create the Action object, which contains the action that Exchange Server should take if the property condition equals the property value we specified. The Action object has two properties we need to set: *ActionType* and *Arg*. *ActionType* specifies the type of action to take on the item. For example, you can automatically delete the item, move the item to a different folder, or bounce the item back to the person who submitted it. Table 16-2 shows you the possible values for the *ActionType* property.

Name	*Value*	*Description*
ACTION_MOVE	*1*	Move the message to the folder object specified in *Arg*.
ACTION_COPY	*2*	Copy the message to the folder object specified in *Arg*.
ACTION_DELETE	*3*	Delete the message.
ACTION_REPLY	*4*	Respond to the message with the message specified in *Arg*.
ACTION_OOFREPLY	*5*	Respond to the message with the Out-of-office message specified in *Arg*.
ACTION_FORWARD	*6*	Forward the message to the recipient list specified in *Arg*.
ACTION_DELEGATE	*7*	Delegate the message to the recipient list specified in *Arg*.
ACTION_BOUNCE	*8*	Return the message to the sender for the reason specified in *Arg*.
ACTION_TAG	*9*	Tag the message to set the property specified in *Arg*.
ACTION_MARKREAD	*10*	Mark as read.
ACTION_DEFER	*11*	Defer action.

Table 16-2. *Values for the* ActionType *property.*

In the code, the *ActionType* property was set to *1*, which moves the item to the folder we specify in the *Arg* property. Depending on the value specified for *ActionType*, you might need to set the *Arg* property to a value. If you pick the delete action, you do not have to specify the *Arg* property on your Action object.

As you can see, the *Arg* property and the *Action* property work together to create the action for your rule. In our example, the code finds the specific folder we're moving items to by using CDO, and then it sets the Action object's *Arg* property to the CDO Folder object. If we do not specify this folder, the rule will not work.

After we have a condition and an action for the rule, we need to create the actual rule by creating a Rule object. The Rule object has a large number of properties that you can use, but we'll look at only a subset of them—*Actions*, *Name*, and *Condition*.

The *Name* property contains the friendly name for the rule. The *Action* property returns the Actions collection, which contains all the Action objects for the rule. We add a new rule to the collection by using the *Add* method on the collection. The *Add* method takes two parameters, the first being the position in the collection where the Action object should be placed. You can have multiple actions for a rule such as a forward action and a reply action. The second parameter for the *Add* method is the Action object you want to add to the collection.

The final property we need to set on the Rule object to successfully create the object is the *Condition* property. The *Condition* property should be set to the condition object we created for the rule. As you will see later, you can have multiple conditions for a rule, but there is a catch—you need to link all the conditions together using another type of object, the LogicalCondition object.

Now that the properties for the new Rule object are set, all we need to do is add the object to the Rules collection. To do this, we pass our Rule object to the *Add* method on the Rules collection. The first parameter of this method, which is blank in the code, is an optional integer that specifies the position before the insertion point for the new rule. Since this parameter is not specified, the new rule is inserted at the end of the collection. If you do specify a position for this property, you must update the indices for the Rules object by calling the *UpdateIndices* method and then the *Update* method on the object.

The second parameter of the *Add* method is the object that contains the new Rule object you want to add. We already created and set the properties of this object in the code, so that's it! We just created a new rule. Now we're going to look at some of the other condition objects we can use to set more complex conditions for our rules. Because a lot of the steps are similar for these other types of conditions, I'm going to highlight only the steps that differ and are required for each condition type.

Specifying a Logical Condition

Most of the time, when you create rules, you will not use only one property as your condition. You'll have multiple conditions, such as specifying only those messages that are of low importance and sent directly to you. To create multiple conditions, you need to use the LogicalCondition object in conjunction with the other condition objects. The next example shows how a LogicalCondition object is used in conjunction with two PropertyCondition objects to create a rule that checks to see whether messages are of low importance and sent directly to you. If the rule finds a message that meets these conditions, the message is moved to the To Me subfolder of the Inbox.

```
Dim oSession As MAPI.Session
Set oSession = CreateObject("MAPI.Session")
oSession.Logon
Set oRules = CreateObject("MSExchange.Rules")

'This can also be Public Folders.
'You need Owner permissions on the folder to create
'and enable a rule.
oRules.Folder = oSession.Inbox

'Set the property value for a condition in the rule
'to the importance of the message
Set ImpProp = CreateObject("MSExchange.PropertyValue")
ImpProp.Tag = CdoPR_IMPORTANCE
ImpProp.Value = 0

'Set a condition so that this rule
'looks for all nonimportant messages
Set ImpCond = CreateObject("MSExchange.PropertyCondition")
ImpCond.Value = ImpProp
ImpCond.PropertyTag = CdoPR_IMPORTANCE
ImpCond.Operator = 7

'Set a property value for messages sent to me
Set MeProp = CreateObject("MSExchange.PropertyValue")
MeProp.Tag = CdoPR_MESSAGE_TO_ME
MeProp.Value = True
'Set a condition so that this rule
'looks for all messages sent to me
Set MeCond = CreateObject("MSExchange.PropertyCondition")
MeCond.Value = MeProp
MeCond.PropertyTag = CdoPR_MESSAGE_TO_ME
MeCond.Operator = 7

Set LogCond = CreateObject("MSExchange.LogicalCondition")
LogCond.Operator = 1
LogCond.Add ImpCond
LogCond.Add MeCond

Set oFolder = oSession.Inbox.Folders("To Me")

'Set the action for the Rule object
Set oAction = CreateObject("MSExchange.Action")
oAction.ActionType = 1
oAction.Arg = oFolder

'Create the actual rule
Set oRule = CreateObject("MSExchange.Rule")
oRule.Name = "Imp Rule"
```

(continued)

```
oRule.Actions.Add 1, oAction
oRule.Condition = LogCond

'Add it to the Rules collection
oRules.Add , oRule
oRules.Update

oSession.Logoff
```

In the code, two PropertyCondition objects and two PropertyValue objects are created to specify the conditions for the rule. To link the two conditions, the code creates a LogicalCondition object. The LogicalCondition object is actually a collection of other condition objects from which you can add or delete objects.

To add the two conditions to the LogicalCondition object, the code uses the *Add* method of the LogicalCondition object. The *Add* method takes a Condition object as its parameter and will add the object to the collection. Once all the condition objects are added to the collection, the logic that links the two or more conditions must be set. To do this, we use the *Operator* property on the LogicalCondition object. This property has three possible values: *L_AND (1)*, *L_OR (2)*, and *L_NOT (3)*. Since we want all messages sent directly to the person and messages of low importance to be the only messages moved to the folder, we set the *Operator* property on the LogicalCondition object to be *1*, or *L_AND*, which causes Exchange Server to fire the rule only if both conditions are met on the item.

Searching for Specific Content

The Rules component provides the ContentCondition object so that you can search for specific content. This object allows you to search for specific text in either the message body or the message subject. You can use this searching capability to fire off rules that perform specific actions. For example, you can use the ContentCondition object to create a simple profanity filter for a discussion application. If any offensive words are placed into the message body or message subject, you can automatically delete the message or move it to a folder for an administrator to look at.

To show you how to use the ContentCondition object, the code example we have been working with has been updated to search the message body of incoming items for the phrase *New Policy*. Now our rule fires only when an item is of low importance, sent directly to the person, and contains the phrase *New Policy* in the message body. If the rule finds a message that meets these conditions, the message is moved to the To Me subfolder of the Inbox. Here is the code that creates this rule:

```
Dim oSession As MAPI.Session
Set oSession = CreateObject("MAPI.Session")
oSession.Logon
Set oRules = CreateObject("MSExchange.Rules")
```

```
'This can also be Public Folders.
'You need Owner permissions on the folder to create
'and enable a rule.
oRules.Folder = oSession.Inbox

'Set the property value for a condition in the rule
'to the importance of the message
Set ImpProp = CreateObject("MSExchange.PropertyValue")
ImpProp.Tag = CdoPR_IMPORTANCE
ImpProp.Value = 0

'Set a condition so that this rule
'looks for all nonimportant messages
Set ImpCond = CreateObject("MSExchange.PropertyCondition")
ImpCond.Value = ImpProp
ImpCond.PropertyTag = CdoPR_IMPORTANCE
ImpCond.Operator = 7
'Set a property value for messages sent to me
Set MeProp = CreateObject("MSExchange.PropertyValue")
MeProp.Tag = CdoPR_MESSAGE_TO_ME
MeProp.Value = True

'Set a condition so that this rule
'looks for all messages sent to me
Set MeCond = CreateObject("MSExchange.PropertyCondition")
MeCond.Value = MeProp
MeCond.PropertyTag = CdoPR_MESSAGE_TO_ME
MeCond.Operator = 7

'Set a property value for messages with New Policy
Set ContProp = CreateObject("MSExchange.PropertyValue")
ContProp.Tag = CdoPR_BODY
ContProp.Value = "New Policy"

'Set a condition so that this rule
'looks for all with the New Policy keywords
Set ContCond = CreateObject("MSExchange.ContentCondition")
ContCond.Value = ContProp
ContCond.PropertyType = CdoPR_BODY
ContCond.Operator = 1   'Substring

Set LogCond = CreateObject("MSExchange.LogicalCondition")
LogCond.Operator = 1
LogCond.Add ImpCond
LogCond.Add MeCond
LogCond.Add ContCond

Set oFolder = oSession.Inbox.Folders("To Me")
```

(continued)

```
'Set the action for the Rule object
Set oAction = CreateObject("MSExchange.Action")
oAction.ActionType = 1
oAction.Arg = oFolder

'Create the actual rule
Set oRule = CreateObject("MSExchange.Rule")
oRule.Name = "Imp Rule"
oRule.Actions.Add 1, oAction
oRule.Condition = LogCond

'Add it to the Rules collection
oRules.Add , oRule
oRules.Update

oSession.Logoff
```

As you can see from the code, to successfully create a ContentCondition object, you must perform two steps. First you must create a PropertyValue object and fill in its properties with the CDO property you're interested in searching, and then you must fill in three properties on the ContentCondition object—*Value*, *PropertyType*, and *Operator*. Set the *Value* property to contain the PropertyValue object that you create. The *Value* property tells the ContentCondition object the value for which the rule should search in incoming messages. Set the *PropertyType* property to the same CDO property set for the *Value* property on the PropertyValue object. *PropertyType* tells the ContentCondition object which CDO property to search in for the value. The *Operator* property contains a hex value that specifies the type of search to perform for the specified value. This search can be an exact match, a substring, or a prefix. You can have only one of these three types of searches. Table 16-3 shows the settings for the *Operator* property. However, you can combine the last three settings in the table—IGNORECASE, IGNORENONSPACE, and LOOSE—with the search type. For example, to specify a search that looks for a substring and ignores cases and nonspaces, you would set the *Operator* property to *&H30001*.

Name	*Hex Value*	*Description*
FULLSTRING	*0*	Full string
SUBSTRING	*1*	Substring
PREFIX	*2*	Prefix
IGNORECASE	*10000*	Ignore case
IGNORENONSPACE	*20000*	Ignore nonspace
LOOSE	*40000*	Ignore high bits (maps Unicode to corresponding ANSI values)

Table 16-3. *Values for the* Operator *property.*

Searching for a Particular Bitmask

Sometimes you'll want to retrieve a particular property on a message—most commonly, the *CdoPR_MESSAGE_FLAGS* property—to see if the property meets a certain criterion. The *CdoPR_MESSAGE_FLAGS* property contains a bitmask that describes whether the message has attachments or was sent from the Internet. The following example adds to the rule we've been creating a condition that searches incoming messages to determine whether they have attachments:

```
Dim oSession As MAPI.Session
Set oSession = CreateObject("MAPI.Session")
oSession.Logon
Set oRules = CreateObject("MSExchange.Rules")

'This can also be Public Folders.
'You need Owner permissions on the folder to create
'and enable a rule.
oRules.Folder = oSession.Inbox

'Set the property value for a condition in the rule
'to the importance of the message
Set ImpProp = CreateObject("MSExchange.PropertyValue")
ImpProp.Tag = CdoPR_IMPORTANCE
ImpProp.Value = 0

'Set a condition so that this rule
'looks for all nonimportant messages
Set ImpCond = CreateObject("MSExchange.PropertyCondition")
ImpCond.Value = ImpProp
ImpCond.PropertyTag = CdoPR_IMPORTANCE
ImpCond.Operator = 7

'Set a property value for messages sent to me
Set MeProp = CreateObject("MSExchange.PropertyValue")
MeProp.Tag = CdoPR_MESSAGE_TO_ME
MeProp.Value = True

'Set a condition so that this rule
'looks for all messages sent to me
Set MeCond = CreateObject("MSExchange.PropertyCondition")
MeCond.Value = MeProp
MeCond.PropertyTag = CdoPR_MESSAGE_TO_ME
MeCond.Operator = 7

'Set a property value for messages with New Policy
Set ContProp = CreateObject("MSExchange.PropertyValue")
ContProp.Tag = CdoPR_BODY
ContProp.Value = "New Policy"
```

(continued)

```
'Set a condition so that this rule
'looks for all with the New Policy keywords
Set ContCond = CreateObject("MSExchange.ContentCondition")
ContCond.Value = ContProp
ContCond.PropertyType = CdoPR_BODY
ContCond.Operator = 1   'Substring

'Create a condition for messages with attachments
Set BitCond = CreateObject("MSExchange.BitmaskCondition")
BitCond.Value = 16
BitCond.PropertyTag = CdoPR_MESSAGE_FLAGS
BitCond.Operator = 2

Set LogCond = CreateObject("MSExchange.LogicalCondition")
LogCond.Operator = 1
LogCond.Add ImpCond
LogCond.Add MeCond
LogCond.Add ContCond
LogCond.Add BitCond

Set oFolder = oSession.Inbox.Folders("To Me")

'Set the action for the Rule object
Set oAction = CreateObject("MSExchange.Action")
oAction.ActionType = 1
oAction.Arg = oFolder

'Create the actual rule
Set oRule = CreateObject("MSExchange.Rule")
oRule.Name = "Imp Rule"
oRule.Actions.Add 1, oAction
oRule.Condition = LogCond

'Add it to the Rules collection
oRules.Add , oRule
oRules.Update

oSession.Logoff
```

As you can see in the code, when creating a bitmask condition, you need to set three properties on the object: *Value*, *PropertyTag*, and *Operator*. The *Value* property takes the value with which the *PropertyTag* value is masked. In this example, we placed the value *16* in the *Value* property to create a rule that looks for attachments. The *PropertyTag* value specifies the CDO property that you want to mask with the *Value* property. In this example, we used the *CdoPR_MESSAGE_FLAGS* property. The *Operator* property specifies the bitmask operator for the property and can take one of two values: *B_EQZ (1)* or *B_NEZ (2)*. *B_EQZ* creates a bitmask and checks to see whether the returned value is zero. *B_NEZ* creates a bitmask and checks to see

whether the returned value is nonzero. In this example, we created the bitmask and checked to see whether the value was nonzero.

In this section, we've seen some of what the Rules component can do. In the section titled "Project Application," we'll see how to create rules that fire on all incoming messages. The Rules component has other conditions that you can set and other capabilities—it doesn't just create rules but also reads and modifies existing rules. To learn more about these other capabilities, refer to the Exchserv.chm file on the companion CD.

ACL COMPONENT

To provide you with the ability to query and modify access control lists (ACLs) on folders, Microsoft offers an ACL COM component. This component works in conjunction with the CDO library to allow you to change permissions on folders for which you are the owner. Before the ACL component was available, you had to either manually hack the hex properties on the folder or write a C/C++ program to manipulate folder permissions by using Extended Messaging Application Programming Interface (MAPI). The first of those solutions is not desirable since future versions of the product could break your code and make your code more error prone, and the second solution is hard to implement unless you know the C/C++ programming language. The Project sample application, which we're about to examine, uses the ACL component extensively, so you'll learn about it in context.

PROJECT APPLICATION

To help you learn how to use the Rule and ACL components in a full application rather than in code snippets, I put together an application named Project. This application shows you the new COM components and ties together the CDO objects that we've learned about, such as the core messaging and rendering CDO objects. From this application, you can get an idea of how to build your own fully featured CDO applications. First, we'll examine what the application does and highlight some of the programming behind it. I won't go over the CDO portions of the code in great detail, so you should set up the application and browse through its sample code to get a better understanding of the CDO library.

The Project application allows users to collaborate in a virtual project workspace on the Web, where they can create new folders in which they post documents or messages. In these folders, users can discuss items and view information using different view types. Project owners can modify permissions for project members either at the project level or at the individual folder level, and they can set up autonotification for team members when new items are placed in the project folders.

Setting Up the Project Application

Before you can install the application, you must have a Windows NT 4.0 Server or Windows 2000 Server with certain software installed. Table 16-4 describes the installation requirements for the application.

Minimum Software Requirements	*Installation Notes*
Exchange Server 5.5 Service Pack 1 with Outlook Web Access	Service Pack 3 is recommended.
Internet Information Server 3.0 or later with Active Server Pages	Internet Information Services 4.0 or later is recommended.
CDO library (cdo.dll), CDO Rendering library (cdohtml.dll)	Exchange Server 5.5 Service Pack 1 installs CDO library 1.21 and CDO Rendering library 1.21. Outlook installs CDO library 1.21.
Microsoft Posting Acceptor	Posting Acceptor 1.01 is available as a subcomponent of the Microsoft Site Server Express 2.0 component with the Windows NT 4.0 Option Pack. The Posting Acceptor DLL (Cpshost.dll) is also available with the Project files. If you do not have the Posting Acceptor properly installed, you will not be able to upload attachments to new messages.
For the client:	
Microsoft Internet Explorer 4.0, Outlook	You can run the client software on the same machine or on a separate machine.

Table 16-4. *Installation requirements for the Project application.*

To install the Project application, first copy the Project folder from the companion CD to your Web server where you want to run the application. Start the IIS administration program. Create a virtual directory that points to the location where you copied the Project files, and name the virtual directory *project*. If you want to be able to attach files to messages, make sure you enable the Execute and Write permissions for the virtual directory. Execute permissions are necessary because the Posting Acceptor DLL (Cpshost.dll) in the Project folder is used for uploading files. Write permissions are necessary because attachments are uploaded by default to the Temp subfolder of Project. Since enabling both Execute and Write permissions for a directory is potentially dangerous, you will want to set up a completed application to avoid this configuration.

Included with the Project files is a file named Project.pst. Make sure the read-only flag for this file is unchecked. Launch Outlook, and from the File menu, point to Open and then choose Personal Folders File (.pst). In the Open Personal Folders dialog box, select the Project.pst file, and click OK. In the Outlook Folder List, expand the Project Application file folder. While holding the Ctrl key, copy the Projects folder to All Public Folders. Right-click on the Projects public folder and select Properties. On the Permissions tab of the Properties dialog box, set the permissions for users. For users who can create projects, assign at least Publishing Author permissions.

> **NOTE** You must copy the Projects folder to All Public Folders or the application will not work. If you cannot install the application there, you can modify the code contained in the Project application so that it looks for the folder in another location, or you can retrieve the folder by using its EntryID.

Included with the Project files is a Components folder, which contains the AcctCrt, Rules, and ACL components. Since AcctCrt is not used in this application, just register Rule.dll and ACL.dll using the Regsvr32 utility:

```
regsvr32 rule.dll
regsvr32 acl.dll
```

To start the Project application, enter the following URL in Internet Explorer: *http://*yourservername*/project*.

> **NOTE** When the Project application was tested on different configurations, some pages initially were not displayed. If this happens to you, try clicking the Refresh button or pressing F5.

Architecture of the Application

The architecture for the Project application is centered around Public Folders. The Projects public folder contains the main project folders, as shown in Figure 16-2. Under each main project folder are the folders that contain the project's content, such as a document database or a discussion group.

To create projects, the user must have permission to create subfolders under the Projects folder. If a user has this permission, a Create A New Project hyperlink (also referred to as a button) appears on the user's Projects page (informally known as the project workspace page). The Projects page is shown in Figure 16-3. This page shows all projects for which the user has top-level permission to view items.

When the user clicks on the Create A New Project button, an HTML wizard, which walks the user through creating a new project, appears. The steps in the wizard ask the user to name the project, write a project description, and set default permissions for both authenticated and anonymous users. The fourth page in the wizard is shown in Figure 16-4.

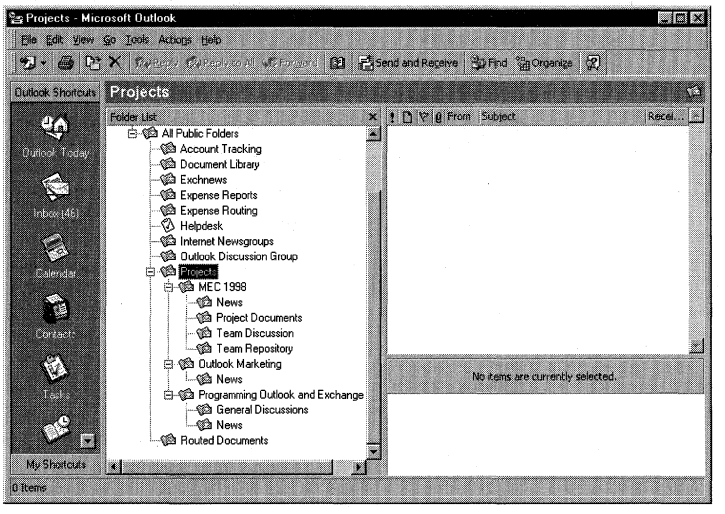


Figure 16-2. *The folder structure for the Project application, shown in Microsoft Outlook.*

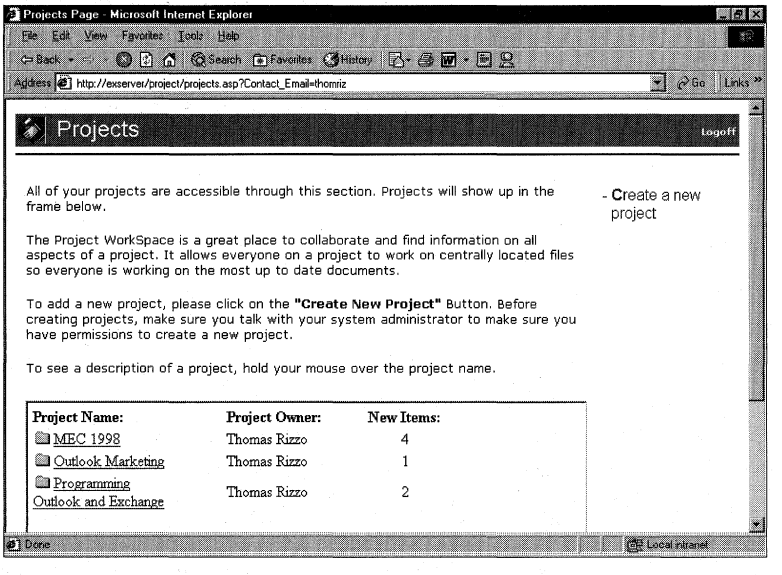


Figure 16-3. *The Projects page, also known as the project workspace page, shows all projects for which the current user has permissions. Users who have proper permissions can create new projects.*

When a user clicks on the hyperlink for a project for which they have permissions, the chosen project page appears. It contains folders for the current project, as shown in Figure 16-5. The options that appear are determined by the permissions set for the current user of the project. For example, if the user is the project owner,

the user can modify the project permissions for the folder. Figure 16-6 shows the Assign Permissions page, where project owners can set permissions.

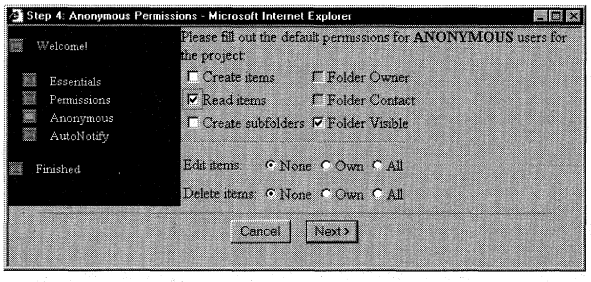


Figure 16-4. *The Create New Project Wizard. On this page of the wizard, the project owner can set permissions for anonymous users who can browse through the project.*

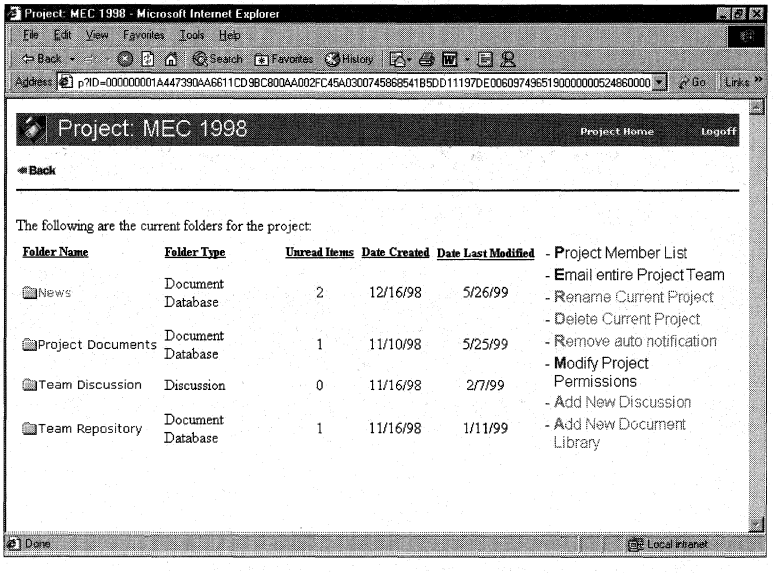


Figure 16-5. *A selected project page. From this page, users can view the folders contained in a project.*

Users who want to view project members for the current project can select the Project Member List link from the Project page. The project member list, shown in Figure 16-7, is created by querying the project permissions using the ACL object.

When a user clicks on a folder to view the project content, a Web interface that allows the user to see folder documents and discussions appears. By default, the folder contains a number of views from which the user can select. Figure 16-8 shows the threaded discussion view for a folder.

The project folders support autonotification. Autonotification is implemented by creating a rule in the Projects public folder that automatically forwards any new

items to project members. This part of the application uses the Rules object, which we discussed earlier.

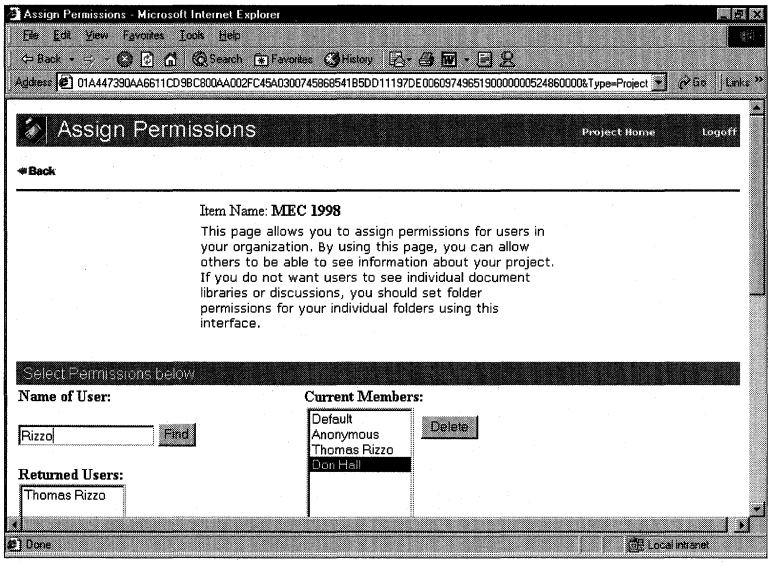


Figure 16-6. *The Assign Permissions page for a project. From this page, owners can assign permissions for users.*

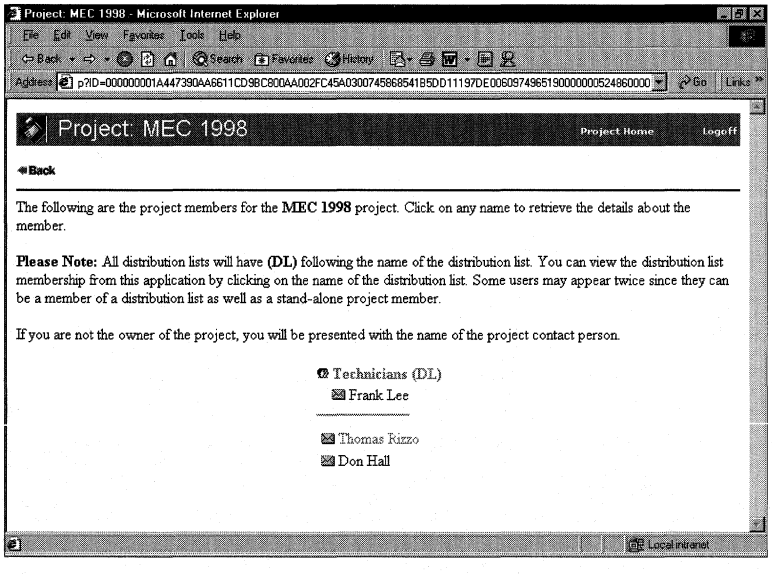


Figure 16-7. *A project member list. If the user is not an owner of the project folder, the user will see only the main contacts for the project and not the actual project member list.*

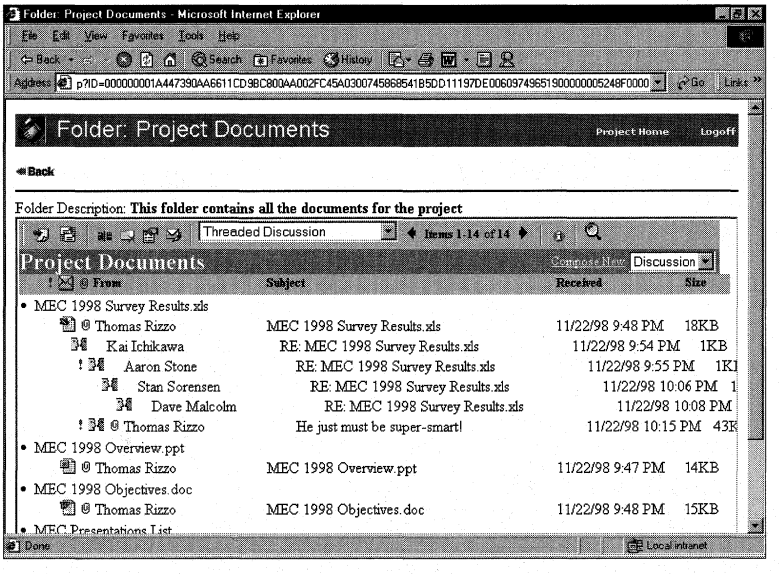


Figure 16-8. *Viewing the information in a project folder using the threaded discussion view.*

To post a new item or reply to an item in the folder, the user employs the composition form. In this form, users type information they want to post. The post form includes an option to upload file attachments to a Web server by using the Posting Acceptor. There's one caveat when using the Posting Acceptor: a bug in the component destroys a user's session variables the first time the user successfully uploads a file. To provide a workaround for this bug, the application tells the user that when she uploads a file for the first time, she will need to log on to the application again to reestablish the ASP session variables. Figure 16-9 shows the Reply To Group form.

Note that the application can be expanded to include support for searching all items in the folder or in the project. I've set up the application so that it can be integrated with Microsoft Site Server. Site Server provides full-text indexing and retrieval of both items in a Public Folder and the attachments in those items. Furthermore, Site Server automatically indexes custom properties on your messages so that you can create search pages that search by these custom properties.

In Chapter 17, we'll examine Site Server in detail. I've also included on the companion CD an example that demonstrates how to use Site Server and Exchange Server together to implement full-text indexing and search capabilities. (Figure 16-10 shows a custom search page to be used with Site Server.) Using this information, you should have a great start in creating custom search applications that use the collaborative technology of Exchange Server.

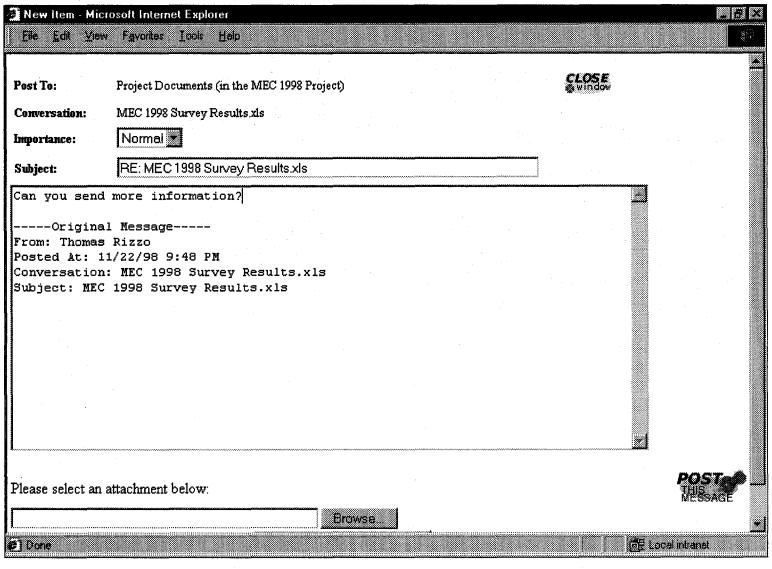


Figure 16-9. *The Reply To Group form allows users to post their responses in a discussion format as well as upload files to the server as attachments.*

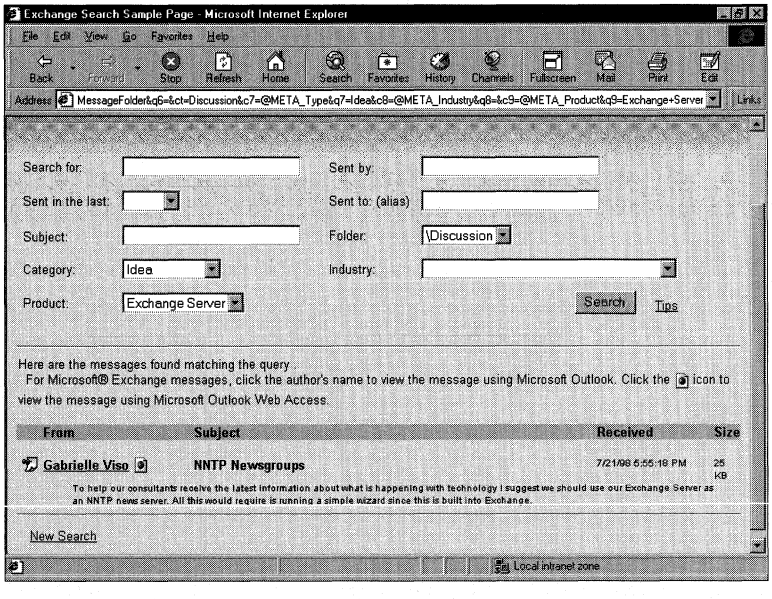


Figure 16-10. *A custom search page built using Site Server. This search page, included on the CD, allows you to search across Public Folders using built-in and custom properties on items.*

Implementing the Projects Application

Most of the Project application is CDO-based, so in the book I won't be covering the parts that deal with CDO. But take a look at the source code for the application on the companion CD because it illustrates well how to use different CDO components—it customizes the Rendering objects, uses CDO properties, and uses many core CDO messaging objects. I'll focus my discussion here on the implementation of the new COM components discussed in this chapter, ACL and Rules.

The best example of using the ACL component in the Project application is in the file permlist.asp. The permlist.asp file is used to update the permissions for a user on a folder based on the options selected by the project owner. Other files also use parts of the ACL component. For example, when a user first accesses the Web page where all the projects are listed, the ACL component retrieves the project owner from the list. The following code, from the permlist.asp file, highlights how to use the ACL component:

```
'Retrieve the ID for the current address entry
strAEID = Request.QueryString("AEID")

If Request.QueryString("Command") = "Update" then
   'Check to see if user or built-in account.
   'If not built-in, do the CompareIDs.
   boolSameID = False
   if Not(strAEID = "ID_ACL_ANONYMOUS" or _
   strAEID = "ID_ACL_DEFAULT") then
      'Check to make sure users are not trying to update
      'permissions for themselves.
      'If they are, do nothing.
      'If you want to allow this, remove the comparison code.
      boolSameID = oSession.CompareIDs(strAEID, _
         oSession.CurrentUser.ID)
   end if
   if boolSameID = False then
      set aclobject = server.createobject("MSExchange.aclobject")
      aclobject.cdoitem = oCurrentFolder
      set fldr_aces = aclobject.aces
      'For users, convert the ID to a string for it to
      'work correctly
      set ACE = fldr_aces.item(cstr(strAEID))

      'More of this code is discussed later in the chapter...
   End If
End If
```

The first task the code performs is to retrieve the AddressEntry ID of the user, whose ACL on the folder will be updated. This ID is used later to obtain the access

control entry (ACE) for the user. Next, the code checks to see whether the update command is sent along with the query string. If it is, the code knows that it must update the ACL for the user of the folder.

Next, the code checks the AddressEntry ID to make sure it's not a built-in account such as Default or Anonymous. Checking the AddressEntry ID enables the code to stop any errors that would occur later when the *CompareIDs* method is called on the CDO Session object.

> **NOTE** Every folder has two built-in accounts, Default and Anonymous. Default is used to set permissions for any user accessing the folder who doesn't already have explicit permissions. Anonymous is used when users are accessing the folder anonymously through Outlook Web Access (OWA) or other anonymous CDO applications. You can set permissions on these two built-in accounts programmatically using the ACL component.

In the Project application, I added code to prevent the owner of a folder from changing his own permissions on the folder, ensuring that the creator of the folder remains the owner. If the owner wants to change his own permissions, he can change them through Outlook, or he can remove the code that checks whether the passed-in ID and the ID for the current user of the application are the same.

After the code checks the AddressEntry ID, the code creates an instance of the ACL object by calling the *Server.CreateObject* method in ASP with the ProgID MSExchange.ACLObject. Once the object is created, the code sets the *CDOItem* property of the ACLObject to be the current folder for which we want to modify permissions. The *CDOItem* property must be set before you try to retrieve the collection of user permissions on the folder.

The code then sets an object to the ACEs collection using the *ACEs* method on the ACLObject. The ACEs collection contains all permissions on the folder for all users. You can scroll through this collection to see who has permissions on the folder. The file members.asp allows you to do this.

The ACEs collection contains some methods and properties you will want to use in your code, such as the *Item* method, the *Count* property, the *Add* method, and the *Delete* method. In the preceding code, the *Item* method retrieves the specific ACE object for the user, whose ID is passed along the query string.

> **NOTE** The AddressEntry and Recipient objects in CDO return short-term EntryIDs, which are valid only for the current CDO session, whereas the ACL component uses long-term EntryIDs. You can pass a short-term EntryID to the component, and the ACL component will be able to find the correct ACE object for the user. However, when you ask for the *ID* property on the ACE object, the code will return the long-term EntryID for the user. Watch out for this functionality, especially if you store the short-term EntryID in a variable and then try to compare it to the long-term EntryID in your code.

The following code snippet, taken from the selectuser.asp file, shows you how to add a new user to the ACEs collection. The code creates a new ACE object and sets the object's properties, such as the desired rights and the ID of the user for whom the rights should apply. (Note that the *ID* property can also take the constants ID_ACL_ANONYMOUS and ID_ACL_DEFAULT to set the rights for the Anonymous and Default built-in accounts.) After the ACE object's properties are set, the code uses the *Add* method on the ACEs collection to add the new object to the collection. When it's done, it will call the *Update* method on the ACLObject.

```
elseif strCommand = "Add" then
   on error resume next
   'Retrieve the ID for the new user.
   'Remember address entry ID (AEID) is a short-term EntryID.
   strAEID = Request.QueryString("AEID")
   'Get the user name
   strText = Request.QueryString("Text")
   'Create the user in the ACE collection and don't give permissions
   set acls = server.createobject("MSExchange.aclobject")
   acls.cdoitem = oCurrentFolder
   set fldr_aces = acls.aces
   'Create a new ACE and add member
   set newace = createobject("MSExchange.ACE")
   newace.ID = strAEID
   'Since people can see only folder contacts, create all new users
   'with the Folder contact permission. If you don't want this,
   'use the line that gives no permissions.
   newace.Rights = &H0600 'Folder contact
   'newace.rights = &H0400   ' no permissions
   fldr_aces.add newace
   acls.Update
   'Update strAEID with the long-term EntryID
   strAEID = newace.ID
   'Now write some client script to add to list box
   Response.write _
      "<SCRIPT DEFER FOR=window EVENT=OnLoad LANGUAGE=vbscript>" & _
      chr(10)
   Response.write "<!--" & chr(10)
   Response.write "set memberframe = _
      window.Parent.frames.item(""members"")" & chr(10)
   Response.write _
      "set oMemberList = memberframe.document.all.Members" & _
      chr(10)
   Response.write "set newelement = _
      document.createElement(""OPTION"")" & chr(10)
   Response.write "newelement.value = """ & strAEID & """" & chr(10)
   Response.write "newelement.text = """ & strText & """" & chr(10)
```

(continued)

```
   Response.write "oMemberList.Add(newelement)" & chr(10)
   Response.write "//-->" & chr(10)
   Response.write "</SCRIPT>"
end if
```

You use the *Delete* method in much the same way you use the *Update* method. The following code, taken from the members.asp file, shows you how to use the *Delete* method. To delete a user's folder permissions, you pass the ID of the user for whom you want to delete permissions from the folder. Then you call the *Update* method on the ACLObject.

> **NOTE** If you granted the Default or Anonymous built-in accounts permission to access a particular folder, when you delete permissions on that folder, the user or distribution list might still be able to access it and its items.

```
if Request.QueryString("Command") = "Delete" then
   strAEID = Request.QueryString("AEID")
   'Check that user is not trying to delete herself.
   'If she is, do nothing.
   if not(oSession.CompareIDs(strAEID, oSession.CurrentUser.ID)) then
      set aclobject = server.createobject("MSExchange.aclobject")
      aclobject.cdoitem = oCurrentFolder
      set fldr_aces = aclobject.aces
      'Must convert the ID to a string for it to work correctly
      fldr_aces.delete cstr(strAEID)
      aclobject.Update
      set aclobject = Nothing
   end if
end if
```

The last code snippet we'll examine, which follows, is taken from the permlist.asp file. It shows you how to use the ACE object and its properties. You actually set user permissions on the ACE object by using different Boolean properties, such as *CreateItems* and *FolderVisible*, and then committing the changes to the Exchange server by calling the Update command on the ACLObject. The one property that the next example does not show how to use is the *Rights* property, which returns a bitmask containing the user's rights on the folder. The include file, named acl.inc, contains the specific values for this bitmask.

```
'Scroll through the form and update the ACL
   if Request.Form("CreateItems") = "on" then
      ACE.CreateItems = True
   else
      ACE.CreateItems = False
   end if
   if Request.Form("ReadItems") = "on" then
      ACE.ReadItems = True
```

```
    else
       ACE.ReadItems = False
    end if
    if Request.Form("CreateSubFolders") = "on" then
       ACE.CreateSubFolders = True
    else
       ACE.CreateSubFolders = False
    end if
    if Request.Form("FolderOwner") = "on" then
       ACE.FolderOwner = True
    else
       ACE.FolderOwner = False
    end if
    if Request.Form("FolderVisible") = "on" then
       ACE.FolderVisible = True
    else
       ACE.FolderVisible = False
    end if
    if Request.Form("FolderContact") = "on" then
       ACE.FolderContact = True
    else
       ACE.FolderContact = False
    end if

    Select Case Request.Form("EditItems")
       Case "EditNone"
          ACE.EditOwn = False
          ACE.EditAll = False
       Case "EditOwn"
          ACE.EditOwn = True
       Case "EditAll"
          ACE.EditAll = True
       End Select

    Select Case Request.Form("DeleteItems")
       Case "DelItemsNone"
          ACE.DeleteOwn = False
          ACE.DeleteAll = False
       Case "DelItemsOwn"
          ACE.DeleteOwn = True
       Case "DelItemsAll"
          ACE.DeleteAll = True
    End Select

    'Update the object
    aclobject.Update
    set aclobject = Nothing
end if
```

Using the Rules Component to Fire on All Incoming Messages

When we examined the Rules component earlier, we did not review one aspect of it—the ability to fire rules on all incoming messages, not just messages that meet certain criteria. The sample application uses this capability to autoforward any new items posted to the folders in the project to all the team members. The following code snippet, taken from ProjectMain.asp, shows you how to create a rule that fires on every incoming message:

```
elseif Request.QueryString("Command") = "AddNotify" then
   set oProjectFolder = Session("ProjectFolder")
   strProjectFolderID = oProjectFolder.ID
   'On error resume next

   'Need to scroll through each subfolder in the current project
   set oFolders = oProjectFolder.Folders

   for each oFolder in oFolders
      'Create Rules object
      set oRules = Server.CreateObject("MSExchange.Rules")
      oRules.Folder = oFolder
      'Check to make sure the rule doesn't already exist
      Dim intFoundit
      intFoundit = 0
      'Just in case user is not the owner of the folder,
      'on error resume next
      for each oRule in oRules
         if oRule.Name = "AutoNotify" then
            intFoundit = 1
         end if
      next

      if intFoundit = 0 then
         'Create the new rule
         IDCount = 0
         'We can always assume there is at least one person in the
         'ACLs for the folder who is the owner.
         'If there is no one in the ACLs, a user probably
         'wouldn't even get to this Web page.
         Redim strIDs(0)
         set acls = server.createobject("MSExchange.aclobject")
         acls.cdoitem = oFolder
         set fldr_aces = acls.aces
         if fldr_aces.count > 0 then
            'On error resume next
            for each fldr_ace in fldr_aces
               strID = fldr_ace.ID
```

```
            if not(strID = "ID_ACL_ANONYMOUS" or _
            strID = (ID_ACL_DEFAULT") then
            'Get the AddressEntry object
               set oAE = oSession.GetAddressEntry(strID)
               if Not(oAE is Nothing) then
                  'Redim the array
                  Redim Preserve strIDs(IDCount)
                  strIDs(IDCount) = cstr(oAE.ID)
                  'Bump the count of the array
                  IDCount = IDCount + 1
               end if
            end if
         next
      end if

      'Create new Action for Rule object
      set oAction = Server.CreateObject("MSExchange.Action")
      oAction.ActionType = 6    'Forward
      oAction.Arg = strIDs

      Set oExists = CreateObject("MSExchange.ExistsCondition")
      oExists.PropertyTag = ActMsgPR_MESSAGE_CLASS

      'Create a new Rule object
      set oNewRule = Server.CreateObject("MSExchange.Rule")
      oNewRule.Name = "AutoNotify"
      oNewRule.Actions.Add 1,oAction
      oNewRule.Condition = oExists

      'Add the new Rule object to the Rules collection
      oRules.Add , oNewRule
      oRules.Update
      Set oRules = Nothing
   end if
next
```

This code creates a new instance of the Rules component, and then scrolls through the collection of rules for the component to see whether the AutoNotify rule already exists. If the rule does exist, the code does not perform any action.

If the AutoNotify rule does not exist, the code uses the ACL component to retrieve all the users who have permissions on the folder. By using the *ID* property on the ACE object, which we discussed earlier, the code creates an array of EntryIDs for all the users. This array is used later as an argument to a method in the Rules object.

Once the array of EntryIDs is initialized, the code creates an Action object for the new rule we're creating. The action for the rule is to forward all new items to a group of users. To specify the users, the *Arg* property on the Action object is passed the array of EntryIDs that we created.

Once the Action object is created and initialized, the code creates an ExistsCondition object, which is used in the Rules object to determine whether a particular property exists on the incoming item. (Note that the component does not check whether the value in the property is valid, only that it exists.) To specify which property to look for, you must set the *PropertyTag* property on the ExistsCondition object. The code sets the *PropertyTag* property to be the message class of the item, because this property is always guaranteed to exist on an item. Then the code uses the methods we discussed earlier to create the actual rule and add it to the Rules collection on the component.

NOTE This chapter gave you a brief introduction to the power of the three new COM components available for Exchange Server in the Microsoft Platform SDK. These components help round out the already extensive development environment of Exchange Server. For more information on any of these components as well as the other technologies included with Exchange Server, refer to the Exchserv.chm file on the companion CD, as well as the Platform SDK section of the MSDN Library.

Chapter 17

Search Solutions Using Site Server 3

Microsoft Exchange Server allows you to store a multitude of document types, but while it's a great mechanism for storing information, it doesn't allow you to easily search for this information. And your Exchange Server public folder hierarchy can quickly get hairy. Faced with these two limitations, your users will undoubtedly have a hard time finding the information they need.

This information retrieval problem isn't only a problem for Exchange Server. In your organization, you probably have a broad scale of information repositories such as file shares, databases, and Web sites. Since different types of data exist in these locations, it's hard to normalize search parameters. Plus, you might even have multilingual documents stored in a single location, so you might not find the information simply because it is in a different language or a different format and your search doesn't find a match.

ENTER SITE SERVER

Have no fear—Microsoft Site Server can help you solve this search dilemma. Site Server not only provides full-text indexing of the most common document types, including Microsoft Office documents, but it also supports a rich object library that you can use to build some powerful search applications for Exchange Server and Microsoft Outlook.

In this chapter, we'll quickly look at the requirements for Site Server. We'll also look at the object model included with Site Server that makes building search applications easy. This object model extends the Microsoft ActiveX Data Objects (ADO) object model, so if you're already familiar with ADO, you've got a head start on creating Site Server solutions.

WHAT ABOUT INDEX SERVER?

There are some key differences between Microsoft Index Server and Site Server. Index Server is a good solution only if you're searching file-based content and don't need to search multiple data types, such as Exchange Server data. In contrast, Site Server can easily search such multiple data types. Furthermore, Site Server provides a better enterprise solution than Index Server.

Whereas both these products use the same search engine, Site Server has many features that Index Server doesn't have. These include a distributed, multithreaded crawler that can gather content from multiple data sources including file, Web (intranet and Internet), database, and Exchange Server data sources; and a configurable schema that you'll learn to use in your programming. (See the "Site Server Search Object Model" section later in this chapter.) Finally, you can disperse your Site Server catalog to multiple machines so that users can search it much more quickly.

SEARCH CAPABILITIES OF SITE SERVER

Site Server supports full-text and property searching of multiple data sources. These data sources include http, file, Exchange Server, and ODBC-compliant databases. Site Server implements searches using a flexible and powerful distributed process called a crawler. Once the crawler has indexed the data sources you specify, the newly created catalogs are compiled and propagated to the search servers you specify. Since you can have multiple search servers, you can gain the best performance by load balancing users' queries across these search servers.

At the Microsoft Windows NT level, Site Server runs two services: Gatherer and Search. The Gatherer service crawls the content, extracts the information, compiles the catalog, and then propagates the catalog to the required hosts. The Search service allows you to search the catalog for data.

When crawling content, the Gatherer service extracts the full-text content, properties, and links. One advantage of the Gatherer service is that it contains built-in filters for Office documents. This means the Gatherer service can open Office documents and pull out custom properties as well as built-in Office document properties such

as Author or Last Save Time. The Gatherer service retrieves these properties by using a plug-in filter. Certain filters, such as Office document filters, come with Site Server. However, because the plug-ins implement the *IFilter* interface documented in the Platform Software Development Kit (SDK), you can create your own filters for the specific types of documents you use. The Adobe PDF filter, available from Adobe's Web site, is an example of such a custom filter. This filter allows Site Server to index PDF files stored in data sources that Site Server can crawl.

Once you've finished indexing the content, the real fun begins. Now you're ready to search the content. Site Server is a powerful search product. Its Search service supports wildcard, free-text, and regular expression searches. You might be thinking to yourself that no user would want to use a regular expression search. But don't think of Site Server's features from the perspective of a user performing a search. Instead, think of the applications you can build that leverage these powerful features.

The Search service is also multilingual. Site Server determines the language of the document it's crawling. The Search service then uses the correct word-breaking module, which identifies specific words in a document, and the correct word-stemming module, which identifies grammatically correct variations of each word. (This is an example of word stemming: fly, flying, flown, and flew.) Site Server will also ignore noise words such as "the," "a," or "do"—that is, words that are unlikely to carry searchable information. You can configure and add to the noise word lists that Site Server provides by using a text editor.

If you do scan multilingual documents, you can have Site Server search in multiple languages or you can specify a single language for the search. In your search applications, you can even detect the language of the person attempting the search and automatically default to that language when querying the Site Server catalog.

All the capabilities we've just discussed are available in the object model that Site Server provides for searching. To help you get more acquainted with some of the finer capabilities of Site Server Search, I included the Site Server help files on the companion CD. You'll find a lot of useful information about regular expression, free-text, and word-stemming searches. I highly recommend that you look at the documentation and give some of these types of searches a try.

INFRASTRUCTURE REQUIREMENTS FOR SITE SERVER

Let's look at the Exchange Server–specific features and requirements of using Site Server and Exchange Server together. We'll also examine how to configure a host, how to set up a crawl, and how to begin searching Exchange Server information. Plus, we'll discuss the security implications of using Site Server and Exchange Server together.

Exchange Server Requirements for Site Server

When using Exchange Server and Site Server together, you need to run the two products on different computers. When setting up Site Server, tell the product which machine running Exchange Server it should point to in order to crawl for content. The machine that Site Server is pointing to should have all the public folders either homed or replicated to the Exchange server or to another Exchange server on the same site.

You can use Site Server to index and search the public folders homed on other sites. Then you need only set up public folder affinity correctly to achieve the best performance and set the default timeout period to crawl the content accordingly. You don't want Site Server to time out while trying to obtain Exchange Server content.

NOTE Site Server can index only Exchange Server public folders, not private folders.

Figure 17-1 shows how to set the Exchange Server information in the Microsoft Management Console (MMC) administration for Site Server Search. Notice that you must also set the Exchange Server organization and site.

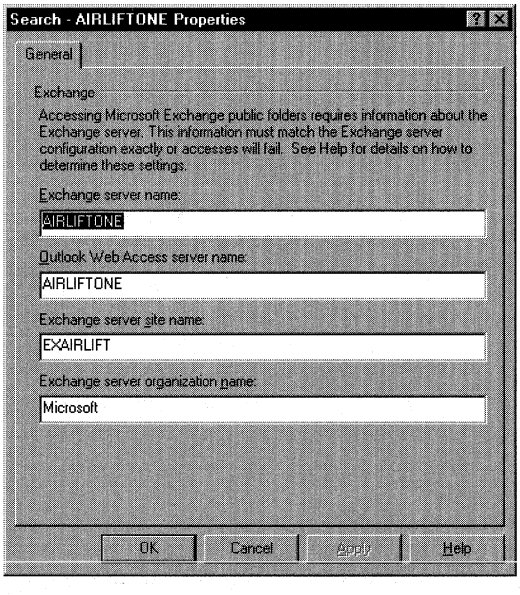


Figure 17-1. *Setting the properties of the Exchange server that Site Server will talk to.*

Once you set the properties for the Exchange server you want Site Server to use for public folder access, you can specify the Outlook Web Access (OWA) server used to read information returned from Exchange Server searches. This specification

is optional because you can also use Outlook to view information returned by the Search service. Site Server includes an ActiveX control that detects whether the user has Outlook installed and launches the appropriate method of reading the search results. As a developer, you can force this ActiveX control to default to Outlook Web Access or Outlook depending on your needs. You'll see how to do this later in this chapter.

Setting Up Your Search Hosts

Before attempting to crawl a public folder, you need to configure some accounts and the security on all your systems, including Exchange Server, Site Server, and Windows NT. Different accounts, such as administrative access accounts, exist for propagating completed catalogs and content access accounts. We'll look at content access accounts only, because they cause problems for many developers and IT professionals who configure Site Server and Exchange Server.

Default Content Access Account

When Site Server crawls an Exchange server, the Search service either uses the default content access account specified on your search server or it uses a site-specific account. I don't recommend using site-specific access accounts for Exchange Server unless it's completely necessary—for example, when Exchange Server sites span untrusted Windows NT domains. It's better to use a single content access account—so that you make administration easier—and to track the use of this account when crawling content.

Your content access account must be configured with a domain account that has administrative rights on the Configuration object for each site that hosts the public folders you plan to crawl. This account must also have administrator privileges on the computer running Site Server and must be a member of an administrative group such as Site Server Administrators. For more information on this topic, see the Site Server documentation.

Search Service Account

Besides the default content access account, you need to configure the Windows NT account used by the Search service. This account must meet the same criteria as the default content access account mentioned earlier.

> **WARNING** By default, the Search service will run as the System account. Since this account has no permissions on the Exchange Server public folders, your crawling and searches will fail on Exchange Server properties. Watch out for this! To set the Windows NT account for the Search service, use the Service dialog box shown in Figure 17-2.

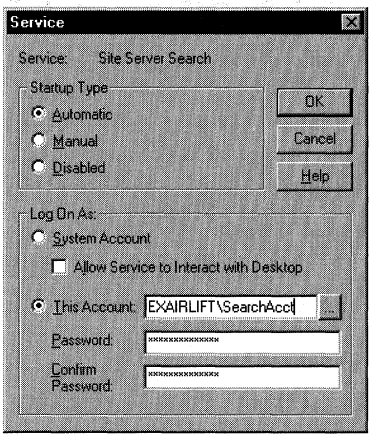


Figure 17-2. *Configuring the Windows NT account used by the Search service.*

Timeout Periods

You need to increase the default timeout for crawling in the Site Server administration program. By default, the timeout is 10 seconds. For Exchange Server data, change this to at least 60 seconds. If you're going to crawl replicated public folders in other sites, you'll need a longer timeout to allow Site Server to crawl those folders.

Setting Site Server to Crawl an Exchange Server Public Folder

Once you've completed the infrastructure work, setting up Site Server to start crawling Exchange Server information is a snap. Site Server provides two administration modes: MMC mode and Web-based mode. I'll show you the technique using the MMC administration, but you can also perform these steps using the Web-based one.

The first step is to create a new catalog build definition, which is where you specify the starting addresses of the items you want to crawl in your catalog. Site Server supports multiple start addresses for multiple document types, meaning that in a single catalog you can have Exchange Server, file, and http information and you can use a single query to search all this data. Figure 17-3 and Figure 17-4 show how to use the New Catalog Definition Wizard to set up a crawl of Exchange Server information. You can rerun these steps to add other types of crawls to your catalog.

After you're finished with the wizard, Site Server can start building the catalog for your data sources. You can then set up a build schedule so that Site Server can recrawl the information in your data sources at the times you specify. Note that Site Server supports incremental crawls so that changes are put into the catalog. Also, be careful if you have any slash (/) or percent (%) characters in your public folder names. If you use such names for start addresses, you'll need to replace the slash with %2F and the percent with %25.

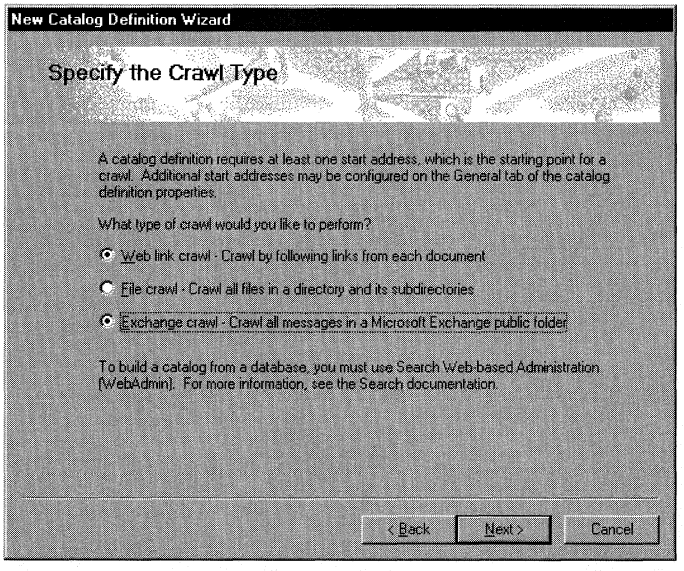


Figure 17-3. *Using the New Catalog Definition Wizard to define the type of information you want to crawl.*

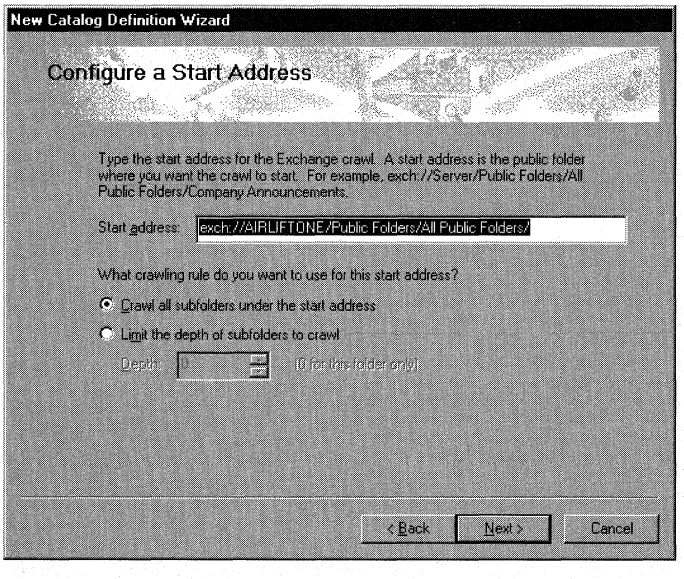


Figure 17-4. *Specifying the start address for your data source.*

Security Implications

There are two security implications of using Site Server and Exchange Server together that you should know about. Be assured that Site Server doesn't compromise an Exchange Server system. Site Server will return only the Exchange Server information that a specific user requests, and only that user will be able to see the search results.

First, Site Server does not store the access control lists (ACLs) of Exchange Server items in the catalog. Instead, it queries the Exchange Server for the permissions at search time. To improve performance, Site Server refreshes a cache of these ACLs only every 8 hours. After you change Exchange Server public folder permissions and then create a full or incremental build, users will retain their old permissions for 8 hours. You can configure the interval Site Server uses to refresh Exchange Server permissions by modifying the registry. Instead of listing the gory details here, I'll point you to Knowledge Base article Q198892.

Second, if you require https access to your Outlook Web Access servers, this can affect the default search pages and your custom-built search pages when using Outlook Web Access to view content. By default, Site Server will return links to Exchange Server content using only http. You'll have to modify the returned links programmatically to change http to https for your applications.

CREATING CUSTOM SEARCH APPLICATIONS

Site Server provides some powerful indexing and search capabilities. However, Site Server does not provide a targeted way for users to search the information contained in its catalogs. Fortunately, Site Server has an object model that you can use to build search applications that allow users to target the information in their searches. We'll take a look at this object model and the different user interfaces from which you can use it to allow users to search Exchange Server information.

Site Server Search Object Model

Site Server includes an object model that makes building your search applications easier. This object model consists of two objects: Query and Util. The ProgID for the Query object is *MSSearch.Query*; for the Util object, it's *MSSearch.Util*. While the Util object library is interesting, only a few of its methods are useful for Exchange Server developers. Therefore, we won't look at the Util object library; instead, we'll focus on the Query object library because this is primarily the one you'll use to build search applications. Figure 17-5 shows the object model for the Query object library.

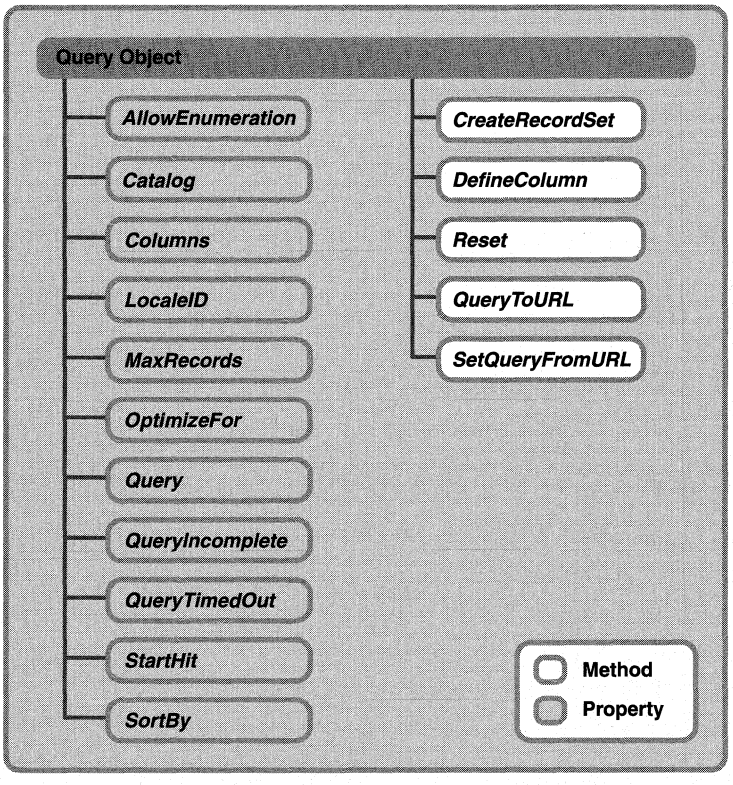


Figure 17-5. *The Query object library with methods and properties.*

Query Object Model

Using the Query object model, you can create powerful search applications. You use the Query object model to create and execute queries against a Site Server catalog. This object model extends the ADO object model, providing a mechanism by which you can count, retrieve, and view the results of your queries.

Let's take a look at the properties of the Query object.

- *Allow Enumeration.* By setting this property to *True*, you allow your query to enumerate parts of the catalog recursively. This means you can perform more complex queries—but at a performance cost. This property defaults to *False* because most searches can be performed without enumeration. Set this property to *True* only in rare instances such as when using multiple wildcards or advanced regular expression searches.

- *Catalog*. This string property specifies the Site Server catalog you want to search. If you have selected a default catalog, you don't have to set this property. You can specify multiple catalogs to search by separating their names with a comma.
- *Columns*. This string property specifies the columns from the search catalog that you want to display in your search results. The names of the columns are case insensitive and should be separated by commas.
- *LocaleID*. This number property specifies the Win32 Locale ID. Based on this number, Site Server uses the word breaker and word stemmer of that language for your search. If you're building Web-based search applications, this property will default to the Locale ID of the browser.
- *MaxRecords*. This number property allows you to limit the number of search results returned per page. By default, this property is *0*, which returns all search results. You should definitely modify this property in your applications if you want to perform your queries faster. For best results, set this property to *25* or less to help your users limit their searches to items that most closely match their criteria.
- *OptimizeFor*. This string property tells Site Server how to optimize its searching of a catalog. You have a few different options for this property that will affect the results displayed for your search. The default values for this property are *recall* and *hitcount*. Let me explain what *recall* means. When you specify *recall*, results are trimmed when the query is executing, which might make performance slower but will result in an accurate hit count. (Result trimming is the removal of items that don't match the criteria or for which the user doesn't have authorization to view.)

  Instead of *recall*, you can specify *performance*, which conducts its security checks after the maximum number of hits is calculated. You might get an inaccurate hit count total because the user might not be able to view some of those hits due to the lack of authorization on the items. You can alternate between *performance* and *recall*, but you cannot use both in this property at the same time.

  The *hitcount* and *nohitcount* values specify whether you want to calculate the total number of matches found. If you specify *hitcount*, performance will be a little slower because the entire catalog must be processed before displaying the first result.

- *Query*. This string property specifies the query you want to use to search the catalog. For example, this property might appear this way:

```
"( @FileWrite < 1999/11/26 AND > 1999/11/
24 ) & ( @DocTitle Outlook ) _
   & ( @MessageClass IPM.Note.* Or IPM.Post* )"
```

  Usually in your application you won't generate the string for the *Query* property directly. Instead, you'll use the conventional query string variables, which we'll discuss later in this chapter.

- *StartHit*. This number property can be a single number or an array of numbers, depending on whether you're searching multiple catalogs in a single query. The *StartHit* property specifies the starting result number for each catalog and is commonly used with paged results. *StartHit* allows you to determine which result to start with if more results are available than displayed in your search application. For example, if you display only 50 results at a time and provide a More Results button in your application, you should set the *StartHit* property to *51*. This property is used in conjunction with the ADO extended property *NextStartHit*, which you'll see later in this chapter.

- *SortBy*. This string property specifies which columns you want to sort by and whether to sort in ascending or descending order. You can sort only by columns that are retrievable by Site Server. The catalog schema specifies whether a column can be retrieved. Site Server supports sorting by up to four columns. You specify each column using its name followed by either [a] or [d] for ascending or descending and use commas to separate multiple columns—for example, Rank[d], Title. Try to sort by rank or another column that applies to the type of items your users are searching; otherwise your sorting might not make sense to users of your application.

- *QueryTimedOut (read-only)*. This property is available only after you use the *CreateRecordset* method. This property returns *True* if the query exceeded the time limit you specified for the catalog and *False* if the query didn't exceed the time limit.

- *QueryIncomplete (read-only).* This property is available only after you use the *CreateRecordset* method. Also, it should be used only if the enumeration property is set to *False*. If enumeration is set to True, *QueryIncomplete* always returns *False*. This property can tell you whether the query was resolved. A query might not be resolved for a number of reasons, such as being too complex or referring to nonexistent columns. If you can't successfully create a recordset for your query results, check this property to see whether the query itself is at fault.

Now let's look at the methods of the Query object:

- *CreateRecordset.* This method takes one argument, *Sequential*, because Site Server Search can create only sequential, forward-only cursors. When you call this method, make sure that all the properties of the Query object are set to your preferences. This method will create an ADO recordset containing the search results from your query. You can then use standard ADO methods and properties to scroll through the recordset. You'll see how to do this later in the chapter in the sample applications. Also, Site Server extends the ADO object model with some extra properties that we'll look at later in this chapter.
- *DefineColumn.* This method defines a new display name for a column in your query. You need to pass this method the data type, property set ID, property ID, and property name as a string to specify the column you're interested in. You can reassign names for built-in or custom properties that you've already defined. Instead of using this method, if you plan to define a new name for a column or a new column, it's better to modify the definecolumns.txt file that's under the Site Server directory. This will make your column changes permanent so that you don't have to define the column with each query. Furthermore, this text file will give you the list of property set IDs and property IDs. (A property ID can either be a unique number or name.) The Site Server property set IDs and property IDs are different from the Exchange Server ones, so be careful not to confuse them. Here's an example of using the *DefineColumn* method:

```
Q.DefineColumn "CustomColumn (DBTYPE_VECTOR | OR DBTYPE_I2) =
    d1b5d3f0-c0b3-11cf-9a92-00a0c908dbf1 CustomColumn"
```

- *Reset.* This method clears the settings in the Query object. If you want to start with a fresh Query object, call this method.

- *QueryToURL.* Call this method when you want to convert the settings of the Query object to a URL-encoded string. This is useful if you want to pass the query from one Web page to another. You'll see the format of this URL-encoded string when we discuss conventional query string variables later in this chapter. The following is an example of this method:

```
strQuery = Q.QueryToURL
```

- *SetQueryFromURL.* Since Site Server can change the settings into a query string, it can also set the Query object from the variables in a URL query string. This is what *SetQueryFromURL* does. This method expects a URL encoded string. Site Server then parses the string, looks at the values, and overwrites any settings already present on the Query object with the settings from the string, ignoring unrecognized settings. Here's an example of using this method from Microsoft Active Server Pages (ASP):

```
<% Q.SetQueryFromURL("ct=MyCatalog&c2=@All&q2=Outlook") %>
```

ADO Recordset Extensions

In addition to the standard methods of the Recordset object, such as *Move*, and the properties of the Recordset object, such as BOF and EOF, Site Server adds some extensions that are useful for search applications. However, Site Server does not support the full range of ADO Recordset methods and properties. Table 17-1 lists which methods and properties are supported and unsupported.

	Supported	***Unsupported***
Methods	*Close*, *GetRows*, *Move* (forward-only), *MoveNext*, *Supports*	*AddNew*, *CancelBatch*, *CancelUpdate*, *Clone*, *Delete*, *MoveFirst*, *MoveLast*, *MovePrevious*, *NextRecordset*, *Open*, *Requery*, *Resync*, *Update*, *UpdateBatch*
Properties	*BOF*, *CacheSize*,*CursorType*, *EOF*, *Filter*, *MaxRecords*, *Source*, *Status*	*ActiveConnection*, *AbsolutePosition*, *AbsolutePage*, *Bookmark*, *EditMode*, *LockType*, *PageCount*, *PageSize*, *RecordCount*

Table 17-1. *ADO Recordset methods and properties supported and unsupported by Site Server.*

Site Server adds five extended properties to the Recordset object, and these are described in the following list. To use these properties, use the following syntax: Recordsetobj.Properties("ExtendedProperty").

- *CatalogSeqNums.* This property should be used only for multiple catalog searches. It holds new entries in the catalogs between searches. It maps to the *CrawlLastModified* property in the Search Gatherer Property Set. If you're searching only a single catalog, use the *CrawlLastModified* property and specify it as one of your query variables. Once you retrieve this property, whose value is a set of numbers separated by semicolons, store the values for each user. Then you can offer a search that displays only items that have been modified since the last catalog search. The following shows an example of searching only new items if *CatalogSeqNums* returned *10;15* for two catalogs. Note that this search uses the standard query syntax rather than the conventional query string variables you'll see later in the chapter.

```
( @All Outlook ) & ( @CatalogSeqNums > 10;15 )
```

- *MoreRows.* Use this Boolean property after you create your recordset. It returns *True* if more search results are available than are currently being displayed and *False* if no more search results are available. You can use this property to determine whether you need enable the More Results button in your search application.
- *NextStartHit.* You can use this method to determine where the display of the next batch of results should start, with respect to the results currently being displayed on the page. You will receive a comma-delimited string of numbers for multiple catalog searches. You should then feed those numbers into the *StartHit* property of the Query object so that you can display successive pages of search results.
- *RowCount.* If you have set the *OptimizeFor* property on the Query object to the *HitCount* value, the *RowCount* property will return the total number of records found in all catalogs searched. Because determining total number of hits might be time consuming, Site Server defaults to returning only 200 hits per search. Don't be alarmed if you know your search has more than 200 hits. You can configure Site Server to return all hits, either by setting the calculated number of results for each catalog in Site

Server administration to a large number or by setting the *MaxRecords* property of the Query object to *0*. For catalogs that contain Exchange Server information, keep the number of results returned fairly low. You should use this property in conjunction with the *RowLimitExceeded* property.

- *RowLimitExceeded*. This Boolean property specifies whether the *RowCount* for the catalog was exceeded. Since you can determine whether there are more results than yielded by the *RowCount* property, you can use the *RowLimitExceeded* property to display results such as "1 – 50 items of more than 200 found". If you specify *nohitcount* for the *OptimizeFor* property on the Query object, *RowLimitExceeded* will always return *False*.

Building an ASP Search Application

The most common way that developers build search pages for Site Server is by using ASP applications. This allows you to provide your search application with a Web interface while taking advantage of the Site Server object model's easy-to-use, Web-based methods. We'll take a look at leveraging public folders for a knowledge base application and providing a rich search of this knowledge base using Site Server.

Before we dive into the code, I'd like to point out a few things about the sample application. This application includes a custom Outlook form for submitting information into the knowledge base and for reading information from the knowledge base. Figure 17-6 shows this form.

To make it easy for Web users to work with the knowledge base application, the form is also converted into an ASP-based form using the Outlook HTML Form Converter. This makes it possible for non-Outlook users to view the knowledge base articles. Figure 17-7 shows this version of the form.

The knowledge base form in Figure 17-6 and Figure 17-7 implements some custom Outlook fields such as Category, which specifies the type of information contained in the knowledge base article—for instance, best practice, idea, issue, or resolution. The form also uses a field called Product, which specifies the product called out in the knowledge base article—for example, Exchange Server, Outlook, or Office. Finally, the form uses a field called Industry, which specifies the industry that the knowledge base article refers to. You'll see how Site Server allows you to search these custom properties with minimal effort. Figure 17-8 shows the ASP solution for this search.

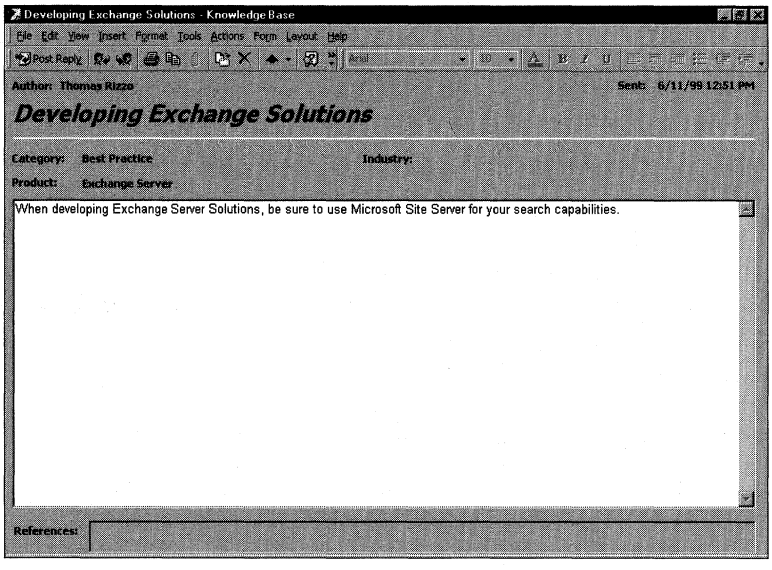


Figure 17-6. *The Outlook form for the knowledge base application.*

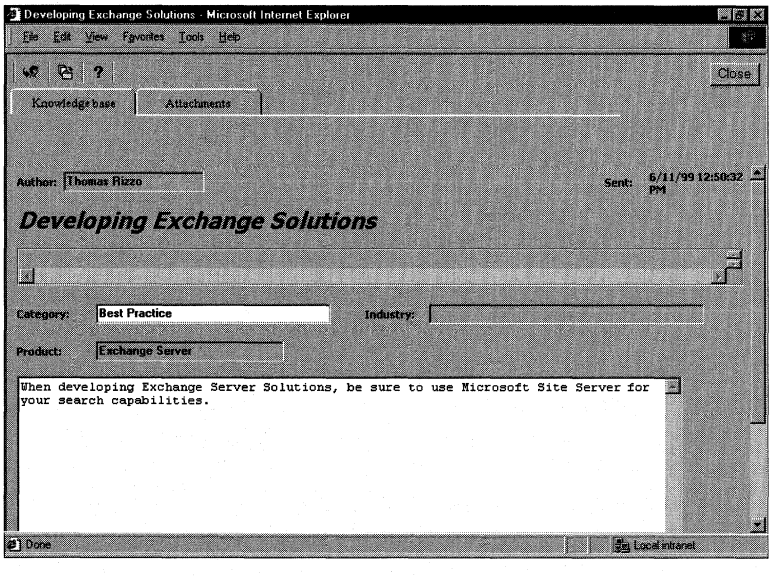


Figure 17-7. *The HTML version of the Outlook knowledge base form.*

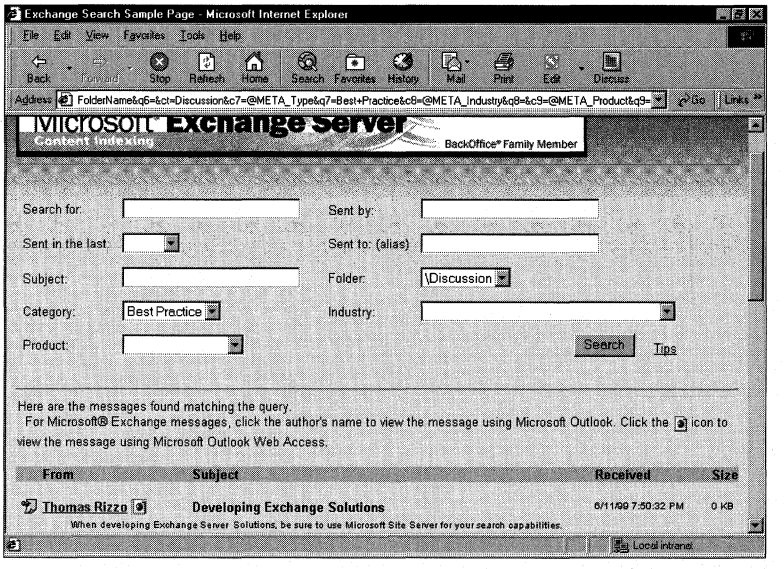


Figure 17-8. *The ASP solution that allows you to search the knowledge base application.*

Conventional Query String Variables

When building custom search applications, you first need to look at Site Server's capabilities for conventional query string variables, which are two-letter tags that you can specify along with a Web address. From these two-letter tags, Site Server can create and populate query variables used in searching catalogs. Even though these variables are targeted toward Web search pages, you can leverage them in other applications, as you'll see momentarily.

The conventional query string variables are designed to identify the property you want to query and the value to use for the query criteria. You can also use these two-letter tags to specify properties on the Query object. For example, the "*ct*" tag identifies the catalog to use for the search. In the "*q*n" tag, q specifies the conventional query string variable and n specifies a number. You should use the "*q*n" tag in conjunction with the "*c*n" tag because the "*c*n" tag specifies the column you want to search for the "*q*n" value. You should use the "*o*n" tag only with numeric and date columns. Here is an example query string from a typical search:

```
q1=&c2=@DocAuthor&q2=&c3=@filewrite&o3=%3E%3D&q3=1y&c5=
@MessageEmailAddress&q5=&o4=@DocTitle&q4=&c6=@MessageFolder&q6=
&ct=Discussion&c7=@META_Type&q7=&c8=@META_Industry&q8=&c9=
@META_Product&q9=
```

Table 17-2 specifies the different values you can have for these query string tags; each corresponds to a Query object property. In addition to tags that correspond to Query object properties, this table also includes query string tags that correspond to multicolumn searches. This is where the "*qn/cn*" tags come into play. If you use the "*qu*" tag to set the *Query* property on the Query object, you should not use the multicolumn search tags as well.

Tag	*Description*
"*ae*"	Allows enumeration. If you pass a nonzero digit, enumeration will be allowed for the query.
"*ct*"	Identifies the catalogs to search. You should pass a string specifying catalogs separated by commas for multiple catalog searches.
"*mh*"	Specifies the maximum number of hits. By setting this to *0*, you specify that all records should be returned.
"*op*"	Indicates what to optimize for. The value *x* specifies performance, *r* specifies *recall*, and an optional *h* specifies no *hitcount*.
"*ou*"	Specifies the full query that you want to perform. For example, you could pass "(@FileWrite < 1999/11/22)" using this tag to search for all items written before 11/22/1999.
"*sb*"	Specifies the starting result number for each catalog.
"*sd*"	Associated with the *SortBy* property on the Query object. Sorts in descending order—for example, sd=rank.
"*so*"	Sorts in ascending order, for example, so=rank.
"*cn*"	Specifies a column that you want in your query. If you leave this blank but specify a corresponding "*qn*" tag, Site Server will assume that you want to query the contents of the full-text index for the catalog. You should precede your column names with an at sign (@) when using this tag—for example, @FileWrite.
"*on*"	Operator that specifies which comparison operator you want to use. You should use this tag for numeric and date queries only. The valid operators for this tag are =, !=, >, >=, <, and <=.
"*qn*"	The query term you want to search for. If you specify a "*qn*" tag without a "*cn*" or an "*on*" tag, Site Server will search the full-text contents for this query term.

Table 13-2. *Tags for conventional query string variables.*

USING OPERATORS WITH DATE VALUES

Site Server provides the ability to perform searches on dates using both relative and absolute values. Relative dates express the current date and time. You express these dates using the minus sign (–) and an integer unit followed by a time unit. The time units can be years (y), months (m), weeks (w), days (d), hours (h), minutes (n), and seconds (s). For relative dates, you can use the operators less than (<), greater than (>), less than or equal to (<=), or greater than or equal to (>=). An example of a relative date query is @FileWrite >= –2y. This query would return any documents that were last saved on a date greater than or equal to the current date/time two years ago. Relative date queries are probably the most common queries that you'll perform.

Site Server also supports absolute date queries. When using this type of query, you must specify a date in one of these formats:

- yyyy/mm/dd hh:nn:ss
- yyyy-mm-dd hh:nn:ss

You might not want to use absolute date queries because of this formatting and because Site Server assumes the date you pass in is relative to the Universal Time Coordinate (UTC), formerly Greenwich Mean Time (GMT). Your users might get confused if they enter a specific date and values they didn't expect to meet that date are returned because of time zone differences.

Note that the equal sign (=) should not be used with absolute dates. Site Server will try to perform an exact match on the date and time, including milliseconds, for your documents. Instead, if you need to target a specific time period, use less than (<) and greater than (>). For example, to show all documents saved on November 25, 1999, you would use the following query syntax:

```
@FileWrite < 1999/11/26 AND > 1999/11/24
```

Exchange Server Property Set

Before you start building a search application, you need to understand which Exchange Server properties Site Server can index. By default, Site Server indexes only five Exchange Server–specific properties and maps a number of its built-in properties to Exchange Server values. Let's take a look at the Exchange Server property set in Site Server and see how to search custom Exchange Server properties.

The five Site Server properties that directly support Exchange Server are *MessageClass*, *MessageDisplayName*, *MessageDisplayCC*, *MessageFolderName*, and *MessageFolderPath*. All these properties are indexed, but the only one that is retrievable as a column, by default, is *MessageFolderName*. Although you can query on the other properties, you can't retrieve them for display in your resultset. Table 17-3 describes each of these properties in more detail.

Name	*Description*
MessageClass	Message class of the item. This can be *IPM.Note*, *IPM.Post*, or any valid Exchange Server message class. This property is useful to query on if you want to limit your search to only messages, posts, documents, or other types of items stored in an Exchange Server public folder.
MessageDisplayName	Display name of the recipient. For posts, you might want to use *MessageFolderName* instead to allow users to search by the folder the item was posted into.
MessageDisplayCC	Display name of any carbon copy recipients.
MessageFolderName	Name of the folder the message was posted into.
MessageFolderPath	Path to the folder that the message was posted into. You might want to specify a wildcard to search for items posted into a particular section of your public folder hierarchy.

Table 17-3. *Built-in Exchange Server properties.*

The built-in Site Server properties shown in Table 17-4 are mapped to Exchange Server values for Exchange Server data sources. These properties are indexed and retrievable by Site Server.

Name	*Description*
DocTitle	Subject of the message.
DocAddress	URL of the message. This property leverages the Outlook Web Access server name you put into the Site Server administration program.
Description	First 300 characters of the message. You can use this property to provide an autopreview feature for your search applications.

Table 17-4. *Built-in Site Server properties.*

Free Documents in Public Folders

Besides supporting built-in Exchange Server properties, you can have Site Server index Office document properties of the free documents in public folders. These properties include the author and category properties. Note that in order to provide Office document support for documents stored in Exchange Server, you need to have Site Server Service Pack 3 installed on a computer running Site Server.

Exchange Server and Outlook Custom Properties

Site Server also supports automatic searching on Exchange Server or Outlook custom properties. You don't have to specify custom properties to search on them; Site Server will automatically provide this capability to your search application. Site Server uses the HTML META property set for searching on these properties. To use the META property set, you must preface the name of your custom property with a @META_propertyname. For example, if your custom property in Exchange Server is named *Username*, to search on that property for a value of *Tom*, you would specify in your query @META_Username Tom. If your Exchange Server custom property contains spaces, wrap your META tag in quotes, as in "@META_User name".

By default, Site Server indexes string and date properties but not numbers. Furthermore, these properties are not retrievable. This means you can search using custom properties, but you cannot display the values contained in those properties as part of the results returned in the ADO recordset. Your best bet when using Site Server and Exchange Server together is to try to use only string properties wherever possible because you'll have to modify the definecolumns.txt file to search date properties.

When using custom properties, you can also tag HTML documents with the same display name for your custom properties using the HTML META property format. This allows you to use the META property search capabilities across HTML and Exchange Server documents in your Site Server searches. You'll see how to use the capabilities of Site Server and Exchange Server custom properties later in this chapter.

> **NOTE** You can find a list of all Site Server columns in the Site Server help file included on the companion CD.

Security Requirements for the ASP Search Application

When setting up your virtual directory to allow searches against Exchange Server information, you must authenticate the user before Site Server can return information to the browser. This authentication is required so that the search doesn't return information that a particular user shouldn't see. Because of this authentication requirement, you should disable anonymous access to the virtual directory where you place your search pages and require Basic or Windows NT Challenge/Response authentication, or you should add code to your search page to authenticate the user directly in Microsoft Visual Basic Scripting Edition (VBScript) or JavaScript code.

If you don't authenticate the user in your code or through the directory security in Microsoft Internet Information Services (IIS), such a user will be able to search and view any Exchange Server information contained in the Site Server catalog that anonymous users are allowed to view. The only problem you might run into would be when search results are returned to the user and that user tries to browse those search results using Outlook Web Access. If you haven't added the particular public folder where that search result resides into the anonymous folders in the Exchange Server administration program, the user won't be able to see the results.

Getting Input from the User

The first section of search.asp processes existing input from the user. Since the application divides the search results into groups of 10 hits, it uses session variables in ASP to remember the criteria the user entered for his search across multiple search result pages. If you customize this application to search on additional properties, be sure to modify this section so that when the user clicks to see more search results, the search criteria are not lost at the top of the HTML page. The code that follows gathers the input from the user and sets the ASP session variables appropriately if the page being displayed is a More Results page of a query:

```
<% 'Process Input Section %>
<% if Request.QueryString <> "" then %>
<% 'Initialize or set nonpredefined variables
   'from information posted to page
   'Purpose of DisplayText variable is to have only the words from
   'the query for display in the introductory sentence and in the More
   'Results link
      if Request("DisplayText")= "" then
         DisplayText= Request("q1")
      else
         DisplayText = Request("DisplayText")
      end if

      'Purpose of RecordNum variable is to display which records
      'are displayed on each results page
      if Request("RecordNum") = "" then
         RecordNum=1
      else
         RecordNum=Request("RecordNum")
      end if

      'Purpose of session variables is to store information for use in
      'editing a search. Only information from the initial query is stored:
      'these variables are not updated on More Results pages.
      'If you add search criteria in your search page, make sure
      'to add their initial values to the Session object.
      'Server.HTMLEncode is required to store queries with double quotes.
      if Request("sh") = "" then
```

```
            Session("q1")=Server.HTMLEncode(Request("q1"))
            Session("q2")=Request("q2")
            Session("ct")=Request("ct")
         Session("c3")=Request("c3")
         Session("o3")=Request("o3")
      Session("q3")=Request("q3")
         Session("o4")=Request("o4")
      Session("q4")=Request("q4")
         Session("c5")=Request("c5")
      Session("q5")=Request("q5")
         Session("c6")=Request("c6")
      Session("q6")=Request("q6")
      Session("c7")=Request("c7")
         Session("q7")=Request("q7")
      Session("c8")=Request("c8")
         Session("q8")=Request("q8")
      Session("c9")=Request("c9")
         Session("q9")=Request("q9")
         end if
         'Read session object into variables
              q1=Session("q1")
              q2=Session("q2")
              ct=Session("ct")
      c3=Session("c3")
         o3=Session("o3")
      q3=Session("q3")
         o4=Session("o4")
      q4=Session("q4")
         c5=Session("c5")
      q5=Session("q5")
         c6=Session("c6")
      q6=Session("q6")
         c7=Session("c7")
      q7=Session("q7")
         c8=Session("c8")
      q8=Session("q8")
         c9=Session("c9")
      q9=Session("q9")

%>
<% end if %>
```

The second section of the search.asp file is the search initiation section. This section displays the form in which the user can specify the criteria for the search. Notice that the inputs on the form are assigned specific names. This allows the ASP page to leverage the conventional query string variables. Be aware that Site Server supports only nine columns for the conventional query string variables. So don't try adding a c10/q10 combination to the ASP search page. It won't work.

You can see the use of the HTML META property set in this next section of code. The search.asp file allows users to search using three custom properties from the knowledge base application. You can also see the use of the Exchange Server property set.

```
<% 'Search Initiation Section
   'Show the search form. If the search is being edited, read
   'the information that has been previously stored in the
   'Session object.
%>
   <p>

   <table cellpadding=5 border=0>
      <form action="Search.asp" method="get">
      <tr>
         <td nowrap><font size=2>
         <% L_QueryLabel_text = "Search for:" %>
         <% = L_QueryLabel_Text %> 
         </font>
      </td>
      <td>
      <input type="hidden" name="c1" value="@All">
         <input type="text" name="q1" size="25" maxlength="100"
            value="<% = q1 %>">
      </td>
      <td width=5></td>
      <td>
         <%
         'Author
         L_AuthorLabel_text = "Sent by"
         %>
         <font size=2><% = L_AuthorLabel_text %>:</font>
      </td>

      <td>
         <input type="hidden" name="c2" value="@DocAuthor">
         <input type="text" maxlength=100 size=25 name="q2"
            value="<% = q2 %>">
      </td></tr>

      <tr>
      <td>
      <font size=2>
      Sent in the last:
      </font>
      </td>
      <td>
      <input type=hidden name="c3" value="@filewrite">
      <input type=hidden name="o3" value=">=">
```

```
      <select name="q3">
      <option value="" <%if Request("q3")="" then %>selected<%end if%> >
      <option value="-1d" <%if q3="-1d" then %>selected<%end if%> >day
      <option value="-1w" <%if q3="-1w" then %>selected<%end if%> >week
      <option value="-1m" <%if q3="-1m" then %>selected<%end if%> >month
      <option value="-1y" <%if q3="-1y" then %>selected<%end if%> >year
      </select>
      </td>
      <td></td>
      <td><font size=2>
      Sent to: (alias)
      </font></td>
      <td>
   <input type=hidden name="c5" value="@MessageFolderName">
      <input type=text name="q5" size=25 maxlength=50 value="<% = q5 %>">
      </td>
      </tr>
      <tr><td>
      <font size=2>Subject:</font>
      </td>
      <td>
      <input type=hidden name="o4" value="@DocTitle">
      <input type=text name="q4" size=25 maxlength=50 value="<% = q4 %>">
      </td>
      <td></td>
      <td><font size=2>
      Folder:
      </font></td>
      <td>

      <input type=hidden name="c6" value="@MessageFolderName">
         <input type=hidden name="q6" size=25
            maxlength=50 value="<% = q6 %>">
         <select name="ct" value="ct">

<% 'Enter new catalog names here
%>

      <!-- Use the following syntax for to search all catalogs
      <option value="Discussion,ProductKnowledge,Business,list_Servers">
         All Indexed Folders
      -->
      <!-- You need to put your catalog name here -->
         <option value="Discussion"> \Discussion


         </Select>
      </td>
      </tr>
```

(continued)

```
   <tr>

<!-- Add search by category -->
   <td><font size=2>
   Category:
   </font></td>
<td><input type=hidden name="c7" value="@META_Type">
   <select name="q7">
   <option value="" <%if q7="" then %>selected<%end if%> >
   <option value="Best Practice" <%if q7="Best Practice" then %>
      selected<%end if%> >Best Practice
   <option value="Idea" <%if q7="Idea" then %>selected<%end if%> >Idea
   <option value="Issue" <%if q7="Issue" then %>
      selected<%end if%> >Issue
   <option value="Resolution" <%if q7="Resolution" then %>
      selected<%end if%> >Resolution
   </select>

</td>
<td></td>

<!-- Add search by industry -->
   <td><font size=2>
   Industry:
   </font></td>
<td><input type=hidden name="c8" value="@META_Industry">
   <select name="q8">
   <option value="" <%if q8="" then %>selected<%end if%> >
   <option value="Communication and Entertainment"
      <%if q8="Communication and Entertainment" then %>
      selected<%end if%> >Communication and Entertainment
   <option value="Distributed Services"
      <%if q8="Distributed Services" then %>
      selected<%end if%> >Distributed Services
   <option value="Financial Services" <%if q8="Financial Services"
      then %>selected<%end if%> >Financial Services
   <option value="Government" <%if q8="Government" then %>
      selected<%end if%> >Government
   <option value="Healthcare" <%if q8="Healthcare" then %>
      selected<%end if%> >Healthcare
   <option value="Information Systems(
      Hardware/Software)" <%if q8="Information Systems(
      Hardware/Software)" then %>selected<%end if%> >
      Information Systems(Hardware/Software)
   <option value="Manufacturing" <%if q8="Manufacturing" then %>
      selected<%end if%> >Manufacturing
   <option value="Professional Services" <%if q8="
      Professional Services" then %>selected<%end if%> >
      Professional Services
   <option value="Transportation" <%if q8="Transportation" then %>
```

```
      selected<%end if%> >Transportation
   </select>
</td>
</tr>
<tr>

<!-- Add search by product -->
   <td><font size=2>
   Product:
   </font></td>
<td><input type=hidden name="c9" value="@META_Product">
   <select name="q9">
   <option value="" <%if q9="" then %>selected<%end if%> >
   <option value="(Other)" <%if q9="(Other)" then %>
      selected<%end if%> >(Other)
   <option value="Access" <%if q9="Access" then %>
      selected<%end if%> >Access
   <option value="Excel" <%if q9="Excel" then %>
      selected<%end if%> >Excel
   <option value="Exchange Server" <%if q9="Exchange Server" then %>
      selected<%end if%> >Exchange Server
   <option value="Mail" <%if q9="Mail" then %>selected<%end if%> >Mail
   <option value="Office" <%if q9="Office" then %>
      selected<%end if%> >Office
   <option value="Outlook" <%if q9="Outlook" then %>
      selected<%end if%> >Outlook
   <option value="PowerPoint" <%if q9="PowerPoint" then %>
      selected<%end if%> >PowerPoint
   <option value="Visual Basic" <%if q9="Visual Basic" then %>
      selected<%end if%> >Visual Basic
   <option value="Visual C++" <%if q9="Visual C++" then %>
      selected<%end if%> >Visual C++
   <option value="Word" <%if q9="Word" then %>selected<%end if%> >Word
</select>
</td>

   <td colspan=5 align=right>

   <% 'Links to tips and search button
      L_SearchButton_label = "Search"
      L_SearchTips_link = "Tips"
   %>
         
      <input type="submit" value="<% = L_SearchButton_label %>">
         
      <font size=2><a href="tips.htm"><% = L_SearchTips_link %></a>
      </font>
      </td>
```

(continued)

```
      </tr>
   </table>
  </form>
<% 'End of Search Initiation Section %>
```

The next section of the code checks whether the user has an ActiveX control installed on his computer to launch Outlook to view the item. This ActiveX control is provided with Site Server. The filename for the control is exciol.ocx, and the *CodeBase* property for the control points to C:\siteserver\knowledge\search\controls\exciol.ocx by default.

This control has one method that you can call: *DisplayMsg*. The *DisplayMsg* method takes the URL to the item, which you can obtain from the *DocAddress* property in the ADO recordset of results. Then the ActiveX control will automate Outlook to display the item. If the user doesn't have Outlook, or if you don't want to let the user view the results using Outlook, she can use Outlook Web Access to view the item.

```
<% if Request.QueryString <> "" then %>
<%
'If site visitors use Outlook to view Exchange Server messages,
'the Exciol control is installed on their computers
if ExchangeViewer="both" or ExchangeViewer="outlook" then %>
<object id="Exciol" height=0 width=0
   CLASSID="CLSID:DAFD7A40-73FF-11D1-A811-00AA006EAC9D"
   CODEBASE="/siteserver/knowledge/search/controls/
exciol.ocx#version=5,5,2148,0"
   TYPE="application/x-oleobject">
</object>

<script language="vbscript">
   Sub DisplayMsg(EntryID)
      Exciol.DisplayMsg(EntryID)
   End Sub
</script>
<% end if %>
<% if ExchangeViewer="both" or ExchangeViewer="owa" then %>
<% 'When linking to Outlook Web Access, open the link in a new window %>
<script language="javascript">
function openNewWindow(fileName,theWidth,theHeight) {
   window.open(fileName,"NewWindow","toolbar=0,location=0,directories=0,
      status=1,menubar=1,scrollbars=1,resizable=1,width="+theWidth+",
      height="+theHeight)
}
</script>
<% end if %>
```

The next section of code creates the user's query by using the Site Server Query object. As you saw earlier, the Query object has a number of properties that you can use to create complex queries against Site Server catalogs. In the search.asp code, the first step is to create the Query object by calling *Server.CreateObject* with the ProgID of the Query object, *MSSearch.Query*. Since the application passes all the information about the query along the URL query string, you can use the method *SetQueryFromURL* to have the Query object automatically parse the URL query string and set the Query object properties according to the specified conventional query string variables.

```
<% 'Set query object properties
   set Q = Server.CreateObject("MSSearch.Query")
   'Define all required query properties: the query itself, and all columns
   'that will be used in the results
   Q.SetQueryFromURL(Request.QueryString)
   Q.Catalog = request("ct")
   Q.OptimizeFor="nohitcount,performance"
   Q.MaxRecords = 10
   Q.Columns = "DocAuthor,DocTitle,DocAddress,Description,Size,FileWrite"
   %>
```

The next step in creating the search results is to actually create the recordset of items that match your criteria. To do this, you need to call the *CreateRecordset* method on the Query object. This method will create an ADO sequential, forward-only recordset that you can use to display your results. As you can see in the code, different properties are pulled from the Site Server catalog to display the resultset. The code uses standard ADO methods and properties to scroll through the resultset.

```
<% 'Create the recordset holding the search results
   on error resume next
   set RS = Q.CreateRecordSet("sequential")
   if err <> 0 then
   createerror = err.description
   end if
'If the query can't be executed, print out the error description
   if err then
      Response.write "  " & createerror

   'If no matches are found, display a message
      elseif RS.BOF and RS.EOF then
         Response.write "<p>"
         if Q.QueryIncomplete=true then
         L_TooComplex_Error = "The query is too complex. " & _
            "Try using a simpler query.For information on search " & _
            "syntax, see <a href=tips.htm>Search Tips</a>."
```

(continued)

```
            Response.write L_TooComplex_Error & "<p>"
         else
            L_NoMatch_Error = "No messages matching your query were " & _
            "found. For suggestions on how to broaden your search, " & _
            "see <a href=tips.htm>Search Tips</a>."
         Response.write L_NoMatch_Error & "<p>"
      'Display link for a new search
      L_NewSearch_link = "New Search"
      %>
        <a href="Search.asp"><% = L_NewSearch_Link %></a>
      <%
         end if

      else 'If query could be executed, display results

      'Set up the table for displaying results
%>
      <table cellpadding=0>
         <tr>
         <td colspan=2><font size=2>

            <% 'Make introductory sentences match the information
               'displayed
               if Request("sh") = "" then
                  L_Match_text = _
                     "Here are the messages found matching the query"
               else
                  L_Match_text = _
                     "Here are more messages found matching the query"
               end if
                  Response.write L_Match_text & "<b> " & _
                     DisplayText & "<b>."
               %>
               <br> 
      <% if ExchangeViewer="both" then
               L_OWA_Info = "For Microsoft&reg; Exchange messages, " & _
                  "click the author's name to view the message " & _
                  "using Microsoft Outlook. Click the " & _
                  "<img srC=html.gif width=16 height=16 " & _
                  "border=0 align=middle align=middle> " & _
                  "icon to view the message using Microsoft " & _
                  "Outlook Web Access."
               Response.write L_OWA_Info
            end if %>
         <br> 
      </font></td>
   </tr>
</table>
```

```
<table cellpadding=0 cellspacing=0>
<tr bgcolor=cccccc>

<%

L_KBytes_text = "KB"
L_Received_text = "Received"
L_Subject_text = "Subject"
L_Size_text = "Size"
L_From_text = "From"

%>
<td></td>
<td><font size=2><b><% = L_From_text %></b></font></td>
<td></td>
<td><font size=2><b><% = L_Subject_text %></b></font></td>
<td></td>
<td><font size=2><b><% = L_Received_text %></b></font></td>
<td></td>
<td><font size=2><b><% = L_Size_text %></b></font></td>
<td></td>

</tr>
<%
'Set up loop to iterate through results
   Do while not RS.EOF

      'Determine format type; set up title for and URL for links
         if InStr(RS("MimeType"), "text/exch") then
            DocType="exchange"
         else
            DocType="doc"
      end if

   'If message title is blank, use "No subject" instead
      if RS("DocTitle") <> "" then
         Title = RS("DocTitle")
      else
         L_Untitled_text = "No subject"
         Title = L_Untitled_text
      end if

  'Provide alternate text if no author exists. HTMLEncode is
  'required to handle author fields with < or >.
     if RS("DocAuthor") <> "" then
        Author = Server.HTMLEncode(RS("DocAuthor"))
     else
```

(continued)

```
            L_NoAuthor_text = "Author unknown"
            Author = L_NoAuthor_text
        end if

        'Set up link itself. Link depends on whether item is document or
        'Microsoft Exchange Server message.
        if DocType="doc" then
            Link = RS("DocAddress")
            Image = "html.gif"
        elseif DocType="exchange" then
            Image = "owa.gif"
            if ExchangeViewer="owa" then
                Link = "JavaScript:self.openNewWindow(" & chr(34) & _
                    RS("DocAddress") & "&usemainwnd=1" & chr(34) & ", 640,500)"
            elseif ExchangeViewer="outlook" or ExchangeViewer="both" then
                Link = "VBScript:self.DisplayMsg(" & chr(34) & _
                    RS("DocAddress") & "=1" & chr(34) & ")"
            end if
                if ExchangeViewer="both" then
                OWALink = "JavaScript:self.openNewWindow(" & chr(34) & _
                    RS("DocAddress") & "&usemainwnd=1" & chr(34) & ", 640,500)"
            end if
        end if

    %>

    <% 'Create table row for each result %>
    <tr><td> </td></tr>
        <tr>
            <% if DEBUGINFO=true or Request("debug") <> "" then %>
            <% 'Display column with record number %>
                <td valign=top><font size=2><% = RecordNum %>.</font></td>
        <% end if %>
        <td valign=top>
            <a href='<% = Link %>'><img src="<% = Image %>" hspace=6 border=0>
                </a>
        </td>
        <td width=20% valign=top><font size=2>
            <a href='<% = Link %>'><b><% = Author %></b></a>
        <% if ExchangeViewer = "both" and DocType="exchange" then %>
            <a href='<% = OWALink %>'>
                <img src="html.gif" height=16 width=16 border=0 align=top></a>
        <% end if %>
        </font></td>
        <td width=5> </td>
        <td width=55% valign=top><font size=2>
            <b><% = Title %></b>
        </font></td>
```

```
   <td width=5> </td>
   <td valign=top nowrap><font size=1>
       <% = RS("FileWrite") %>
   </font></td>
   <td width=5> </td>
   <td valign=top><font size=1>
      <% iSize = CInt(CLng(RS("Size"))/1024) %>
      <% Response.write iSize & " " &  L_KBytes_text%>
   </font></td>


</tr>
<tr>
<td></td>
<td colspan=6><font size=2>
<table cellpadding=0 cellspacing=0>
<td width=30> </td>
<td><font size=1>
<% = RS("Description") %>
</font></td>
</tr>
</table>
</td>
</tr>
<% 'Increment the results
   RS.MoveNext
   RecordNum = RecordNum + 1
   Loop
%>
<% 'If query times out, display message stating that more results are
   'available %>
<%
   if Q.QueryTimedOut = TRUE then
   Response.write "<tr><td></td><td colspan=8><font size=2><b>"
   L_QueryTimedOut_error = "Not all matching messages were returned." & _
      "To see all the messages, at the top of the page, " & _
      "click Search again."
   Response.write " <p>" & L_QueryTimedOut_error
   Response.write "<b></td></tr><font size=2>"
end if
%>
</table>
```

The final section of code implements the More Results functionality at the bottom of the Web page. Since 10 results appear on a page, you need to implement this functionality so that the user can look at all the results returned in the query. This section of code uses the *MoreRows* property to determine whether there are more results than currently displayed. If there are more rows, the *StartHit* property is set

on the Query object to the start hit on the next page. Then the code generates the hyperlink for the More Results text to pass the same query string to search.asp so that the application performs the same query on the catalog but returns the next resultset.

```
<hr>
<table cellpadding=4>
<tr><td><font size=2>

<% 'Display a More Results link if there are more results pages

      if RS.Properties("MoreRows") = true then

         Q.StartHit = RS.Properties("NextStartHit")

         'Repeat query with new start hit. The query must include any
         'custom variables you've used: in this case, Text and RecordNum.
         Response.write "<a href=search.asp?" _
            & Q.QueryToURL & "&" _
            & "Category=" & Request("Category") & "&" _
            & "DisplayText=" & Server.URLEncode(DisplayText) & "&" _
            & "RecordNum=" & RecordNum _
            & ">"
         L_MoreResults_link = "More Results"
         Response.write L_MoreResults_link & "</a>  "
      end if
      %>

      <% 'Display link for a new search %>
      <% L_NewSearch_link = "New Search" %>
        <a href="Search.asp"><% = L_NewSearch_Link %></a>

   </font></td></tr>
</table>
```

EXTENDING OUTLOOK WITH SITE SERVER

You can leverage Site Server from your Outlook 2000 applications in a number of ways. These methods range from simply hosting the Web Browser control in an Outlook form to perform your queries, to writing a full-blown COM add-in that leverages the Site Server object model to create a rich search application.

In this section, we'll examine four methods for integrating Site Server into Outlook. First, you'll see how you can host your ASP search application as a folder home page. Next, you'll discover how to easily host the Web Browser control in order to host a customized version of an ASP search page. Then, you'll learn how to add controls to an Outlook form in conjunction with the Web Browser control to make

your Outlook application more powerful. We will also take a close look at add-ins and their features. Finally you'll see a custom COM add-in that performs a total search solution for Outlook users by using Site Server catalogs.

Hosting the ASP Search Application as a Folder Home Page

One option for integrating your Web-based Site Server search applications into Outlook 2000 is to host your search application as a folder home page. While this is the easiest way to quickly provide search capabilities in Outlook, it leaves a lot to be desired from the standpoint of Outlook integration. Users would have to navigate to the folder hosting the Web page to use the search capabilities.

Hosting the ASP Search Application in an Outlook Form

Another option for integrating Site Server into your Outlook environment is to host the Web Browser control in an Outlook form. This makes it easy for users to initiate a search; rather than going to a folder, they can simply bring up a form. Figure 17-9 shows an Outlook form that implements this solution.

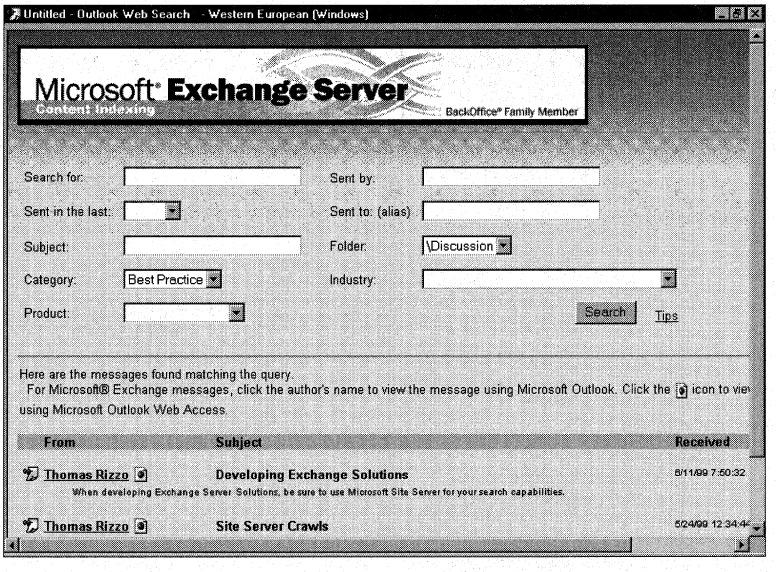


Figure 17-9. *The Web Browser control hosted inside an Outlook form to provide search capabilities.*

Notice in the figure how all the command bars that are usually available in the form have been disabled. You can use some VBScript code behind the Outlook form

to disable the command bars, as the following code shows. This code also grabs the Web Browser control on the form and forces it to navigate to the ASP search application.

```
Function Item_Open()
   'Disable the command bars
   for each commandbar in Item.GetInspector.CommandBars
      commandbar.enabled = False
   next
   set oBrowser = _
      Item.GetInspector.ModifiedFormPages("Search").Controls("WebBrowser1")
   oBrowser.Navigate "http://airliftone/search/search.asp"
End Function
```

Extending the ASP Search Application Using Outlook Controls

While the previous example is better than a folder home page, you can take the integration a bit further by adding some custom controls to the form with the Web Browser control. For example, instead of a user having to guess the display name of the person who sent or posted the message, why not use the Global Address List (GAL) of Exchange Server and allow your users to select the person? Figure 17-10 shows a version of the Outlook form that uses Collaboration Data Objects (CDO), a tool that helps you implement a better user interface that's more integrated with Exchange Server and Outlook functionality.

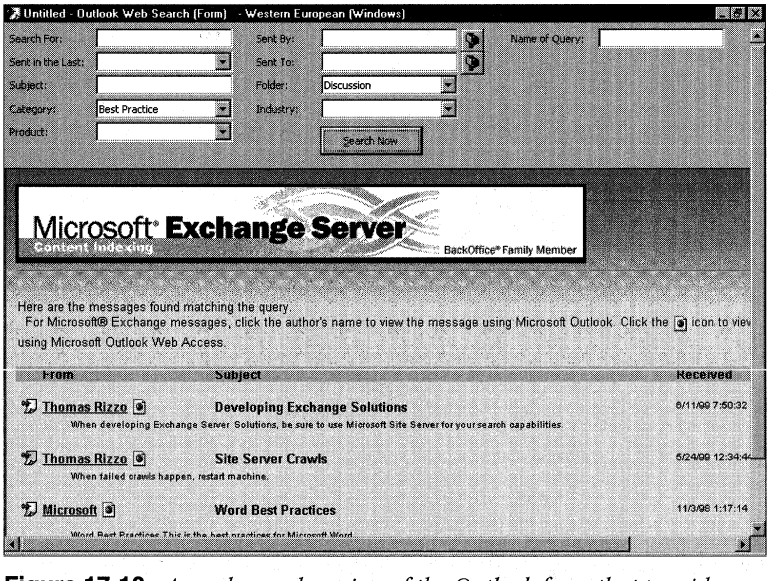


Figure 17-10. *An enhanced version of the Outlook form that provides controls for selecting users and search criteria.*

Once the user selects her criteria, she can click the Search Now button and the form will generate a query string using Site Server's conventional query string variables. The form will then automate the Web Browser control to navigate to a modified version of the ASP search application that has only the Web-based version of selecting criteria for the search removed. One nice thing about using this Outlook form is that the user can type in a name for the query and save the query into the folder for later use. Here is the code behind this Outlook form:

```
Dim oBrowser
Dim strURL
strURL = ""
strServerName = "airliftone"

Function Item_Open()
   'Disable the command bars
   for each commandbar in Item.GetInspector.CommandBars
      commandbar.enabled = False
   next
   set oBrowser = _
      Item.GetInspector.ModifiedFormPages("Search").Controls("WebBrowser1")
   oBrowser.Navigate "http://" & strServerName & "/search/olsearch.asp"
End Function

Sub AddtoURL(strString)
   strURL = strURL & strString
End Sub

Sub cmdSearchNow_Click()
   'Initiate the search
   'Build the URL
   'Grab each component and pop it into the URL
   AddtoURL "c1=@All&q1=" & Item.UserProperties("SearchFor").Value
   AddtoURL "&c2=@DocAuthor&q2=" & Item.UserProperties("Sent By").Value
   'Figure out what time frame the user selected
   strTime = Item.UserProperties("Sent in the Last").Value
   if strTime = "" then
      q3 = ""
   elseif strTime = "day" then
      q3 = "-1d"
   elseif strTime = "week" then
      q3 = "-1w"
   elseif strTime = "month" then
      q3 = "-1m"
   elseif strTime = "year" then
      q3 = "-1y"
   end if
   AddtoURL "&c3=@FileWrite&o3=>=&q3=" & q3
```

(continued)

```
    AddtoURL "&o4=@DocTitle&q4=" & Item.UserProperties("strSubject").Value
    AddtoURL "&c5=@MessageFolderName&q5=" & _
       Item.UserProperties("Sent To").Value
    AddtoURL "&c6=@MessageFolderName&q6=&ct=" & _
       Item.UserProperties("Folder").Value
    AddtoURL "&c7=@META_Type&q7=" & Item.UserProperties("strCategory").Value
    AddtoURL "&c8=@META_Industry&q8=" & _
       Item.UserProperties("Industry").Value
    AddtoURL "&c9=@META_Product&q9=" & Item.UserProperties("Product").Value
    oBrowser.Navigate "http://" & strServerName & "/search  " & _
       "olsearch.asp?" & strURL

End Sub

Sub cmdSentByAddress_Click()
   FindAddress "Sent By", "Search for Sent By", "Sent By"
End Sub

Sub cmdSentToAddress_Click()
   FindAddress "Sent To", "Search for Sent To", "Sent To"
End Sub

Sub FindAddress(FieldName, Caption, ButtonText)
   On Error Resume Next
   Set oCDOSession = application.CreateObject("MAPI.Session")
   oCDOSession.Logon "", "", False, False, 0
   txtCaption = Caption
   if not err then
      set orecip = oCDOSession.addressbook ( _
         Nothing, txtCaption, True, True, 1, ButtonText, "", "", 0)
   end if
   if not err then
      item.userproperties.find(FieldName).value = orecip(1).Name
   end if
   oCDOSession.logoff
   oCDOSession = Nothing
End Sub
```

Building an Outlook 2000 COM Add-in for Site Server

The final and best option in my opinion is to leverage the COM add-in capabilities of Outlook 2000 to create a well-integrated search solution. Before discussing the code for the COM add-in, let me first describe the features of the add-in because we won't look at all the code.

The add-in extends the menus and toolbars of Outlook to provide users with quick access to its interface. The add-in also provides a rich set of querying capabilities. Users can create AND or OR clauses in their searches. Once the criteria are set, the add-in uses the Flexgrid control to display the results, as shown in Figure 17-11.

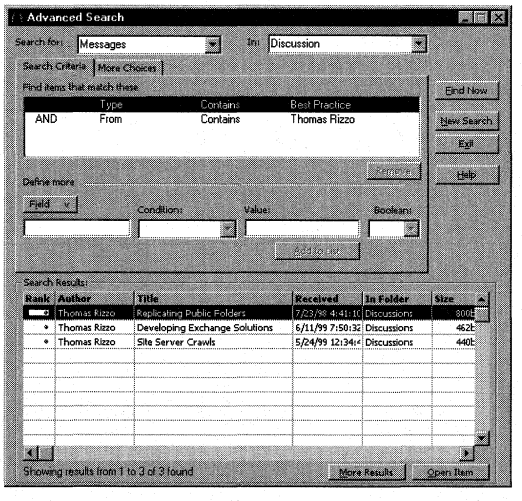


Figure 17-11. *The COM add-in with multiple AND clauses and the results of the query in the Flexgrid control.*

The add-in displays the most common search fields for Site Server information, but it also allows you to automatically detect custom fields contained in the folders being cataloged by Site Server. This feature makes it easier for users to determine all the custom fields that they can search on in the Site Server catalog. Figure 17-12 illustrates this feature.

In addition to detecting custom fields, the add-in automatically detects all your Site Server catalogs. This allows the user to quickly decide which catalog to search, while giving you the ability to add new catalogs without modifying the form. This capability uses an object model for Site Server that we haven't discussed yet. In a moment, you'll learn how to leverage the Site Server SearchAdmin object model to add some great features to your search add-in.

When viewing the results of the query, the add-in supports showing the ranking as both graphics and numbers. It also supports autopreviewing messages using the *Description* property you learned about earlier. The add-in also provides sorting on the column headings in the Flexgrid control. Figure 17-13 shows the autopreview feature.

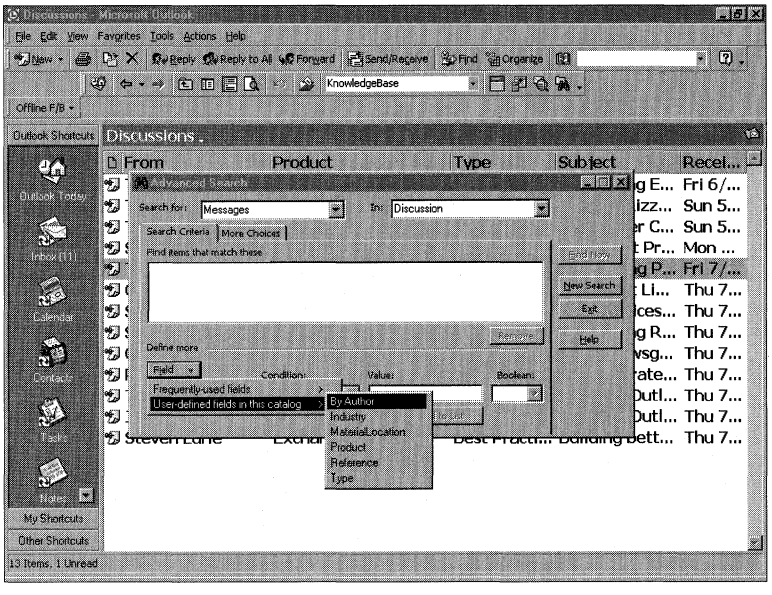


Figure 17-12. *Automatically detecting custom fields in the COM add-in.*

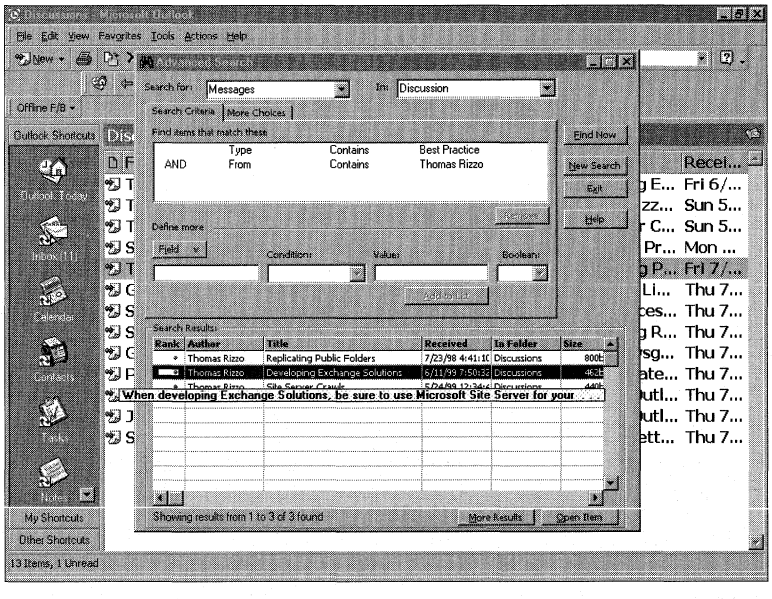


Figure 17-13. *Autopreviewing an item in the search results.*

Another feature this add-in provides is a customizable interface that lets you try different configurations in Site Server, such as choosing different optimization settings, allowing enumeration, and printing the query string variables before sending them to Site Server for debugging purposes. Figure 17-14 shows the configuration page of the add-in.

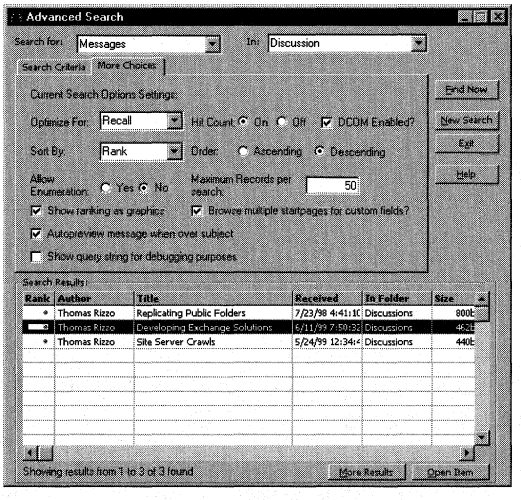


Figure 17-14. *The configuration page of the add-in, where you can set different options to test your queries.*

Finally, this add-in can easily be enabled for DCOM. The only way you can run this add-in without DCOM support is to install Outlook 2000 on a computer running Site Server and perform all your queries there. It's unlikely that your users will want to walk to the server machine to perform their queries. Enabling the add-in for DCOM allows your users to take advantage of the power of Site Server from their own computers.

I want to point out one thing before you look at the code for the add-in. Everything you'll see in the following section you can leverage from ASP, COM add-ins, or event-scripting agent applications. For example, you could add a timer-based event-scripting agent to a top-level folder, and as subfolders are added beneath that top-level folder, you could create new catalogs that index the contents of those folders. Note that you can have a maximum of 32 catalogs on a search server, and each catalog can have a maximum of 5 million documents.

Enabling the Add-in for DCOM

Let's first look at the code that will enable the add-in for DCOM. Building the code to enable the add-in for DCOM actually is straightforward in Microsoft Visual Basic 6. Since the *CreateObject* method in Visual Basic 6 takes an optional string value that specifies the server on which you want to create the object, the add-in simply leverages that capability and passes in the name of the computer running Site Server. This name is a global constant set in the Visual Basic project. The code to create the MSQuery object in the add-in looks like the following, with *strDCOMServer* holding the string that specifies the computer running Site Server on which to create the object:

```
Set q = CreateObject("MSSearch.query.1", strDCOMServer)
```

You might think this is easy and wonder why I mentioned it. The reason I did is that simply putting this code in your add-in won't magically create the component on the computer running Site Server. You need to make sure that the client machine attempting to create the remote component has information in its registry about this remote component; otherwise, the add-in can't create the remote component.

The easiest way to get this information into the client computer's registry and host the remote component is to leverage Microsoft Transaction Server (MTS). By adding a new package to MTS that contains both the MSSearch object and the SearchAdmin object, you can have MTS create a client setup program that will install the correct information about the remote components into the registry. To do this, click on the package in the MTS administration console and choose the Export package option. (See Figure 17-15.)

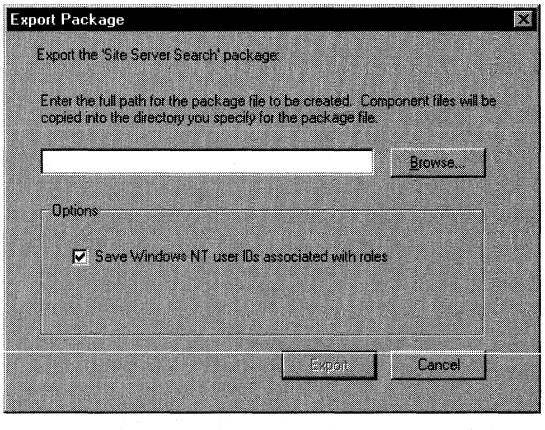


Figure 17-15. *Exporting a package from Microsoft Transaction Server.*

As part of the exported package, you'll find the Clients subdirectory. In the subdirectory, you'll find an executable that you can run on your client computers. This executable will install the necessary information into the registry to allow the client to remotely create the component on another server. Unfortunately, I could not include an exported package for you on the companion CD. Since MTS puts some server-specific information into the client setup program, you don't want my server name as the remote server you're creating the objects on.

When you add the objects to MTS, keep a couple of issues in mind. First, the MMC administration console of Site Server might tell you that search services are not correctly installed. This means you'll have to use the Web-based administration for Site Server to configure your catalogs. Second, you might want to change the identity that the components run under. By default, they'll run under the interactive user on your Windows NT server. If you keep your computers in a data center or reboot them without logging on, it might be difficult to have an interactive user always logged on to the server. Most likely, you wouldn't be able to create the components until someone logs in to the server as an interactive user. Instead, you might simply want to specify a Windows NT or Windows 2000 Server account that has the correct permissions for both components.

Automatically Detecting the Site Server Catalogs

The ASP search application does not provide the ability to automatically detect Site Server catalogs, meaning that as new catalogs get created, the application must be modified to reflect them. The add-in automatically detects all catalogs by using the SearchAdmin object model in Site Server. You'll see some information about the SearchAdmin object model here, but if you want the full documentation of the object model, see the Site Server SDK in the Platform SDK. The SearchAdmin object model can do much more than you'll see here.

Figure 17-16 displays the SearchAdmin object model. As the figure shows, you can reach the build and search servers for Site Server. Under those servers, you can access the properties for those catalogs, the catalog start pages, and other information that might be useful to your application.

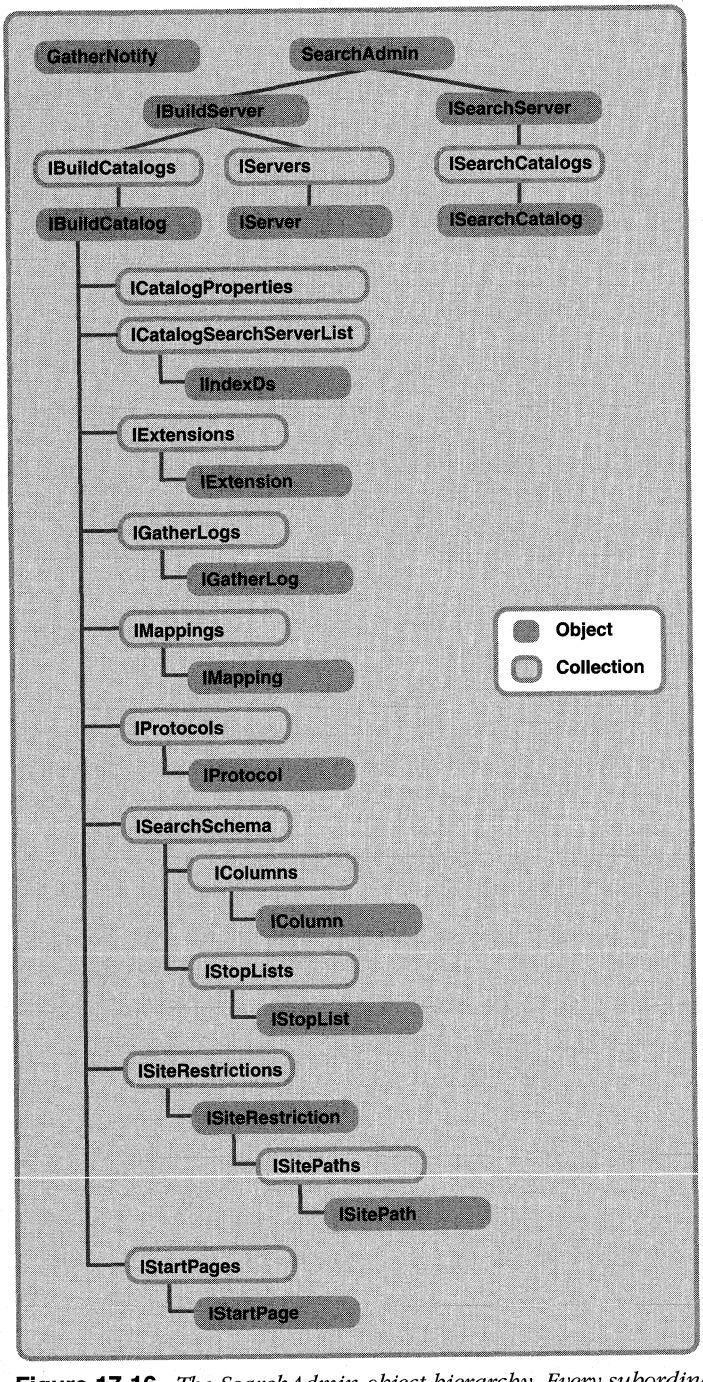


Figure 17-16. *The SearchAdmin object hierarchy. Every subordinate object must be created under the SearchAdmin object.*

The catalog names are retrieved by using the SearchServer object under the SearchAdmin object. From the SearchServer object, the SearchCatalogs object is used. The SearchCatalogs object contains SearchCatalog objects that identify each catalog on the server. In the SearchCatalog object, the *name* property is retrieved and added to the drop-down list in the add-in. Here is the code for retrieving catalog numbers:

```
If objHost Is Nothing Then
   If checkDCOMEnabled = vbChecked Then
      Set objHost = CreateObject("Search.SearchAdmin.1", strDCOMServer)
   Else
      Set objHost = CreateObject("Search.SearchAdmin.1")
   End If
End If

For Each cat In objHost.SearchServer.SearchCatalogs
   Cataloglist.AddItem cat.Name
Next
```

You'll see how to leverage more of the SearchAdmin object model when we discuss retrieving the custom fields for a catalog later in this section. However, you could create catalogs, delete them, or even query their current properties using the SearchAdmin object model. If you plan to build search solutions with Site Server, I recommend you become very familiar with this object model.

Working with Message Types

By using the MessageClass column, the add-in allows a user to pinpoint the exact type of item that she's looking for in the catalog. Through this capability, the user can specify only Outlook items of a certain type—such as mail, post, or contact items—or she can specify any type of item—including Web pages, files, or database information contained in the catalog.

Note that when you're testing the add-in, it defaults to messages only. If you have any free documents, HTML pages, or other nonmessage items contained in the catalog, those items will not appear. For a long time, I thought the add-in wasn't working before realizing my mistake; I then changed the item type to allow all types of items in the Search For drop-down list.

Speaking of item types, Site Server will crawl contacts, appointments, and other Outlook item types stored in Exchange Server public folders. The only catch is that Site Server won't index all the properties on these types of items; Site Server will index only the properties in the Exchange Server property set, and it will map other properties into the corresponding Site Server property set. Therefore, you can search only on Outlook properties that map to well-known Site Server properties. For example, the full name for a contact maps to the subject field in Outlook, so you can search by that full name using Site Server. However, you cannot search by the address or company name of the contact because these do not map to any of the properties that Site Server indexes for Exchange Server.

If you were to implement all the contact properties as custom properties, instead of using the built-in Outlook properties, you could search those contact and custom properties using the HTML META property set. Another option would be to create a "shadow form"—a hidden copy of the data from your contact form in another folder, which contains the contact fields as custom properties. Then you could have Site Server index and search the shadow forms. With the shadow forms, you need to implement code to redirect Outlook to the real contact form.

Working with Dates

One of the things you need to watch out for when specifying the Site Server search parameters is using dates in your application. As mentioned earlier, you need to put dates into a recognized format, such as yyyy/mm/dd. Since most users probably won't enter their dates in this format when performing their searches, the add-in automatically detects dates and formats them. The following code shows how the add-in formats the dates:

```
boolIsDate = False
boolIsDate = IsDate(conditionlist.Column(3, i))
If boolIsDate = True Then
   'Start flipping; first convert to date
   dDate = CDate(conditionlist.Column(3, i))
   dNewDate = Year(dDate) & "/" & Month(dDate) & "/" & Day(dDate)
End If
'Check to see whether IsDate and operand is =
'Need to set for On (or =) < Day after and > Date before
'because Site Server will try to match the hour and minute
'as well!
If boolIsDate And operand = "=" Then
   boolNeedAdditionalOperand = True
   'Change to a new operand
   strNewoperand = "<"
   strNewoperand2 = ">"
   querystring = querystring & "&o" + CStr(argnum) & "=" & strNewoperand
Else
   querystring = querystring & "&o" + CStr(argnum) & "=" & operand
End If
If boolIsDate = False Then
   querystring = querystring & "&q" + CStr(argnum) & _
      "=" & conditionlist.Column(3, i)
ElseIf operand <> "=" Then
   querystring = querystring & "&q" + CStr(argnum) & "=" & dNewDate
ElseIf operand = "=" Then
   'Add a day to the date
   dDatetoLookFor = DateAdd("d", 1, dNewDate)
   'Flip the date
   dDatetoLookFor = Year(dDatetoLookFor) & "/" & Month(dDatetoLookFor) _
      & "/" & Day(dDatetoLookFor)
```

```
   querystring = querystring & "&q" + CStr(argnum) & "=" & dDatetoLookFor
   'Get the field name so we remember it later
   strGlobalFieldName = fieldname
End If
```

As mentioned earlier in the chapter, dates in Site Server are stored according to UTC. Most users will not enter UTC dates but instead will enter dates corresponding to their own time zone. This can yield strange results; the returned values might not look correct to the user but in fact will be correct according to the UTC.

While the add-in doesn't solve this discrepancy for users, I have some suggestions for how you can remedy the problem. First, grab the local time zone information from the user's computer. You have two choices for doing this. You can go to the registry under the key C:\HKEY_Local_Machine\SYSTEM\CurrentControlSet\Control\TimeZoneInformation. The Bias value under this key specifies the current bias, in minutes, between the local time and UTC time. You can use this value to change the date the user entered into a UTC date.

The second way you can solve this problem is by using the Win32 API function *GetTimeZoneInformation*. Then you can retrieve the Bias value and figure out the UTC time. The UTC time will be local time + Bias.

Searching for Custom Fields

The COM add-in provides a nice feature of scrolling through the public folders contained in the catalog and displaying any custom fields contained in those folders. The add-in implements this feature by combining the SearchAdmin object model, a Site Server DCOM helper object, and CDO.

The Site Server DCOM helper object plays an important role in determining the custom fields in the folder. You might run into a situation in which the SearchAdmin object model doesn't allow you to enumerate certain properties or objects contained in the object model. This can happen for a number of reasons, but I've found the primary reason to be enabling authorization checking in MTS. To help circumvent this, I created the Site Server DCOM helper object. This object uses the SearchAdmin object model to retrieve all the Exchange Server start pages contained in the catalog. You need to do this to determine which Exchange Server public folders to search for the custom fields. This is the only way you can locate those folders with Site Server. Here is the code that performs this search:

```
Dim objHost

Public Function GetStartPageURL(strCatalogName) As Variant
   On Error Resume Next
   Dim url()
   Dim boolFoundExchange    ' Found an Exchange Server start page
   iCount = -1              ' Number of Exchange Server start pages
   boolFoundExchange = False
```

(continued)

```
    If strCatalogName <> "" Then
       Set objHost = CreateObject("Search.SearchAdmin.1")
       Set objCatalogBuilder = objHost.BuildServer   'CatalogBuilder
       Set objCatalogs = objCatalogBuilder.BuildCatalogs
       Set objSelectedCat = objCatalogs.Item(strCatalogName)
       Set objStartPages = objSelectedCat.StartPages
       For Each startpage In objStartPages
          If InStr(startpage.url, "exch://") = 1 Then
             boolFoundExchange = True
             ' Redim the array
             iCount = iCount + 1
             ReDim Preserve url(iCount)
             url(iCount) = startpage.url + "|" + _
                CStr(startpage.enumerationdepth)
          End If
       Next
       If boolFoundExchange = False Then
          ReDim Preserve url(0)
          url(0) = ""
       End If
    Else
       ReDim Preserve url(0)
       url(0) = ""
    End If
    GetStartPageURL = url
End Function
```

The Site Server DCOM helper object provides one function, *GetStartPageURL*. This function takes the catalog name and then uses the SearchAdmin object model to find that catalog and enumerate its start pages, which are formatted for Exchange Server as exch:\\Public Folders\All Public Folders\FolderName. When enumerating the start pages, the component figures out whether Site Server is also crawling all the subfolders of the starting folder and appends a pipe character (|) and then the enumeration depth. If Site Server doesn't crawl the subfolders of the starting address, the enumeration depth will be *–1*. This value causes the helper object to place all the paths to the Exchange Server folders into an array; the object then passes that array back to the COM add-in.

The CDO code takes the array passed back by the helper object and parses the array to get the folder paths and enumeration depths for the public folders. CDO also allows you to use the Fields collection to query the items contained in a pubic folder to see whether the items contain custom properties. You want to add these custom properties to the user interface and then search on those custom properties using the HTML META property set format we discussed earlier in the chapter. I won't dig deeply into the CDO code here because Chapter 12 will cover CDO in great detail. However, you should look at the code in Chapter 12 because it shows you how to enumerate subfolders and scan for custom properties contained in a folder.

Working with Ranks

Site Server also provide a helpful feature that ranks returned search results. The Rank column, a column calculated by Site Server, contains a number from 0 through 1000 that specifies how well the returned value matches the search criteria. The larger the number ranking, the better the match.

To help users better interpret the ranks, the COM add-in can show the ranks as either numbers or graphics. The user can customize the rank display and can sort by ranking. The following code illustrates how to show the ranking in the Flexgrid control:

```
If checkShowRankPicture = vbUnchecked Then
   'Set the column alignment to left, center
   Flex1.ColAlignment(1) = 1
   'Unload any pictures
   Flex1.Col = 1
   Flex1.Row = i
   Set Flex1.CellPicture = Nothing
   If rs("Rank") < 1000 Then
      Flex1.TextMatrix(i, 1) = Mid("0000", 1, _
         4 - Len(Trim(CStr(rs("Rank"))))) & Trim(CStr(rs("Rank")))
   Else
      Flex1.TextMatrix(i, 1) = Trim(CStr(rs("Rank")))
   End If
ElseIf checkShowRankPicture = vbChecked Then

   'Show pictures for the ranks
   Flex1.Col = 1
   Flex1.Row = i
   'Set the alignment for the column to center, center
   Flex1.CellPictureAlignment = 4
   iRank = 0
   iRank = CInt(rs("Rank"))
   'Make sure we should even load the picture
   If iRank <> 0 Then
      If (iRank > 750) Then
         Set Flex1.CellPicture = imgRanks.ListImages(4).Picture
      ElseIf (iRank > 500) Then
         Set Flex1.CellPicture = imgRanks.ListImages(3).Picture
      ElseIf (iRank > 250) Then
         Set Flex1.CellPicture = imgRanks.ListImages(2).Picture
      Else
         Set Flex1.CellPicture = imgRanks.ListImages(1).Picture
      End If
   End If
End If
```

The code determines whether to display the graphic or number representation of the rank. If the user wants the graphic representation, the code uses an Imagelist

control containing four images of bullets. The more bullets in the ranking, the better the match. You might want to modify the add-in to also allow users to filter by ranking. For example, you might want to allow users to show only items with a ranking of two bullets or higher.

Working with Non–Exchange Server Data in the Same Catalog

The final feature of the COM add-in that we'll discuss is working with non–Exchange Server data in the same catalog as Exchange Server data. You might want to provide a single catalog that spans Exchange Server, the file system, and the Web so that users can retrieve all that information with a single query. While you could provide this capability using multiple catalogs and Site Server, the add-in allows only a single catalog search. Therefore, the add-in must determine whether the user wants Exchange Server data or non–Exchange Server data.

To implement this, the add-in assesses whether the MessageFolderName column is empty for the record in the ADO recordset. If this column is empty, the item must be non–Exchange Server data. The following code implements this portion of the add-in:

```
If rs("MessageFolderName") = "" Then
   Flex1.TextMatrix(i, 5) = CStr(rs("DocAddress"))
   strType = "Not Exchange"
Else
   strType = "Exchange"
   Flex1.TextMatrix(i, 5) = CStr(rs("MessageFolderName"))
End If
```

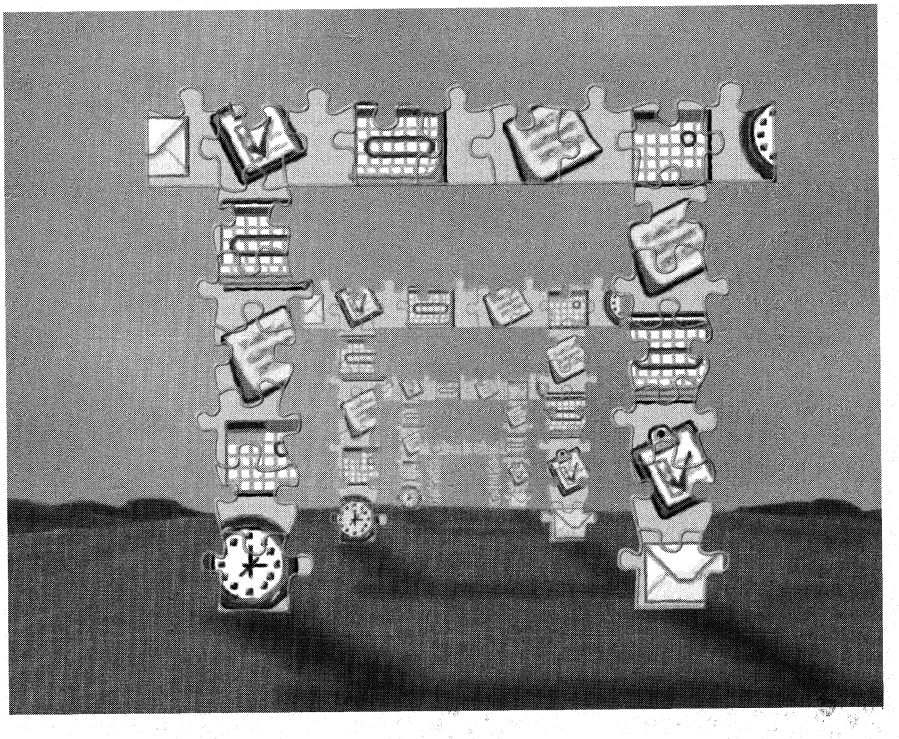
The add-in uses a hidden column in the Flexgrid control for each row to identify the type of data (Exchange Server or otherwise) by using the *strType* variable. This column is accessed when the user double-clicks on the item in the Flexgrid control or selects the row and clicks the item open. The COM add-in checks the type of item, and if the item is a non–Exchange Server item, the code prompts Microsoft Internet Explorer to open the item. If it's an Exchange Server item, the add-in uses the Outlook object model to open the item in Outlook. If you want to, you can use the Exciol.ocx control to open the item from the add-in. However, this probably isn't necessary because the add-in is running in Outlook, and users probably will prefer to view the item using Outlook rather than Outlook Web Access. The code for this functionality follows:

```
Private Sub flex1_DblClick()

Dim wshell As IWshShell_Class
   'See if it's an Exchange Server item
   If Flex1.TextMatrix(Flex1.Row, 8) = "Not Exchange" Then
      'Not an Exchange Server item
```

```
        Screen.MousePointer = 11
        If InStr(Flex1.TextMatrix(Flex1.Row, 0), "=") = 0 Then
           On Error Resume Next
           Set wshell = CreateObject("Wscript.Shell")
           If Err.Number <> 0 Then
              MsgBox "Can't Create Windows Scripting Host on this machine"
           Else
              wshell.Run "iexplore " & """" & _
                 Flex1.TextMatrix(Flex1.Row, 0) & """", 1, False
           End If
           Set wshell = Nothing
        End If
        Screen.MousePointer = 0
     Else
        On Error Resume Next
        Set alpffolder = olns.Folders("Public Folders").Folders( _
           "All Public Folders")
        Set citem = Nothing
        Set citem = _
           olns.GetItemFromID(UCase(Mid(Flex1.TextMatrix(Flex1.Row, 0), _
           InStr(Flex1.TextMatrix(Flex1.Row, 0), "=") + 1)), _
           alpffolder.StoreID)
        Screen.MousePointer = 0
        If citem Is Nothing Then
           MsgBox "The item could not be displayed.  The EntryID is " _
              & (UCase(Mid(Flex1.TextMatrix(Flex1.Row, 0), "="))) & _
              " and the store ID is " & alpffolder.StoreID
        Else
           citem.Display
        End If
     End If

End Sub
```

Part IV

Exchange Server 2000 Development



Chapter 18

Developing with Exchange Server 2000

Microsoft Exchange Server 2000, which builds on the extensive collaborative features of Exchange Server 5.5, offers many exciting new capabilities. This chapter and Chapter 19 will discuss the main enhancements made to the Exchange Server platform. In both these chapters, we'll use a sophisticated sample program, called the Training application, to highlight these new development capabilities.

These are some of the most significant Exchange Server 2000 enhancements:

- OLE DB and Microsoft ActiveX Data Objects (ADO) support
- Friendly URL access to every item in the Exchange Server database—meaning no more globally unique identifiers (GUIDs)!
- XML support
- A richer events model in the types of events supported and the programming of events
- A new version of Collaboration Data Objects (CDO)

- A new version of Microsoft Outlook Web Access (OWA) that's much easier to reuse
- An enhanced platform for real-time collaboration development
- A greatly improved and built-in workflow engine for building workflow applications

> **NOTE** This chapter assumes you are running RC1 of Exchange Server 2000; its samples and code were written to work with RC1. Please note that the semantics and features listed in this chapter may change for the released version of Exchange Server 2000. Be sure to check the Microsoft Press Web site, *http://mspress.microsoft.com,* for any Knowledge Base articles that highlight new code samples for this book.

WHAT IS THE WEB STORAGE SYSTEM?

You might have heard the term *Web Storage System*. What does the Web Storage System technology mean for you as a developer? For the past few years, Microsoft's Exchange Server group has been developing a great data storage technology. The one problem with this database technology is that it shipped only with Microsoft Exchange Server, so it was limited to products bearing the Exchange Server name. However, Microsoft has since renamed this database technology *Web Storage System* and ripped it out of Exchange Server. Not only is the Web Storage System technology now identifiable by name, it can be embedded into other Microsoft products. This means you can use the Web Storage System in situations that require a rich, semi-structured, Web-aware database you can access from a number of different client access methods. Furthermore, the Web Storage System technology provides a rich set of development services, which we'll look at in this chapter.

Exchange Server 2000 will be the first Microsoft product to ship with the Web Storage System technology. Following Exchange Server 2000, you'll see a number of other products shipping with this technology. If you learn the basics of Web Storage System development now, you'll have a great foundation for building applications on both Exchange Server 2000 and other products that ship with the Web Storage System.

Enough background. Let's talk about the core features of the Web Storage System technology. The core features fall into three main areas: data access, programmability, and security.

Data Access Features

One of the major new data access features of the Web Storage System is that it provides a native OLE DB 2.5 provider. This allows developers to write directly to OLE

DB interfaces to get or set information contained in the Web Storage System. Furthermore, it allows developers familiar with ADO to write ADO applications using the Web Storage System as the data store. We'll look closely at the ADO 2.5 support provided by the Web Storage System later in this chapter.

Another data access feature is Web Distributed Authoring and Versioning (WebDAV) support. Since the Web Storage System is tightly integrated with Microsoft Internet Information Services (IIS), the Web Storage System can provide rich access to data over Web protocols such as Hypertext Transfer Protocol (http). However, standard http commands, such as GET and POST, cannot provide you with a rich enough set of features to build collaborative applications. For this reason, the Internet Engineering Task Force (IETF) came up with WebDAV—extensions to http 1.1 that allow you to move, copy, query, and delete resources. For example, with WebDAV, you could create a new folder in the Web Storage System, create a new item in that folder, and then query for the new item using WebDAV and requests formatted in a specific XML format that WebDAV understands. You'll see some examples of using WebDAV in the next chapter when we examine the Training application.

The final way—and in my opinion, one of the most *interesting* ways—to get data from the Web Storage System is by using its Installable File System (IFS) provider. The IFS provider allows you to provide access to your data in the Web Storage System using standard file system programs or interfaces. For example, you can make the documents you create in the Web Storage System available to your application users via Microsoft Windows Explorer. Furthermore, without having to write a single line of code, you can turn any file system–aware application into an interface for your application's data. The Training application we'll look at later in the chapter will show you some ways you can use the IFS provider in your applications.

Programmability Features

The Web Storage System allows both Web and Windows developers a great amount of flexibility in their programming. Programmability features fall into five key areas: schema, form, event, workflow, and XML support. This section will give you an overview of each of these areas; you'll see more in-depth coverage when we look at the Training application.

Schema Support

Built directly into the Web Storage System is an extensive array of schema support. By having the ability to create a schema, developers can define sets of properties that are common to a certain type of item in the Web Storage System. Such support is similar to that of objects that support certain properties. The Web Storage System defines some built-in schemas you can take advantage of, such as a schema for an object or an item. Exchange Server 2000 specifically defines a schema for messages, appointments, and contacts.

One of the neat things about the Web Storage System schema support is that schemas are inheritable. This means that you can inherit schema properties from another schema collection and extend the inherited properties with your own properties. For example, suppose I want to create a customer schema definition that includes properties similar to those of the built-in contact schema as well as some custom properties for my application. I simply tell the Web Storage System that I want to inherit the properties from the built-in contact schema and extend it with my own properties.

By creating a custom schema, you are guaranteed that the correct properties will be returned when you submit searches using the SQL SELECT * syntax. This allows other application developers to simply traverse the ADO fields collection rather than know the actual names of your properties. For performance reasons, however, you might not want applications that use SELECT *, because the Web Storage System will have to return all the properties on the items contained in your search. For most Web Storage System items, this can be in excess of 100 properties.

Even though using schemas isn't required to build applications on the Web Storage System, I highly recommend that you use schemas in your applications, as appropriate. Most of the time, using schemas will be appropriate. However, using a schema might not make sense if you're only creating a one-off, single-use item for which you don't care whether the properties are reused on other items, or for which the property definitions are lost if the item is deleted. Furthermore, you might find creating a schema more tedious than simply appending new fields onto the item using ADO. The code for the workflow process in the Training application will show you an example of this. In that sample, it was easier to just append items onto the process instance in the workflow than to create a full-fledged schema definition for the process instance itself. However, you'll see other code samples in this chapter in which schema and their definitions play an important role.

Web Storage System Forms

The Web Storage System supports an HTML-based forms technology. This technology is divided into three key components: HTML markup for the forms, a server-side Internet Server API (ISAPI) filter, and a forms registry.

To create Web Storage System forms, you need to add some special markup to your HTML-based forms, indicating to the Web Storage System that the fields you're requesting in your form should be pulled from your Web Storage System application. We'll drill into what this markup actually consists of when we look at the Web Storage System form used in the Training application.

Beyond client-side markup, the Web Storage System forms technology also includes a server-side ISAPI extension. This extension captures requests sent to your Web server and checks to see whether the Web browser is requesting an item that

has an associated Web Storage System form. If the item does have an associated Web Storage System form, the ISAPI extension finds that form, and returns the form to the Web browser. Since Web Storage System forms support both standard HTML 3.2 browsers and XML-aware browsers, data binding can occur in one of two places: on the server or on the client. For HTML 3.2 clients, you'll want to do your data binding on the server. This forces the Web Storage System to pull the values for the fields in the form, perform the data binding on the server side, and return only the form's HTML 3.2 representation.

For XML-aware browsers, such as Microsoft Internet Explorer 5, you can perform server-side or client-side data binding. If you perform client-side data binding, the forms engine will return the form and you can perform the data binding on the client side rather than on the server side. This allows you to provide easier manipulation of the data without having to make an extra round-trip to the server each time. For example, you could re-sort or change the format of the data representation to suit your application's needs without incurring the cost of returning to the server to do this.

The final component of the Web Storage System forms technology is the Web Storage System forms registry. By allowing you to specify which forms should appear for specific items, the Web Storage System enables you to customize the default renderings of items. Each built-in item has some default forms that will be rendered for it. Since the forms registry is flexible, you can register forms based on the browser type requesting the item, the language of the browser client, or the type of item that the user requests from the Web Storage System. This flexibility allows you to build different forms for different clients. For example, you could create a microbrowser form for cell phone clients browsing your application using the Microsoft microbrowser technology. We'll look at the forms registry in more detail when we examine the Training application.

Web Storage System Events

Exchange Server 5.5 introduced an event mechanism that allowed application developers to write code to handle events occurring in the Exchange Server database. The Web Storage System further improves on this concept. In Exchange Server 5.5, events are asynchronous, meaning that the event fires after the item is committed to the database. The Web Storage System also supports asynchronous events; however, it supports synchronous and system events too. Synchronous events fire before the item is committed to the database, enabling your application to decide whether the item should be committed or aborted (in which case the item won't be saved). Synchronous events guarantee that the application is the only process making this decision for the item. Users or other processes are blocked until the application finishes processing. The system events notify the applications about key occurrences in the Web

Storage System—for example, a system event might fire after the Web Storage System starts up. Developers could write code to either begin replication or start processing their custom application when this event occurs. These two new classes of events allow you to build even richer applications on the Web Storage System.

Workflow Support

A workflow application is an excellent example of an application you can build with Web Storage System events. The Web Storage System ships with a built-in workflow engine that uses synchronous and system events to perform its functionality. This built-in workflow support enables developers to start writing workflow applications as soon as they obtain a product containing the Web Storage System.

XML Support

The Web Storage System is very Web-centric, which explains how it got its name. The Web Storage System natively supports XML, and you can use it to retrieve and set data. We'll look at the XML support of the Web Storage System more closely in the Training application.

Security Features

Since information security is always a major concern for developers and users, the Web Storage System supports securing data at both the item level and the property level. This allows you to select which users or groups of users can access data contained in the Web Storage System. Furthermore, you can query or modify this access programmatically.

Additional Features

Besides the standard Web Storage System features, Exchange Server 2000 provides some additional features in its implementation of the Web Storage System technology. These features include Messaging Application Programming Interface (MAPI) support, multiple top-level hierarchies (TLH), and a set of management objects that allow you to programmatically manage information in Active Directory and the Exchange Server 2000 Web Storage System. Let's take a look at each of these features.

MAPI Support

Exchange Server 2000 continues to fully support the MAPI interfaces. This means clients that use MAPI, such as Microsoft Outlook 2000, will run against the Web Storage System without modification. As an application developer, you should realize that the applications you have written to MAPI will continue to work. As always, you should nevertheless test your applications to ensure that all functionality continues to behave as expected.

Multiple Top-Level Hierarchies

You might be wondering what a top-level hierarchy, or TLH, is. A top-level hierarchy is simply a tree of folders that has a top-level root folder. So the Public Folder hierarchy that starts with Public Folders and continues to All Public Folders is a top-level hierarchy. To make application development easier, Microsoft has enabled you to have TLHs besides the Public Folder one in your Exchange Server environment. This support allows you to break your applications into multiple hierarchies so that users can avoid crawling through the Public Folder tree to find the desired application. This change also makes it easier for administrators to manage your applications, because they can separate the applications into independent TLHs. You should, when possible, place your applications in a top-level hierarchy other than the Public Folders hierarchy because other top-level hierarchies provide more functionality. However, if you require Outlook access to your application, you will need to keep your application in the Public Folder hierarchy. Outlook cannot view hierarchies other than the Public Folder hierarchy.

Figure 18-1 shows the Exchange System Manager (which replaces the Exchange Administration program) displaying multiple TLHs on a single Exchange server. Notice how the multiple TLHs also allow developers to place applications into different naming contexts.

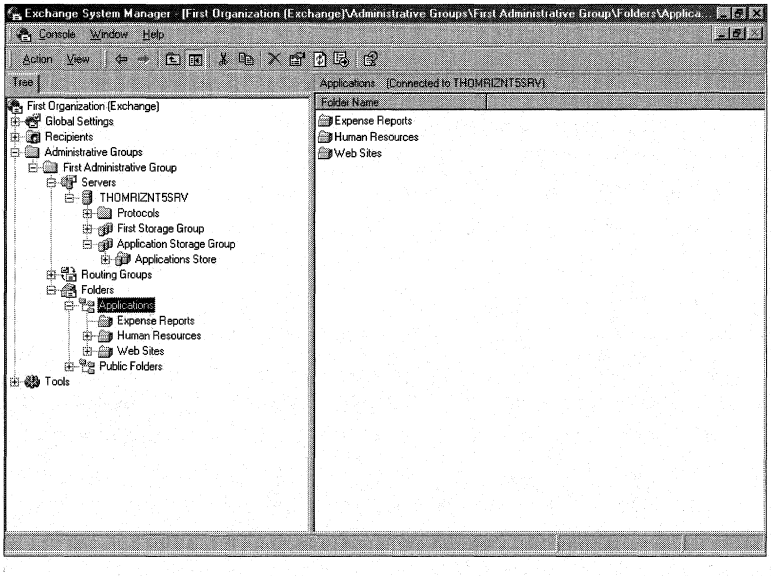


Figure 18-1. *The Exchange System Manager with multiple TLH support. Multiple TLHs greatly benefit application developers.*

CDO for Exchange Management

To provide a programmatic way for developers to manage recipients and servers, the

new version of CDO in Exchange Server 2000 has been extended with a set of management objects called CDO for Exchange Management Objects (EMO). With EMO, you can create, modify, or delete recipients or groups of recipients. Furthermore, with EMO you can manage the structure of the information stored in Exchange Server 2000. Notice I said *structure* and not *content*. EMO doesn't give you the ability to create items, change properties, or perform other such content-related tasks. Instead, you should use CDO and ADO to attain such functionality. EMO is there to help you create storage groups and folder hierarchies, as well as change storage quotas and deleted-item recovery periods.

You'll want to use EMO where appropriate in your applications. Furthermore, if you plan to write any administration components for Exchange Server 2000, such as Microsoft Management Console (MMC) snap-ins, you'll need to learn about and use the EMO object model. We won't discuss EMO any further in this chapter because its interfaces are so straightforward. However, I will point you to the Exchange Server Software Development Kit (SDK), included on the companion CD, so you can learn more about it.

Outlook Web Access

The Web Storage System in Exchange Server 2000 directly supports a new version of OWA. OWA is no longer based on Active Server Pages (ASP) as in previous versions. Instead, the OWA code is directly compiled into the Exchange Server, and the HTML files are generated by the server instead of by interpreted script running on IIS.

There are also a number of enhancements to OWA that you should be aware of. The most significant one is that different versions of OWA exist for different browsers. For non–Internet Explorer 5 browsers, OWA uses standard HTML 3.2 to render information from Exchange Server, as shown in Figure 18-2. In this case, all HTML rendering and data binding take place on the server side.

If you are using Internet Explorer 5, OWA and the Exchange Server are smart. Instead of forcing all the rendering and data binding to occur on the server, OWA takes advantage of the XML and WebDAV support in Internet Explorer 5 to offload processing and rendering to the browser. OWA will push down Exchange Server data as XML and will utilize Internet Explorer to format the XML data by using extensible style sheets (XSL). If the user needs a different set of data, OWA will request only the data from the server and won't have to reload the entire page in the browser. By doing this, the Internet Explorer 5 version of OWA saves you round-trips to the server to re-sort or group the existing XML data on the client. Furthermore, if data is needed from the server, OWA is efficient at retrieving just that data rather than the entire page. Figure 18-3 shows the new version of OWA. Later in this chapter, you'll see how to extend OWA by using URL parameters and Web Storage System forms.

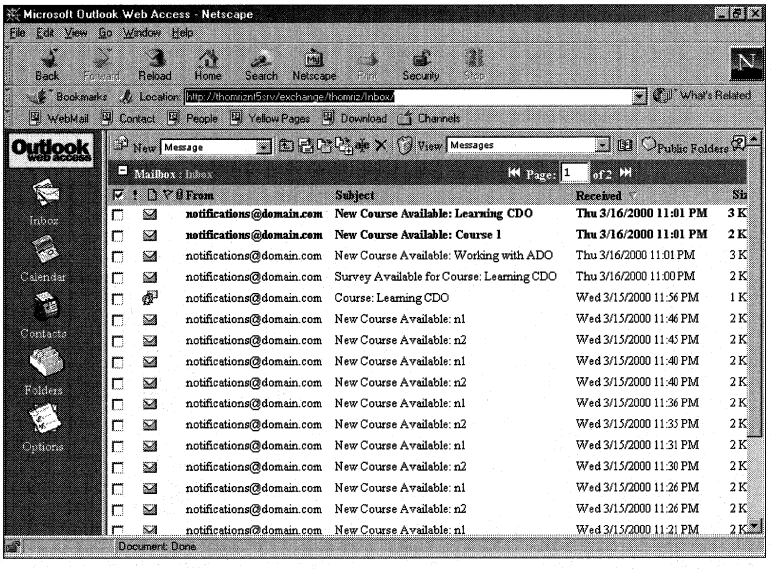


Figure 18-2. *The Netscape Navigator version of Outlook Web Access.*

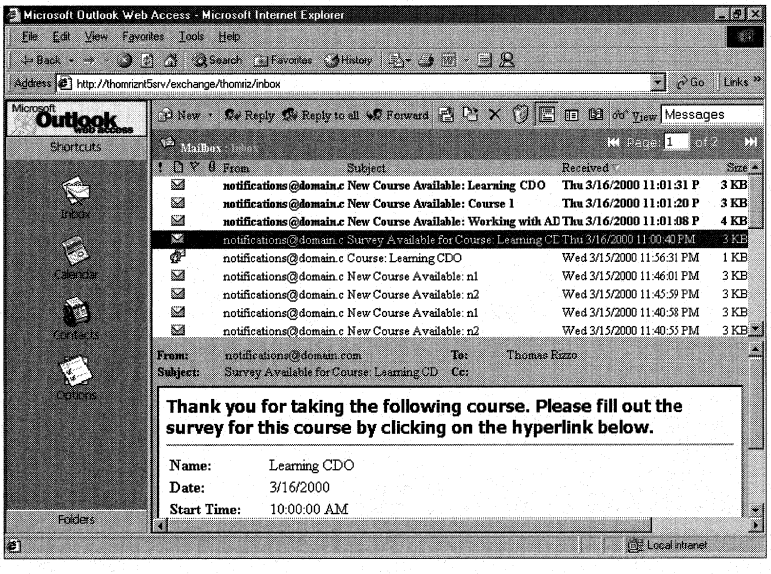


Figure 18-3. *The Internet Explorer 5.0 version of OWA. This version supports XML and WebDAV.*

WHAT ABOUT EXCHANGE SERVER 5.5 APPLICATIONS?

You might be wondering what will happen to the existing Exchange Server 5.5 applications you've written using MAPI, CDO, ADSI, or even the Event Scripting Agent. Have no fear—those applications should continue to run without modification, except for some cases that I'll describe here.

First, MAPI, CDO 1.21, and the Event Scripting Agent are all supported by Exchange Server 2000. Microsoft even supports running the Exchange Server 5.5 version of OWA against Exchange Server 2000. This demonstrates that CDO 1.21 works fine with Exchange Server 2000. In terms of MAPI support, Outlook 2000 uses MAPI and since Outlook continues to work with Exchange Server 2000, it shows how MAPI still works as well.

If you've written applications that use the Directory API (DAPI), you'll need to rewrite your code to use ADSI since Active Directory replaces the Exchange Server directory in Exchange Server 2000. Furthermore, if you've written administration extensions to the Exchange Server administration program, you'll need to rewrite these extensions to the MMC since this is the way you administer Exchange servers.

The major hurdle of moving your Exchange Server 5.5 applications to Exchange Server 2000 is the conceptual differences between the two versions of Exchange Server. While you can continue to run your Exchange Server 5.5 applications as is on Exchange Server 2000, you'll at least have to look at the features of Exchange Server 2000 to see where you can enhance your existing applications. For example, if you have an Outlook forms-based application, you might want to extend it using the new server events in Exchange Server 2000 to add workflow or other functionality. Throughout the rest of this chapter, we'll look at the new features Exchange Server 2000 offers developers. If you have Exchange Server 5.5 applications, as you read this information, look for ways to enhance or migrate existing parts of your applications to Exchange Server 2000.

THE TRAINING APPLICATION

The best way to understand the development capabilities of Exchange Server 2000 is to see an application that takes advantage of the product's new capabilities. That's why I designed the Training application. This application is pretty complex; it contains ASP, XSL, server events, and workflow components. This sample application manages an internal training program site. It also provides a Web interface via which students and instructors can register for, critique, and discuss training courses and materials. I've created a setup program for this application, which will make it easier

for you to get this application started and will show you how to perform some administrative functions for Exchange Server 2000. The setup program will also show you how to perform COM+, IIS, and Active Directory administrative tasks.

Before examining the code and technologies used throughout the Training application, let's discuss exactly what this application does. That way, when we cover each of the application's implementation sections, you'll understand what each implementation actually does within the context of the rest of the application.

Setting Up the Training Application

When you launch the setup for the Training application, the first thing you'll see is the Microsoft Visual Basic setup program I've created. Figure 18-4 shows the user interface for the setup program.

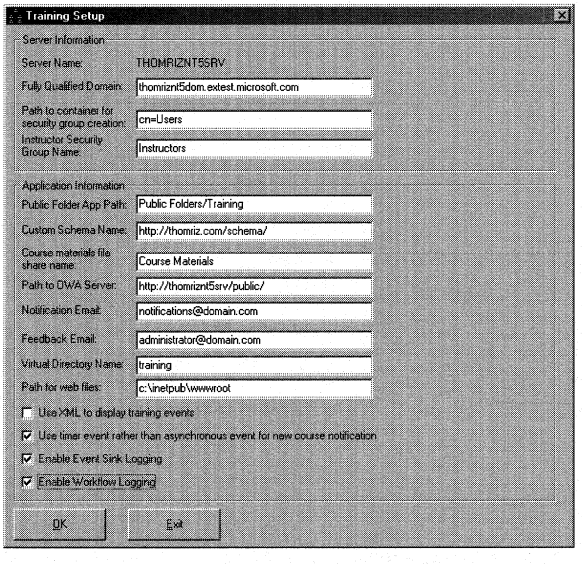


Figure 18-4. *The user interface for the setup program. It requires you to fill in information about where you want the Training application to be placed.*

The setup program takes the information you provide in your setup interface and sets up the Training application, so you don't have to perform any extra steps. The setup program performs the following steps:

1. Establishes an ADO Connection to the Exchange Server so that the Connection object can be used throughout the setup and certain parts of the setup can be transacted and rolled back.
2. Creates the Exchange Server folders.
3. Creates the custom schema, including custom content classes and properties.

4. Creates messages in the Emails folder, which contains the HTML templates for the notification e-mails sent by the application.
5. Prompts you for the categories you wish to create for your training events and then stores those categories in the Configuration folder.
6. Creates a security group in Active Directory that will contain the users who are instructors for the application.
7. Creates the IIS virtual directory and copies the Web files to it.
8. Creates a Windows 2000 file share for the course materials.
9. Copies and registers the event dynamic link libraries (DLL).
10. Registers the event DLL files as a COM+ application.
11. Creates the event registrations for the application that will handle new course notification, survey notification, and survey result compilation.
12. Imports, registers, and enables the workflow process definition.
13. Creates the event registrations for the workflow process.

Although the setup program is quite extensive in what it provides, I won't detail here the non–Exchange Server or noncritical application deployment steps, such as creating virtual directories in IIS using ADSI. To learn more about such steps, look at the source code for the setup program, which you'll find on the companion CD. We'll look closely at the code behind the setup program throughout this chapter.

> **NOTE** You will need to run the setup program of the Training application directly on your Exchange server. You will also need to be an administrator of both the Windows 2000 and Exchange Server systems.

Using the Training Application

Once you're done running the setup program, you can start using the Training application. The application uses a series of Public Folders that store all the application's information. Figure 18-5 shows the folder hierarchy of the Training application. As you can see, the different types of folders in the application range from standard message folders to contact folders and calendar folders.

The classes are contained in the Schedule folder, while student and instructor information are contained in their respective contact folders. The interface of the application is the default training page, which is different for instructors and for students. The instructor home page is shown in Figure 18-6.

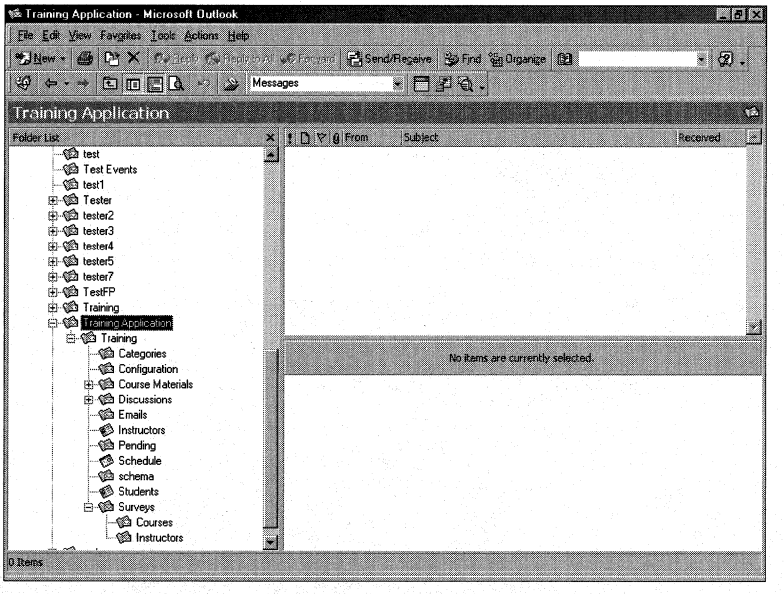


Figure 18-5. *The folder hierarchy for the Training application as it appears in Microsoft Outlook.*

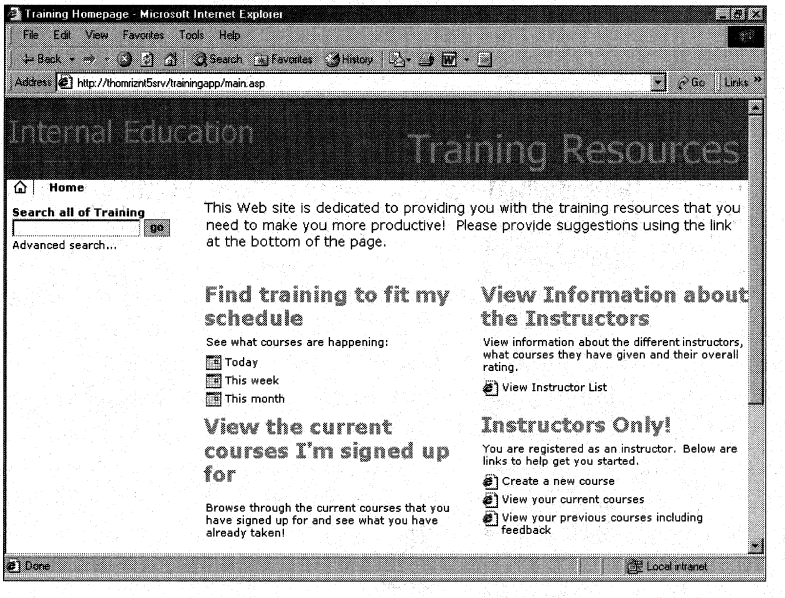


Figure 18-6. *The Training application home page for the sample application as viewed by an instructor.*

From the home page, you can retrieve the schedule of classes; view information about instructors; change your notification preferences; or if you are an instructor, create a new course. The application determines whether you are an instructor by using Windows 2000 Security Groups. The application searches the security group you specify in the setup program and checks to see whether the user accessing the application is a member of that security group. If the user is a member of that security group, the instructor-specific content appears on the Web page. Figure 18-7 shows the Active Directory Users And Computers snap-in with the Instructors security group open.

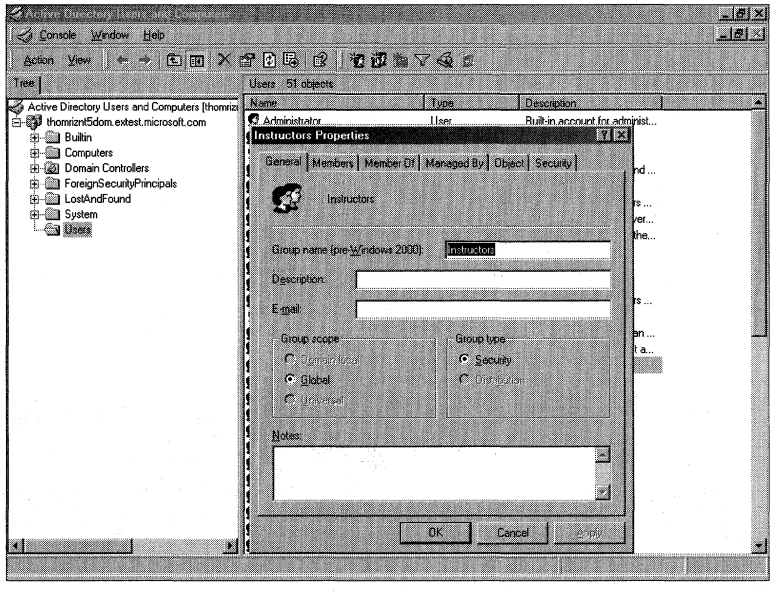


Figure 18-7. *The Instructors security group controls access to instructor-specific functions.*

One thing to note is that the Training application creates its folders in the Public Folder hierarchy. I coded the application this way so that you could see the folders in Outlook. However, if I was to really deploy this application, I would not create the folders in the Public Folder hierarchy because it's Web-based. Instead, I would create a new top-level hierarchy, not visible by Outlook, to contain my application. There are a number of reasons to create separate top-level hierarchies for your applications, including the ability to conduct deep searches of the folders using ADO, and not providing access to the application from Outlook.

Creating a Course

The application allows you to create courses and register students for them. When an instructor creates a course, the application asks the instructor whether she wants to create a file share for course materials or a discussion group for the course. This functionality demonstrates how you can utilize the IFS components of the Web Storage System through the file share capabilities and also illustrates the reusability of OWA. Figure 18-8 shows a course listing that contains links to both the course materials and a discussion group for a particular course.

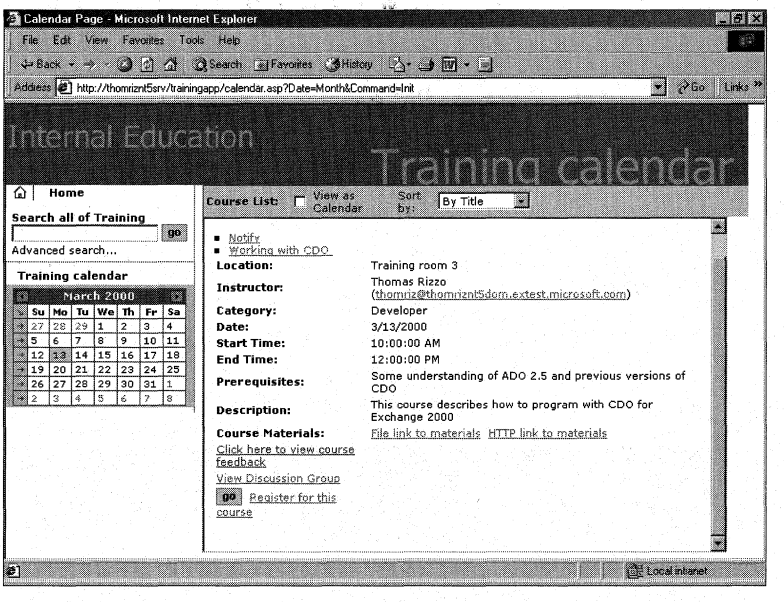


Figure 18-8. *A course listing that shows how you can utilize both IFS and OWA extensibility in your Exchange Server 2000 application.*

After an instructor creates a course, an asynchronous or timer event (depending on which you specified in the setup program), fires in the Schedule folder. This event checks to see whether any students have asked to be notified when new training is available in the specific course category. As the application administrator, you must specify these categories—for example, *Developer*, *End User*, and *IT*. If the application locates students who need to be notified about the training, it sends the students an HTML-formatted message, as shown in Figure 18-9.

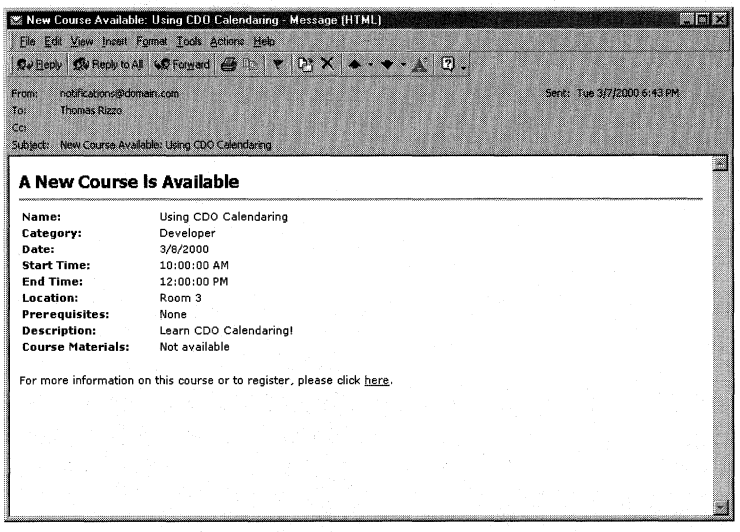


Figure 18-9. *An HTML-formatted message sent to students who want to be notified about new training events.*

The notification preferences of each student are stored on their respective contact record in the Students folder. Students can change their preferences for notification through the application's Web interface, as shown in Figure 18-10.

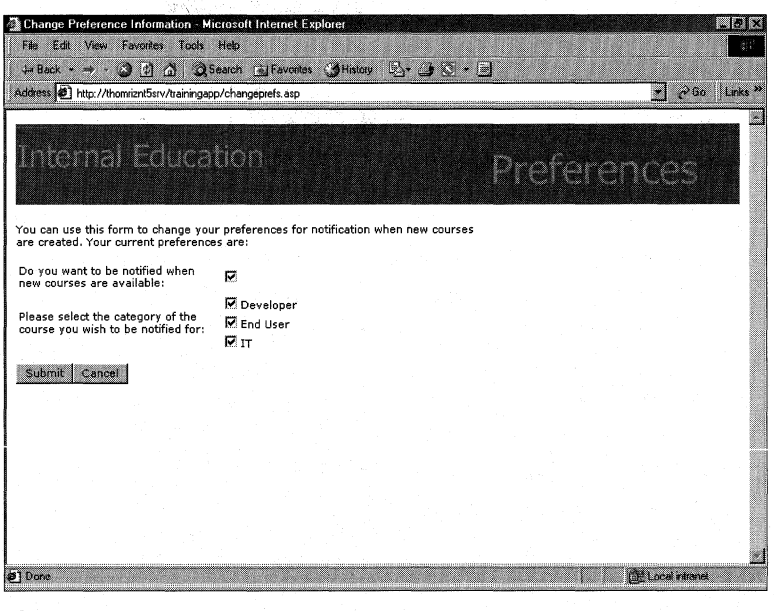


Figure 18-10. *The Web page for changing student notifications.*

Registering for a Course

You might be wondering how a student goes about registering for a class. The application provides two ways for students to browse through the available courses in the Schedule folder. First, a student can specify date ranges from a Date Picker and view the courses in a simple list. The student can then re-sort the list by title, date, or category. The page containing this simple list is shown in Figure 18-11.

> **NOTE** There are two ways you can generate this simple list page. One way is to use ADO inside ASP pages. The other technique involves using the XMLHTTP component of Internet Explorer 5 and requesting XML data from the Exchange server. The Training application then renders that XML data locally, thereby eliminating a round-trip to the server if the user wants to re-sort the list of classes.

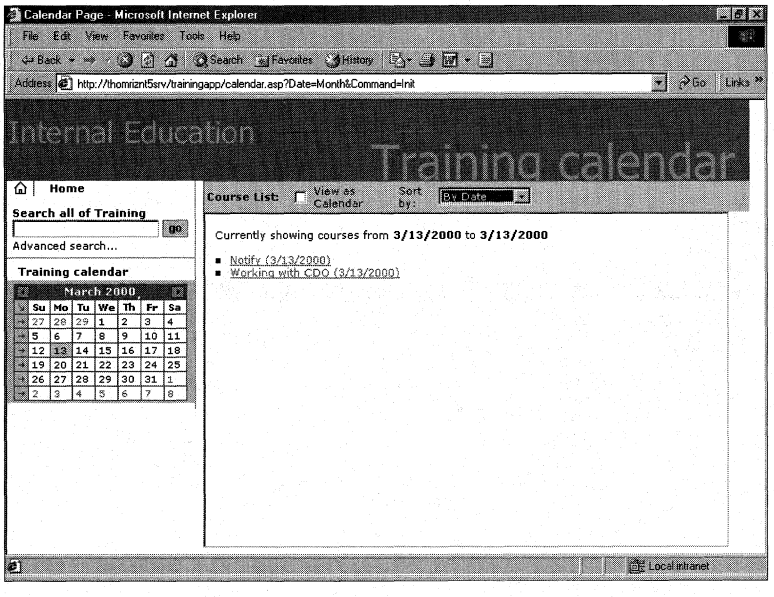


Figure 18-11. *The page showing a simple list of courses for a specific date range.*

The second way the Training application allows students to browse through the available courses is by leveraging the extensibility of OWA. By passing OWA-specific parameters (which you'll learn about later in this chapter), you can make OWA display information. Figure 18-12 shows how the Training application employs the rich calendar views of OWA to display the schedule of courses.

When a user double-clicks on one of the calendar items in the view, the application displays a custom Web Storage System form instead of displaying the standard OWA appointment form. The ability to replace OWA forms with your own is a powerful one. Figure 18-13 shows the Web Storage System form displayed when a user clicks on a training event in the calendar.

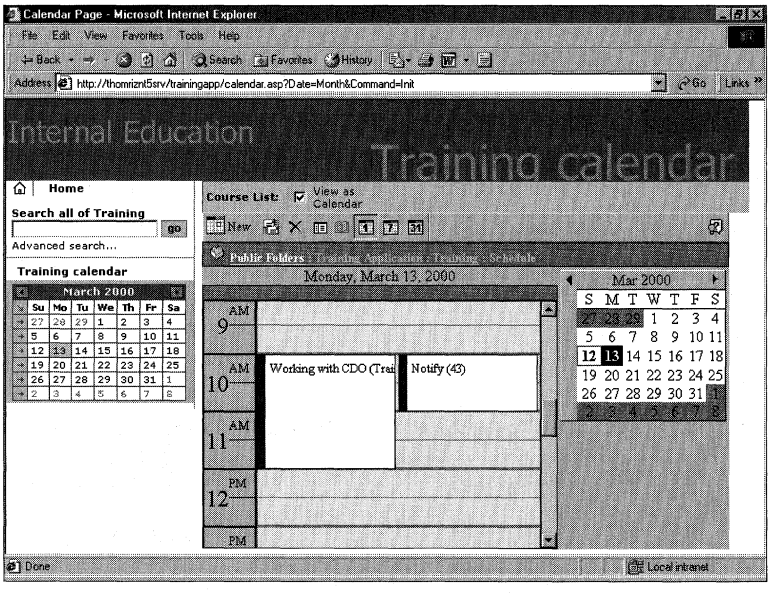


Figure 18-12. *Using OWA and the calendar view to display a list of available courses.*

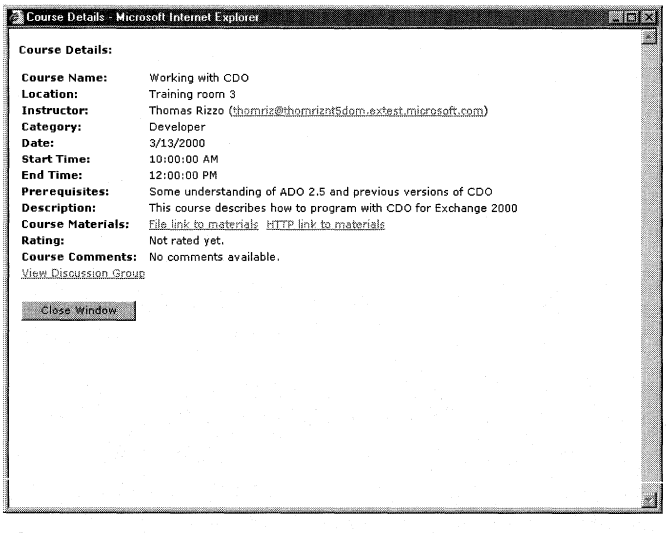


Figure 18-13. *The Web Storage System form that is displayed when a user clicks on an item in the Training application's calendar.*

Searching for a Course

The Training application also offers students and instructors two ways to quickly find course information: via quick search, and via advanced search. Both types of search eventually follow the same code path, but the advanced search provides the user with a more powerful interface for specifying search options. Figure 18-14 shows the Advanced Search page of the Training application. The search capabilities can take advantage of the built-in content indexing of Exchange Server 2000, if you enable content indexing. We'll learn about content indexing and how to leverage it in Chapter 19.

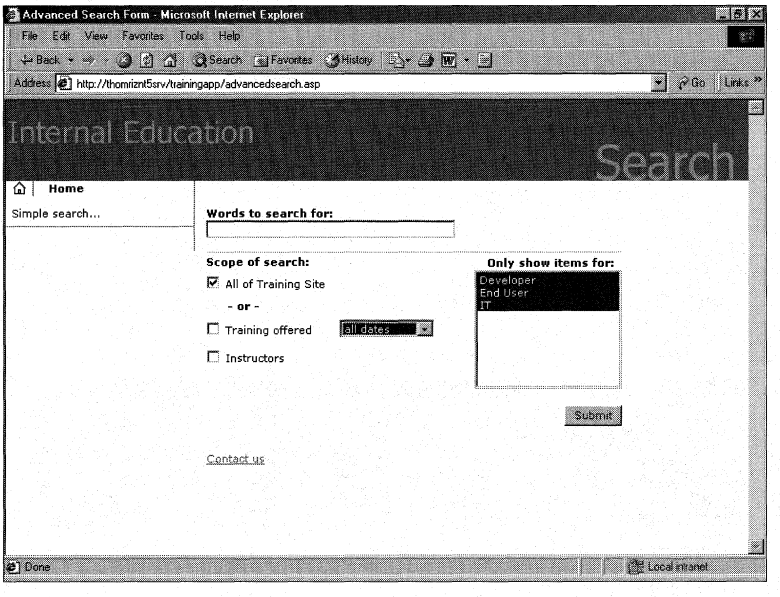


Figure 18-14. *The Advanced Search page for the Training application.*

Using a Workflow Process for obtaining approvals

The Training application utilizes the built-in workflow engine and the graphical Workflow Designer of Exchange Server 2000. If an instructor specifies that a course requires approval, the application starts a workflow process when a student attempts to register for the course. The application next sends an e-mail to the student's manager. The manager can then approve the student to take the course. If the manager rejects the request or doesn't approve it in time, the student can't take the course.

Figure 18-15 shows the workflow process in the Workflow Designer for Exchange Server 2000. Figure 18-16 shows the e-mail that the manager receives when an approval is required for a student to take a course.

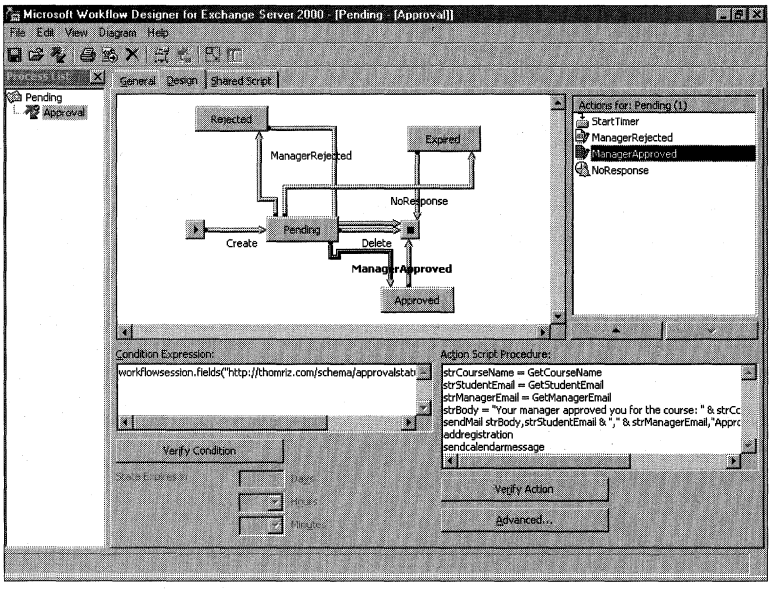


Figure 18-15. *The workflow process shown in the Workflow Designer for Exchange Server 2000.*

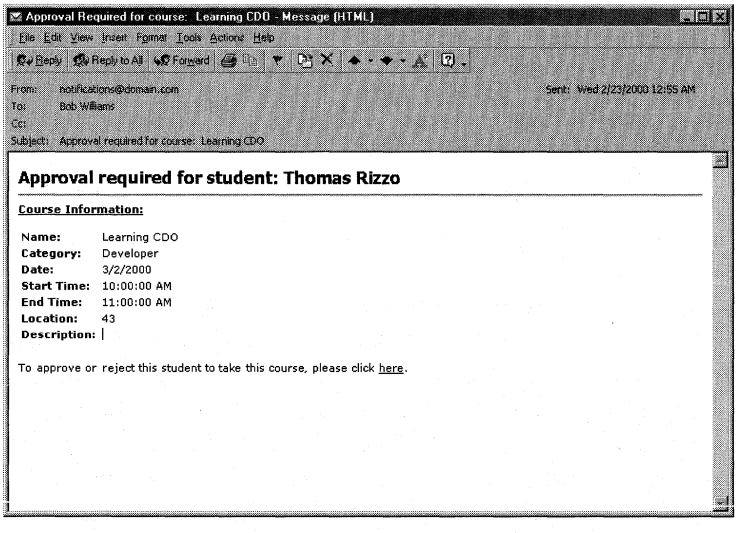


Figure 18-16. *The e-mail sent to the manager who needs to approve a student who wants to take a course.*

Providing a Survey

The Training application also provides a survey component. A timer-based event fires on the Exchange server every night to implement the course survey component. This event checks to see whether any courses have been completed on that day. If a course

has been completed, the timer agent e-mails a notice to the students who were registered for the course, as shown in Figure 18-17. The students can then click on a link in the e-mail message and fill out a survey to rate both the course and the instructor. The application checks to make sure that students don't fill out multiple surveys for either the course or the instructor. Figure 18-18 shows a survey form for a course.

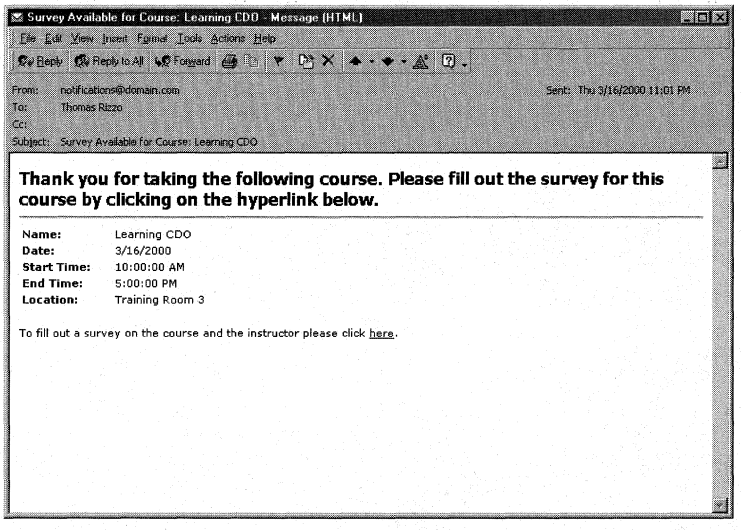


Figure 18-17. *The e-mail notice that a survey is available for a course.*

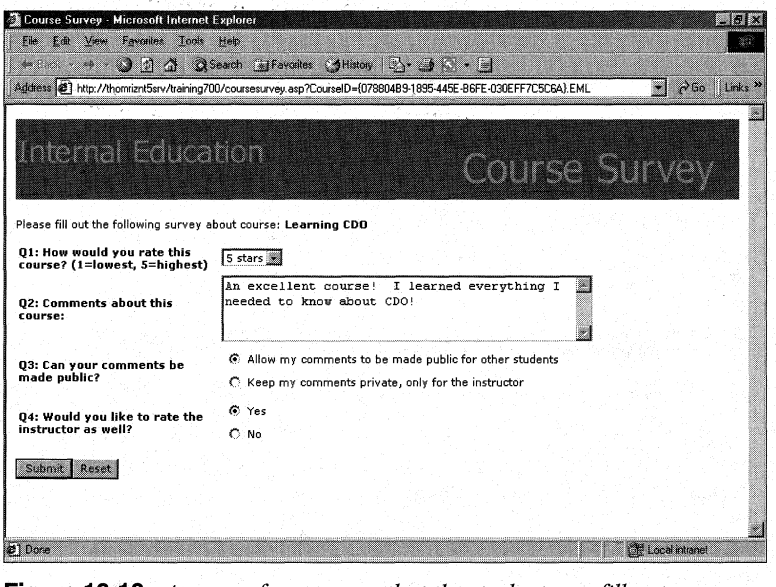


Figure 18-18. *A survey for a course that the student can fill out.*

Once the student fills out the survey, the survey is saved into one of the Surveys folders, depending on whether the survey is a course or instructor survey and on whether another agent fires. This agent collates all surveys received for the course and the instructor and determines an overall rating for each. The agent also takes any comments the users have and adds them to the course or the instructor rating. Figure 18-19 shows how the final results look after the agent completes its processing. Other students can then view ratings and comments about either the instructor or the course.

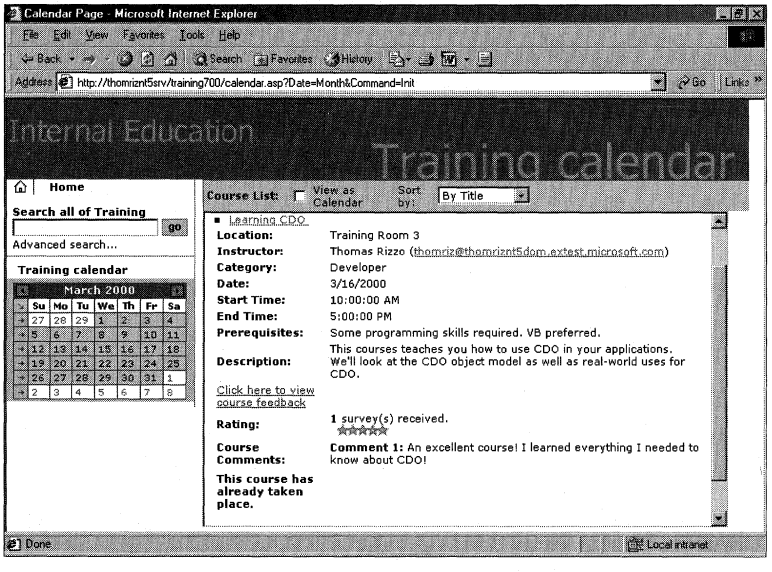


Figure 18-19. *Ratings and comments are shown in the main views for both the instructor and the course. Here, we see the ratings for a course.*

Now that you've had a quick overview of some of the features of the Training application, let's take a look at how the Training application was actually built. We'll look at the technologies used in our programming and at some code snippets from both the setup program and the application itself.

THE WONDERFUL WORLD OF SCHEMAS

Your first consideration when building an Exchange Server 2000 application is whether you need to use the schema support that's built into the product. As mentioned earlier in the chapter, Exchange Server 2000 allows you to create schemas for your applications—meaning you can define classes of items, such as *myproject* or *mycontact*. On those classes of items, you can have default sets of properties, such as *myname* or *mylocation*.

You don't have to create schemas in order to use built-in or custom properties on items in Exchange Server 2000. For example, you could append new properties to an item using ADO and store that item in Exchange Server. You'll need to create a custom schema if you want to reuse your properties across multiple items, rather than with a single, one-off item. Also, if you want your users to easily discover your properties, you'll need to use a schema. This discoverability can be achieved through ADO when you run a SQL SELECT statement against your application, or it can be with the Web Storage System forms you create. Since the Training application uses many custom properties across many items, it creates schemas in its setup program. Let's discuss this part of the program now.

Overview of the Exchange Server Schemas

You need to know a few things before you attempt to create schemas. First, Exchange Server 2000 ships with some schemas already in place for the default items it understands, such as messages, documents, appointments, and contacts. The schemas for these items are stored in a hidden folder in your Exchange server. You can view this hidden folder either by writing a simple program or by looking at the folder through the Mdbvue utility. Figure 18-20 shows how to view the Schema folder using MDBVUE.

> **NOTE** You'll find the Schema folder under the Public Folders tree, in the non-ipm subtree. This folder contains XML files that define the Exchange Server 2000 schemas. Exchange Server 2000 does support defining schema through XML. However, in the setup program for the Training application, I used ADO to create my schema definitions.

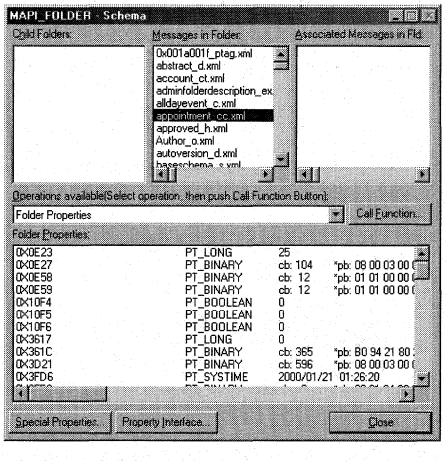


Figure 18-20. *MDBVUE browsing the hidden Schema folder for Exchange Server 2000.*

Second, as part of the base Exchange Server schema, there is the idea of a content class. For example, a message has a content class of *urn:content-classes:message*.

Content classes are groupings of properties that define a certain type of item. If you set the content class of a particular item to *message*, the item receives all the properties from your schema that are associated with the *message* content class.

If you're coming from the Exchange Server 5.5 development environment, you might be wondering how content class and message class correspond. There really is no explicit relationship between content class and message class; you can have a content class that's entirely independent from a message class. If you need to support Outlook or previous versions of OWA, just be sure to explicitly set both the content class and the message class in your application. Don't worry about the custom properties in your content class. Exchange Server makes those available in the PS_PUBLIC_STRINGS name space in MAPI so that Outlook or any MAPI application can retrieve those custom properties.

> **NOTE** Outlook supports field names of up to 32 characters in length in its Field Chooser. So if you create a property with a name longer than 32 characters, you might not be able to retrieve it from Outlook.

Creating Custom Content Classes

The first step when creating a custom content class is to decide what to call it. Make sure that the name you select for your content class doesn't clash with other content class definitions. Once you determine the name of your content class, you'll want to think about the properties contained in it. Since Exchange Server supports a wide range of built-in content classes, it allows you to inherit your content class from an existing one. For example, the Training application has training events. These events require properties that already exist on the appointment content class. However, the training events also need some extra properties that will exist on all training event items. Using inheritance, the Training application inherits the properties from both the appointment content class and the item content class.

Exchange Server 2000 supports both single and multiple inheritance for content classes. The following list contains all the default content classes supported by Exchange Server 2000:

- *urn:content-classes:appointment*
- *urn:content-classes:calendarfolder*
- *urn:content-classes:calendarmessage*
- *urn:content-classes:contactfolder*
- *urn:content-classes:contentclassdef*
- *urn:content-classes:document*
- *urn:content-classes:dsn*

- *urn:content-classes:folder*
- *urn:content-classes:freebusy*
- *urn:content-classes:item*
- *urn:content-classes:journalfolder*
- *urn:content-classes:mailfolder*
- *urn:content-classes:mdn*
- *urn:content-classes:message*
- *urn:content-classes:notefolder*
- *urn:content-classes:object*
- *urn:content-classes:person*
- *urn:content-classes:propertydef*
- *urn:content-classes:recallmessage*
- *urn:content-classes:recallreport*
- *urn:content-classes:taskfolder*

Once you figure out which content classes you want to inherit from, you'll need to create some property definitions for your content class. Property definitions describe the name, type, value, and special characteristics of the properties you want in your content class. Be aware that when you do create a property definition, it can be used across multiple content classes. You'll see this in the Training application, where both instructors and training events share survey result property definitions.

Let's look at some code from the setup program that creates the property definitions for a training event. We'll then step through exactly what's happening in the code.

```
CreateSchemaPropDef "instructoremail", "string", False, True, _
   False, CStr("")
CreateSchemaPropDef "prereqs", "string", False, True, False, CStr("")
CreateSchemaPropDef "seats", "string", False, True, False, CStr("")
CreateSchemaPropDef "authorization", "string", False, True, False, _
   CStr("")
CreateSchemaPropDef "category", "string", False, True, False, CStr("")
CreateSchemaPropDef "surveycount", "string", False, True, False, CStr("")
CreateSchemaPropDef "discussionurl", "string", False, True, False, _
   CStr("")
CreateSchemaPropDef "materialsfilepath", "string", False, True, _
   False, CStr("")
CreateSchemaPropDef "materialshttppath", "string", False, True, _
   False, CStr("")
```

(continued)

```
Private Sub CreateSchemaPropDef(strName, strType, bMultiValued, _
   bIndexed, bReadOnly, varDefaultValue)
   Dim rec As New ADODB.Record
   With rec
      .Open strAppSchemaFolder & "/" & strName, , adModeReadWrite, _
         adCreateNonCollection + adCreateOverwrite

         'Create new property definition
         .Fields("DAV:contentclass") = "urn:content-classes:propertydef"

         'Give it a name
         .Fields("urn:schemas-microsoft-com:xml-data#name").Value = _
            strSchema & strName

         'Set the data type for the property
            .Fields("urn:schemas-microsoft-com:datatypes#type").Value = _
               strType

         'Set the other fields for the property
            .Fields("urn:schemas-microsoft-com:" & _
               "exch-data:ismultivalued").Value = bMultiValued
            .Fields("urn:schemas-microsoft-com:" & _
               "exch-data:isindexed").Value = bIndexed
            .Fields("urn:schemas-microsoft-com:" & _
              "exch-data:isreadonly").Value = bReadOnly
            .Fields("urn:schemas-microsoft-com:" & _
              "exch-data:default").Value = varDefaultValue
            .Fields.Update
            .Close
        End With
End Sub
```

This code creates a new item for my property definition in the Schema folder for the Training application. My recommendation is that, unless absolutely necessary, you don't modify the built-in schema for Exchange Server 2000. Instead, create a Schema subfolder for your applications, and place your property and content class definitions in it. You can inherit from the built-in schemas in Exchange Server 2000 rather than modifying them.

> **NOTE** You need to create property definition names in a unique name space. The setup program asks you what name space you want to use. By default, you should include your Internet domain name, if you have one, since it's guaranteed to be unique. For example, a good name space might be *http://yourdomainname/schema/*. You can then append to that name space the names of the custom properties for your application.
>
> You should also be aware that property names are case sensitive. I recommend lowercasing your entire property name to make your coding easier.

Once the new item is created, the code sets the content class property in the DAV name space to *urn:content-classes:propertydef*. Exchange Server 2000 defines a number of new name spaces for properties, such as DAV, *http://schemas.microsoft.com/exchange/*, or *urn:schemas-microsoft-com:office:office*. The code then sets the *urn:schemas-microsoft-com:xml-data#name* property to the fully qualified name of the property. Next, the code sets the data type, which can be one of a number of different values. The most common ones that you'll use are String, Float, Boolean, DateTime, i2, i4, i8, Uuid, Int, and Bin.base64. I've used strings throughout the Training application to shield me from changes in the underlying builds of Exchange Server. This is definitely not a best practice; after Exchange Server 2000 is released, I'll update the code to use the correct data type for the properties. You should always use the correct data type depending on the needs of your application.

Once the data type is set, you can set other values for your properties. One such value is the *ismultivalued* property. By setting this property to *True*, you tell the Web Storage System that this property supports multiple values of the data type you specified. When working with multivalued properties, you'll want to set and get the values using arrays. Examples of built-in multivalued properties are *urn:schemas-microsoft-com:xml-data#element* and *urn:schemas-microsoft-com:xml-data#extends*, which are properties on the content class definition item. We'll discuss creating a content class definition item in a little bit.

Another property you can set is the *isindexed* property. This property has nothing to do with content indexing (which we'll look at later); instead, it corresponds to the Exchange Server database index. If you set this property to *True*, Exchange Server will create an index for the property so that sorting, grouping, and other operations on it execute quickly. One caveat for using *isindexed*: if you are concerned about load on your Exchange server, don't set this property to *True* for all your properties. Instead, just determine which properties you really need indexing on, and set this property to *True* for those properties. To create the index for your custom properties, you will have to issue the CREATE INDEX command in ADO. We'll learn more about this command and ADO access to Exchange Server 2000 later in this chapter.

The next property you can set is the *isreadonly* property. You don't need to set this property because only properties provided by the Web Storage System can be set to read-only. However, I threw this property in here just to expose you to it. You can query the value of this property from your applications to see whether a Web Storage System property is read-only.

The final property that you'll want to set is the *default* property. This property specifies what the default value, if any, should be for your property when it's created by the application or user. Although you don't have to set this property, you will want to do so in order to initialize your properties to a default value.

Once the code finishes setting these properties using ADO, a call to the *Update* method of the Fields collection is required. When we talk about ADO support in

Exchange Server 2000 later in the chapter, we'll discuss the use of transactions with ADO and Exchange Server. I highly recommend that you use transactions where appropriate in your application. For example, you should use transactions in case you receive an error while processing your ADO commands. This allows you to roll back the transaction rather than leaving your application's data in an inconsistent state. Although the previous code segment doesn't show it, the entire schema creation process is wrapped in a local OLE DB transaction.

Creating Content Class Definition Items

Now that we've created the custom property definitions for our content class, we need to create a content class definition in our schema. To do this, we must create an item in the Schema folder that has the correct content class and properties set so that Exchange Server knows we want to create a custom content class. The following code shows how to create a definition of a custom content class:

```
arrTrainingEventProps = BuildSchemaArray(Array("instructoremail", _
   "prereqs", "seats", "authorization", "category", "surveycount", _
   "discussionurl", "materialsfilepath", "materialshttppath", _
   "publiccomments", "overallscore"))
CreateContentClass "trainingevent", _
   "urn:content-classes:trainingevent", _
   Array("urn:content-classes:item", "urn:content-classes:appointment"), _
   arrTrainingEventProps

Private Function BuildSchemaArray(arrProperties)
   Dim tmpArray()
   For i = LBound(arrProperties) To UBound(arrProperties)
      ReDim Preserve tmpArray(i)
      tmpArray(i) = strSchema & arrProperties(i)
   Next
   BuildSchemaArray = tmpArray
End Function

Private Sub CreateContentClass(strItemName, strContentClassName, _
   arrExtends, arrFields)
   Dim rec As New ADODB.Record
   With rec
      'Create item in application Schema folder
      .Open strAppSchemaFolder & "/" & strItemName, , adModeReadWrite, _
         adCreateNonCollection + adCreateOverwrite
      '
      'Create new content class definition
      '
      .Fields("DAV:contentclass") = "urn:content-classes:contentclassdef"
```

```
        'Name the content class
        .Fields("urn:schemas-microsoft-com:xml-data#name").Value = _
           strContentClassName

        'Enter what other content classes this one extends
        .Fields("urn:schemas-microsoft-com:xml-data#extends").Value = _
           arrExtends

        'Enter the fields that exist on the content class
        .Fields("urn:schemas-microsoft-com:xml-data#element") = arrFields

        'Save change
        .Fields.Update
        .Close
    End With
End Sub
```

The code first calls the function *BuildSchemaArray* to create an array containing the custom schema properties that the new content class will contain. This is an example of how you create values for a multivalued property.

Once the array is built, the code calls the *CreateContentClass* subroutine, which creates a new item in the Schema folder using ADO. The code sets the content class of the item to *urn:content-classes:contentclassdef*, which tells Exchange Server that this is a new content class definition item. The code then sets the name of the new content class definition to the name passed to the subroutine. In this example, the content class name is *urn:content-class:trainingevent*.

The code next sets the *extends* property. This multivalued string property contains a list of all the other content classes that this content class inherits from. A content class can either have no inheritance or can inherit from one or more content classes. What you want to accomplish in your application and the set of properties you need on your custom content class will dictate which content classes you inherit from. Since the *trainingevent* content class is so similar to the *appointment* content class in terms of properties required, the code inherits from the *appointment* content class. The code also inherits from the *item* content class, which most other content classes inherit from.

Finally, the code sets the *element* property. This property is multivalued and should contain all the custom fields that your content class implements. Once this property is set, you can call the *Update* method on the ADO Fields collection to write the data to Exchange Server.

You can also use XML to create content class definitions and property definitions. If you create a correctly structured XML document and place it into the schema folder for your application, Exchange Server will allow you to use these new content classes and properties. The example on the following page shows what two XML files—one for the content class definition and one for the property definition—look like in XML.

```
This XML file would be for the content class definition

<?xml version="1.0"?>
<Schema name='ExchangeSchema' xmlns="urn:schemas-microsoft-com:xml-data"
   xmlns:d="DAV:" xmlns:ex="http://schemas.microsoft.com/exchange/"
   xmlns:cc="urn:content-classes:"
   xmlns:dt="urn:schemas-microsoft-com:datatypes"
   xmlns:s="urn:schemas-microsoft-com:exch-data:"
   xmlns:m="urn:schemas:httpmail:"
   xmlns:h="urn:schemas:mailheader:" xmlns:cs="http://customschemaname/">

<ElementType name="cs:customprop" d:contentclass="cc:contentclassdef">
<extends type="cc:item"/>

<element type="cs:mycustompropname"/>

</ElementType>
</Schema>

This XML file would be for the property definition
<?xml version="1.0"?>
<Schema name='ExchangeSchema' xmlns="urn:schemas-microsoft-com:xml-data"
   xmlns:d="DAV:" xmlns:cc="urn:content-classes:"
   xmlns:dt="urn:schemas-microsoft-com:datatypes"
   xmlns:s="urn:schemas-microsoft-com:exch-data:"
   xmlns:m="urn:schemas:httpmail:" xmlns:cs="http://customschemaname/">

<ElementType name="cs:PNAC" dt:type="string" s:ismultivalued="0"
   d:contentclass="cc:propertydef" s:isindexed="1"
   s:isreadonly="0" s:isrequired="0" s:isvisible="1">
</ElementType>
</Schema>
```

Setting the *expected-content-class* Property

You need to take a few more steps when using custom schemas. First, you must set the property *urn:schemas-microsoft-com:exch-data:expected-content-class* on the folders for your application. This multivalued property tells Exchange Server what content classes to expect in the folder. This property can specify multiple content classes. The query processor in Exchange Server will use this list of expected content classes when you issue a SELECT * statement in the folder to determine which properties—besides the built-in schema ones—the query should return.

You should try to limit your use of SELECT * statements if possible, because the query processor will have to retrieve every built-in and custom property in your content class, which can take a long time. If you know the specific properties you

need to retrieve as part of your resultset, you should specify them in your query. Here's an example of setting the *expected-content-class* property in code:

```
oFolderRecord.Fields _
   ("urn:schemas-microsoft-com:exch-data:expected-content-class").Value = _
   Array("urn:content-classes:appointment", _
   "urn:content-classes:mycontentclass")
```

Setting the *schema-collection-ref* Property

In addition to setting the expected content class property, you need to set the *urn:schemas-microsoft-com:exch-data:schema-collection-ref* property on your application's folders. This property contains a URL that links to the first folder in which you want the Exchange server to look for schema definition items. In your custom applications, you'll set this property to the Schema subfolder. When we discuss using ADO with Exchange Server later in the chapter, you'll see how to determine the URL to set the value of this property to. If you do not set this property on your folders, Exchange Server will default to its built-in Schema folder in the non-ipm subtree. Here is an example of setting this property in code:

```
oFolderRecord.Fields _
   ("urn:schemas-microsoft-com:exch-data:schema-collection-ref").Value = _
   "file://./backofficestorage/domain/public folders/myapp/schema"
```

Setting the *baseSchema* Property

You have one final task to perform to make your application's schema work well: you need to set the *urn:schemas-microsoft-com:exch-data:baseschema* property on your schema subfolder. This multivalued property provides URLs that Exchange Server will search if it doesn't find schema definitions in your schema subfolder. The server will search the set of URLs you provide in the order they're set in the property. So if Exchange Server finds the correct schema definition, it will stop searching subsequent URLs. By setting the *schema-collection-ref* property for your folders, you can have Exchange Server search your schema folder first and then search the built-in schema folder. The following is an example of setting this property:

```
strBaseSchemaFolder = _
   "file://./backofficestorage/domainname/public folders/" & _
   "non_ipm_subtree/schema/"
With oSchemaFolder
   .Fields("urn:schemas-microsoft-com:exch-data:baseschema") = _
      Array (strBaseSchemaFolder)

   .Fields.Update
End With
```

USING ADO 2.5 AND OLEDB 2.5 WITH EXCHANGE SERVER 2000

While Windows 2000 ships with OLEDB 2.5 and ADO 2.5, Exchange Server 2000 includes the OLEDB 2.5 provider for Exchange Server. If you've done any development with Microsoft SQL Server, you know how easy to use and powerful the ADO object model is. The OLEDB provider in Exchange Server 2000, called EXOLEDB, capitalizes on this ease and power. In this section, we'll take a quick look at some of the new features in ADO 2.5. Then, using the Training application, we'll examine how to build ADO applications with Exchange Server.

Note that the OLEDB provider should be used only server-side with Exchange Server 2000. Do not attempt to use the provider remotely from client-side applications, including using the new version of CDO on the client-side. For those applications that require client-side interaction, you can leverage WebDAV or previous versions of CDO or the Outlook Object Model.

New ADO 2.5 Features

Two major enhancements of ADO 2.5 should be of interest to Exchange Server developers: URL binding support, and Record and Stream support. These enhancements are why ADO 2.5 works so well with the type of data stored in an Exchange Server application. There are two reasons these enhancements make using ADO with Exchange Server easy:

- Exchange Server is best at storing semistructured data rather than structured, relational data.
- Exchange Server supports URL addressing for every record in the Exchange Server database.

URL Binding Support

In my opinion, one of the most annoying things about the earlier versions of ADO is all the setup work required to open a data source connection. You have to create a Connection object, set the properties on that object, and then issue a query and retrieve your Recordset from the data source. However, ADO 2.5 lets you bypass all the setup work. You simply provide a URL for the data you want to retrieve, and OLEDB sets up all the connections and manages all the security for you. Following is an example of binding directly to an Exchange Server resource with this new URL support:

```
'You can achieve the binding like this:
Set oRec = CreateObject("ADODB.Record")
oRec.Open "http://server/public/myfolder/myitem", , _
```

```
   adReadWrite, adOpenIfExists
'Or with one statement, like this:
oRec.Open "file://./backofficestorage/domain/" & _
   "public folders/myfolder/myitem", , adReadWrite, _
   adOpenIfExists
```

Notice how I don't have to create a Connection object. OLE DB does this automatically. The technology that makes this URL binding work is the new Root Binder in OLE DB 2.5. The Exchange Server OLE DB provider and other OLE DB providers register themselves with the Root Binder, specifying which types of URLs they support. For example, the Exchange Server OLE DB provider registers for the URL *"file://./backofficestorage"*. Using the file protocol, you can access mailboxes and Public Folders via *"file://./backofficestorage/domain/public" folders/myfolder*.

Be aware that EXOLEDB does not register for the URLs that begin with *http://*. However, the Microsoft OLE DB Provider for Internet Publishing does. Therefore, if you want to use *http://* in the URLs to connect to Exchange Server data, you must explicitly specify that you want to use the EXOLEDB provider. To do so, you might want to create or grab an explicit Connection object that sets the provider to EXOLEDB and then use that object in your ADO methods. In order to work with Exchange Server using the EXOLEDB provider and *http://* URLs, the code similar to the code just shown now looks like this:

```
Set oConnection = CreateObject("ADODB.Connection")
'Open a connection to the Exchange server by using a throwaway
'Record
Set oThrowAwayRec = CreateObject("ADODB.Record")
oThrowAwayRec.Open "http://server/public/"
set oConnection = oThrowAwayRec.ActiveConnection
oConnection.Provider = "EXOLEDB.DataSource"
Set oRec = CreateObject("ADODB.Record")
oRec.Open "http://server/public/myfolder/myitem", _
   oConnection, adReadWrite, adOpenIfExists
```

> **NOTE** There are other ways to create the Connection object. Keep in mind that if you want to use *http://* URLs, you have to make sure the provider is EXOLEDB. Also, notice that to get at Public Folders, you do not use Public Folders in the *http://* URL like you do the *File://* URL. Instead, Exchange automatically creates a public virtual directory that points at your Public Folders. If you create other top-level hierarchies, and you are going to access folders using the *http://* URL, you will want to create virtual directories that point at the hierarchy that contains those folders.

If you plan to use OLE DB transactions in your code, you might want to continue to explicitly create Connection objects. In ADO, you accomplish this by using the *BeginTrans*, *CommitTrans*, and *RollbackTrans* methods.

Because you'll be using URLs extensively with Exchange Server 2000, there is one property that you should commit to memory quickly: the *DAV:href* property, which is available on every item in Exchange Server. This property contains the URL to the associated item, making it easy for you to retrieve the URL and pass it to procedures or functions in your application.

Record and Stream Support

Previous versions of ADO allowed you to create only Recordsets. Yet ADO 2.5 enables you to create Records and Streams. Creating Recordsets is useful for relational data sources because Recordsets are very "rectangular," meaning that the data in the data sources will be uniform. However, nonrelational data sources such as Exchange Server are not very rectangular in nature. In fact, one row in an Exchange Server database could have many more or many less columns than the next row, since Exchange Server provides very flexible schema support even within a table (or folder, as it's called in Exchange Server). This is why creating Records and Streams of information is a welcome addition to ADO 2.5.

Record object

In previous versions of ADO, if you wanted to open a single row, you had to create an entire Recordset to contain that single row. However, with Record support, ADO allows you to pass the URL to a unique item in Exchange Server, and without requiring you to create a Connection or Recordset object, ADO opens that item into your Record object. You can then use the properties and methods of the Record object to manipulate the data. You'll see the Record object used extensively throughout the code samples for the Training application shown later in the chapter.

Stream object

You can use the Stream object to manipulate streams of data, either text or binary. An Exchange Server message is a good example of a stream. When you retrieve the stream of a message, you're given the entire message, its headers, and its content as serialized data. You can then manipulate the data using the methods on the Stream object. Later in Chapter 19 when we look at how to set up the workflow process definition programmatically, you'll see an excellent example of working with streams.

Putting ADO 2.5 to Work with Exchange Server 2000

You'll predominantly use ADO 2.5 to access data in Exchange Server. Exchange Server supports the major features of ADO, including the ability to perform queries with clauses such as WHERE or ORDER BY. However, there is one major restriction for issuing SQL queries against Exchange Server: Exchange Server does not provide JOIN support, meaning you cannot join two Exchange Server folders into a single Recordset.

SELECT Statement

Exchange Server 2000 supports SQL SELECT statements. This is the basic format of a SELECT statement:

```
SELECT * | select-list
   FROM SCOPE( resource-list)
   [WHERE search-condition]
   [order-by-clause]
```

Remember, you should not use SELECT * if you can avoid it. Instead, pass in the property list that you want to retrieve from Exchange Server as part of your SELECT statement. An example of retrieving a specific set of properties from Exchange Server is shown here. Notice that you will need to place property names in quotes in order to retrieve them.

```
'Build the SQL statement
strURL = "file://./backofficestorage/domain/public folders/myfolder/"
strSQL = "SELECT ""urn:schemas:httpmail:subject"" " & _
   "FROM scope('shallow traversal of """ & strURL & _
   """') WHERE ""DAV:iscollection"" = false AND ""DAV:ishidden"" = false"

'Create a new Recordset object
Set rst = New Recordset
With rst
   'Open Recordset based on the SQL string
   .Open strSQL
End With

If rst.BOF And rst.EOF Then
   Debug.Print "No Records!"
   End
End If

rst.MoveFirst
Do Until rst.EOF
Debug.Print rst.Fields("urn:schemas:httpmail:subject").value
   rst.MoveNext
Loop
```

You'll notice that the SELECT statement contains a SCOPE clause and a WHERE clause. The SCOPE clause identifies the resource at which you want the search to begin, such as the URL of a folder. Exchange Server supports deep and shallow traversals of the scope. Requesting a deep traversal of the URL means that Exchange Server will search not only the specified folder, but also all subfolders it contains. As you'd expect, requesting a shallow traversal means Exchange Server will search only the specified folder.

> **NOTE** Deep traversals are not supported in the MAPI Public Folder top-level hierarchy, but shallow traversals are. Deep traversals are supported in other TLHs, or in Mailbox folders, however. If your application requires deep traversals, this is something you should take into consideration.

With the SCOPE clause, you can request multiple folders as long as the traversal is the same for all the folders. For example, you might want to perform deep traversals of multiple folders that do not have a parent-child relationship. This means that a single-level deep traversal would not search all the folders. Using the SCOPE clause, you can specify multiple URLs to search. The following example searches multiple folders using a shallow traversal. Please note, though, that your traversal must be the same for all URLs and that this technique will not work in the Public Folder hierarchy.

```
'Build the SQL statement
strURL = "file://./backofficestorage/domain/myTLH/myfolder/Events/"
strURL2 = "file://./backofficestorage/domain/myTLH/myfolder/Students/"

strSQL = "SELECT ""urn:schemas:httpmail:subject"" " & _
   "FROM SCOPE('shallow traversal of """ & strURL & _
   """', 'shallow traversal of """ & strURL2 & _
   """') WHERE ""DAV:iscollection"" = false AND ""DAV:ishidden"" = false"
```

Use of the SCOPE clause is optional. If you do not specify the clause, the shallow traversal is used. When you have a SCOPE clause without a traversal, the default traversal is the deep traversal. The following example illustrates this:

```
'Build the SQL statement
strURL = "file://./backofficestorage/domain/myTLH/myfolder/Events/"

'Shallow traversal is used
strSQL = "SELECT ""urn:schemas:httpmail:subject"" " & _
   "FROM """ & strURL & """"

'Deep traversal is used
strSQL = "SELECT ""urn:schemas:httpmail:subject"" " & _
   "FROM SCOPE('""" & strURL & """')"
```

The WHERE clause in your Exchange Server queries can be as complex or simple as you want. The examples above simply check whether the items are collections, which is really a check to see whether the items are folders or whether they are hidden. If either of these checks is true, the item is not returned from the query.

The Training application uses some very complex WHERE clauses in its searches. For example, the following SQL statement is taken from this application's showcourses.asp file, which shows the available courses that a student can take:

```
Function TurnintoISO(dDate,strType)
   'Format should be "yyyy-mm-ddThh:mm:ssZ"
   strISO = Year(dDate) & "-"
   if Month(dDate)<10 then
      strISO = strISO & "0" & Month(dDate) & "-"
   else
      strISO = strISO & Month(dDate) & "-"
   end if
```

```
   if Day(dDate)<10 then
      strISO = strISO & "0" & Day(dDate) & "T"
   else
      strISO = strISO & Day(dDate) & "T"
   end if

   if strType = "End" then
      'Make it 23:59:59 PM on the day
      strISO = strISO & "23:59:59Z"
   else
      'Make it first thing in the morning 00:00:01
      strISO = strISO & "00:00:01Z"
   end if

   TurnintoISO = strISO
End Function


'Figure out the sort order from a QueryString variable
if Request.QueryString("SortBy") = "" then
   strSortBy = """urn:schemas:mailheader:subject"""
elseif Request.QueryString("SortBy") = "0" then
   strSortBy = """urn:schemas:mailheader:subject"""
elseif Request.QueryString("SortBy") = "1" then
   strSortBy = """urn:schemas:calendar:dtstart"""
elseif Request.QueryString("SortBy") = "2" then
   strSortBy = """" & strSchema & "category"", _
      ""urn:schemas:mailheader:subject"""
else
   strSortBy = """urn:schemas:mailheader:subject"""
end if


'Figure out the date to show from QueryString

if Request.QueryString("DateStart") = "" then
   dDateStart = Date
else
   dDateStart = CDate(Request.QueryString("DateStart"))
end if

if Request.QueryString("DateEnd") = "" then
   dDateEnd = Date
else
   dDateEnd = CDate(Request.QueryString("DateEnd"))
end if

'Put this date into an ISO format
dISODateEnd = TurnintoISO(dDateEnd,"End")
dISODateStart = TurnintoISO(dDateStart,"Start")
```

(continued)

```
'Build the SQL statement
strSQL = "Select ""urn:schemas:mailheader:subject"","" _ &
   """DAV:href"",""urn:schemas:calendar:dtstart"","" _ &
   """urn:schemas:calendar:dtend"" FROM scope('shallow traversal of """ _
   & strScheduleFolderPath & _
   """') WHERE (""DAV:iscollection"" = false) AND " & _
   "(""DAV:ishidden"" = false) " & _
   "AND (""urn:schemas:calendar:dtstart"" >= " & _
   "CAST(""" & dISODateStart & """ as 'dateTime'))" & _
   "AND (""urn:schemas:calendar:dtstart"" <= " & _
   "CAST(""" & dISODateEnd & """ as 'dateTime'))" & _
   "ORDER BY " & strSortBy
```

Notice that the SELECT statement uses the CAST clause to cast the date specified from the URL to the ASP page into a dateTime data type. This is necessary if you plan to compare dates in your SQL statements. You'll find that you also need to use the CAST clause to set the correct data types for multivalued properties. The format for a multivalued string would be *CAST("Properties" as 'mv.string')*. Use the CAST clause for your custom properties that are not strings, except for Booleans. Otherwise, your custom properties will be evaluated by Exchange as strings.

You'll also notice that the date data type must be formatted into a standard ISO8601 format, such as yyyy-mm-ddThh:mm:ssZ. Exchange Server stores all dates in Universal Time Coordinate (UTC) and expects your date queries to use the ISO8601 format. Therefore, when working with ADO, all dates will be returned as UTC dates rather than in the client's local time zone. If you want time-zone conversion performed for you automatically, you need to use CDO instead.

ORDER BY Clause

Exchange Server supports the ORDER BY clause. To see how to use ORDER BY, see the next code sample taken from the Training application. Exchange Server enables you to sort your records by multiple columns in both ascending and descending order. In the Training application, for example, the user can sort the list of courses by subject, by date, or by category and then by date. You can specify a list of properties in your ORDER BY clause. If you do not specify a sorting order for your property, the default sorting order is ascending. To specify ascending or descending order, use ASC or DESC, respectively.

```
'Figure out the sort order from a querystring variable
if Request.QueryString("SortBy") = "" then
   strSortBy = """urn:schemas:mailheader:subject"""
elseif Request.QueryString("SortBy") = "0" then
   strSortBy = """urn:schemas:mailheader:subject"""
elseif Request.QueryString("SortBy") = "1" then
   strSortBy = """urn:schemas:calendar:dtstart"""
elseif Request.QueryString("SortBy") = "2" then
```

```
   strSortBy = """" & strSchema & "category"","" & _
      ""urn:schemas:mailheader:subject"""
else
   strSortBy = """urn:schemas:mailheader:subject"""
end if

'This will eventually become part of the overall SELECT
'statement, such as:
'SELECT prop FROM URL WHERE condition
'ORDER BY strSortBy
```

LIKE Predicate

Exchange Server 2000 supports the LIKE predicate, which allows you to perform queries using pattern matching of wildcard characters. The format for the LIKE predicate follows:

```
SELECT Select_List | *
   FROM_Clause
   [WHERE Column_Reference [NOT] LIKE 'String_Pattern']
   [ORDER_BY_Clause]
```

You can specify any column you want, as long as its data type is compatible with the string pattern specified. Table 18-1 shows the wildcard characters you can use.

Wildcard	*Character Symbol*	*Description*
Percent	%	Matches one or more characters
Underscore	_	Matches exactly one character
Square brackets	[ ]	Matches any single character in the range or set you specify in the brackets
Caret	^	Matches any single character not within the range

Table 18-1. *Wildcard characters you can use with the LIKE predicate.*

The following code is taken from the search page for the Training application. This code uses the % wildcard to perform a full-text search on certain properties to see whether they contain a substring of the search criteria specified in strText.

```
strFullText = " AND (""urn:schemas:mailheader:subject"" LIKE '%" _
   & strText & "%' OR " & _
   " ""urn:schemas:httpmail:textdescription"" LIKE '%" & strText & _
   "%' OR " & _
   " ""urn:schemas:calendar:location"" LIKE '%" & strText & "%' OR " & _
   " """ & strSchema & "instructoremail"" LIKE '%" & strText & "%' OR " & _
   " """ & strSchema & "prereqs"" LIKE '%" & strText & "%')"
```

Be aware that the LIKE predicate uses the Exchange Server query processor. This is different than the CONTAINS and FREETEXT statements, which we'll examine when we look at content indexing. Both CONTAINS and FREETEXT require you to turn on content indexing before you can use them in your SQL statement. The LIKE predicate is not as fast as CONTAINS or FREETEXT because the Exchange Server query processor must search each item to see whether one of its columns contains the value you specified.

GROUP BY Predicate

Exchange Server 2000 supports the GROUP BY predicate, allowing you to group all rows that have the same value into a single row. This is useful for cases in which you might want to obtain distinct counts based on a specific value. For example, you could use GROUP BY to count the number of people listed in your Inbox who sent you e-mail. Instead of having to scroll through your Inbox and count the different From addresses, you can use the GROUP BY statement, as shown here:

```
strURL = "file://./backofficestorage/domain/mbx/mailbox/inbox/"

'Build the SQL statement
strSQL = "Select ""urn:schemas:mailheader:from"" " & _
   "From scope('shallow traversal of """ & strURL & _
   """') GROUP BY ""urn:schemas:mailheader:from"""
Set conn = New ADODB.Connection
With conn
   .Provider = "exoledb.datasource"
   .Open strURL
End With

'Create a new Recordset object
Set rst = New Recordset
With rst
   'Open Recordset based on the SQL string
   .Open strSQL, conn
End With

If rst.BOF And rst.EOF Then
   Msgbox "No values found!"
   End
End If

iCount = 0
rst.MoveFirst
Do Until rst.EOF
   iCount = iCount + 1
   strFrom = rst.Fields("urn:schemas:mailheader:from").Value
   Debug.Print "From: " & strFrom
```

```
   rst.MoveNext
Loop
Debug.Print "There are " & iCount & _
   " distinct FROM addresses in your inbox."
```

Here is the output from this code sample:

```
From: "System Administrator"
   <postmaster@thomriznt5dom.extest.microsoft.com>
From: "Thomas Rizzo" <thomriz@thomriznt5dom.extest.microsoft.com>
From: <notifications@domain.com>
There are 3 distinct FROM addresses in your Inbox.
```

CREATE INDEX Predicate

You might be wondering how you actually force Exchange Server to create a database index on the properties for which you specified the *isindexed* property to be *True*. The current way in RC1 is to specify a SELECT * statement on the folder. The post-RC1 method and the preferred way to do this is to use the CREATE INDEX statement. Here is the format for this statement:

```
CREATE INDEX * ON scopeURL (*)
```

A command that uses this statement would look like this:

```
CREATE INDEX * ON
   file://./backofficestorage/domain/public folders/my folder/ (*)
```

Note that the scope of this statement is the folder, so if you have multiple application folders, you'll want to run the statement multiple times with the different scopes. One issue to note is that the index created by Exchange is kept around permanently, so you need to issue this statement only once per folder. You cannot explicitly name or delete the index. Creating the index should give your applications faster performance. This index is different from the full-text index, which we'll discuss in the next chapter.

Aliasing Column Names

To make it easier for you to work with the long schema names that Exchange Server provides, the Exchange Server OLE DB provider supports aliasing column names. This means you can give friendly names to the column names of the schema in your SELECT statements. The following is an example of such aliasing:

```
strURL = "file://./backofficestorage/domain/mbx/thomriz/inbox/"

'Build the SQL statement

strSQL = "Select ""urn:schemas:mailheader:from"" AS " & _
   "EmailFrom From scope('shallow traversal of """ & strURL & _
   """') ORDER BY EmailFrom"
```

(continued)

```
'Create a new Recordset object
Set rst = New Recordset
With rst
   'Open Recordset based on the SQL string
   .Open strSQL
End With

If rst.BOF And rst.EOF Then
   End
End If

rst.MoveFirst
Do Until rst.EOF
Debug.Print "From: " & rst.Fields("EmailFrom").Value
   rst.MoveNext
Loop
```

As you'll notice in the code, aliasing requires you to use the AS keyword and then the name of the alias you want for the column. You'll also notice that aliasing is supported in the ORDER BY clause. However, it is not supported in the WHERE or GROUP BY clauses. Also, you'll see that aliasing is supported when using the Fields collection. This makes it easy to create short, memorable aliases for the schema names in your applications.

> **NOTE** Aliasing will work only for a specific SELECT statement; it is not global in nature.

WHICH SQL STATEMENTS ARE NOT SUPPORTED?

Exchange Server 2000 does not support the following SQL statements: SET, DISTINCT, DELETE, INSERT, CONVERT, DATASOURCE, CREATE VIEW, COUNT, SUM, AVG, MIN, and MAX. Also, remember that Exchange Server does not support JOIN.

Common Tasks Performed with ADO

This section describes the eight most common tasks you'll perform when using ADO with Exchange Server: creating folders, creating items, deleting folders or items, copying folders or items, moving folders or items, working with the Fields collection, working with Recordsets, and handling errors.

Creating New Folders

To create a new folder using ADO, you need to construct the URL to the folder and then use the *Open* method on the ADO Record object with the parameter *adCreateCollection*. The following code, taken from the setup program of the Training application, creates the application folders by using ADO:

```
Private Sub CreateFolder(strFolderPath, strFolderName, _
   strExpectedContentClass, strMAPIFolderClass)
   Dim oFolderRecord As New ADODB.Record
   oFolderRecord.Open strFolderPath & strFolderName, _
   oConnection, adModeReadWrite, adCreateCollection
   oFolderRecord.Fields("urn:schemas-microsoft-com:exch-data:" & _
      "expected-content-class").Value = Array("urn:content-classes:" & _
      strExpectedContentClass)
   'Set the Schema collection reference to our schema folder
   'even though it may not exist yet
   oFolderRecord.Fields("urn:schemas-microsoft-com:exch-data:" & _
      "schema-collection-ref").Value = strFolderPath & "schema"
   oFolderRecord.Fields("DAV:contentclass").Value = _
      "urn:content-classes:folder"
   'Set the PR_CONTAINER_CLASS
   'so that Outlook displays it correctly
   oFolderRecord.Fields("http://schemas.microsoft.com/mapi/proptag/" & _
      PR_CONTAINER_CLASS).Value = strMAPIFolderClass
   oFolderRecord.Fields.Update
   oFolderRecord.Close
End Sub

Private Sub CreateFolders()
   On Error GoTo errHandler
   'Create all the folders in a local OLE DB transaction
   'so that all are created or none are created
   oConnection.BeginTrans
   'Create the root folder first
   Dim oRecord As New ADODB.Record
   oRecord.Open strExchangeServerFilePath & txtFolderPath, _
      oConnection, adModeReadWrite, adCreateCollection
   oRecord.Fields("DAV:contentclass").value = "urn:content-classes:folder"
   oRecord.Fields("http://schemas.microsoft.com/mapi/proptag/" & _
      PR_CONTAINER_CLASS).Value = "IPF.Note"
   strPath = oRecord.Fields("DAV:href")
   oRecord.Close
   strPath = strPath & "/"
   CreateFolder strPath, "Categories", "message", "IPF.Note"
   CreateFolder strPath, "Configuration", "message", "IPF.Note"
   CreateFolder strPath, "Course Materials", "message", "IPF.Note"
   CreateFolder strPath, "Discussions", "message", "IPF.Note"
   CreateFolder strPath, "Emails", "message", "IPF.Note"
   CreateFolder strPath, "Instructors", "instructor", "IPF.Contact"
   CreateFolder strPath, "Pending", "message", "IPF.Note"
   CreateFolder strPath, "Schedule", "trainingevent", "IPF.Appointment"
   CreateFolder strPath, "schema", "message", "IPF.Note"
   CreateFolder strPath, "Students", "student", "IPF.Contact"
   CreateFolder strPath, "Surveys", "message", "IPF.Note"
```

(continued)

```
   CreateFolder strPath, "Surveys\Courses", "survey", "IPF.Note"
   CreateFolder strPath, "Surveys\Instructors", "survey", "IPF.Note"
   If Err.Number = 0 Then
      oConnection.CommitTrans
   End If
   Exit Sub

errHandler:
   MsgBox "Error in CreateFolders.  Error " & Err.Number & " " & _
      Err.Description
   oConnection.RollbackTrans
End Sub
```

Notice that the code wrapped in an OLE DB transaction. The *CreateFolder* subroutine takes the full URL to the new folder to be created, creates a new Record object, and calls the *Open* method on that object with the URL. The *Open* method takes several parameters. The syntax for the *Open* method is shown here:

```
Open Source, ActiveConnection, Mode, CreateOptions, Options, _
   UserName, Password
```

To create the folder, you need to pass the *Mode* parameter the value *adReadWrite*, and the *CreateOptions* parameter the value *adCreateCollection*. Exchange 2000 does not support passing a username and password.

If you're using schema, you can set the expected-content-class and schema-collection-ref fields on the folder. You should set the content class of the folder to *urn:content-classes:folder*. Also, if you plan to display the folder in Outlook, you need to set the *PR_CONTAINER_CLASS* property in the MAPI name space, which is *"http://schemas.microsoft.com/mapi/proptag/"*. You should pass the unique hex identifier to the property you're interested in without the leading 0. For example, 0x00212 would be *x00212*. If you don't want to set the MAPI property yourself, you could instead set *http://schemas.microsoft.com/exchange/outlookfolderclass* to *IPF.Note* or another value.

If you're trying to access custom properties you created in Outlook via ADO, you need to use a slightly different format. The following code shows you how to get a custom property, called *MyProp*, that you set through MAPI. This property can be created using MAPI itself, CDO, or Outlook. The long GUID that you see in the code is the ID for the *PS_PUBLIC_STRINGS* property set, where public properties are created in MAPI.

```
strSQL = "Select ""http://schemas.microsoft.com/mapi/string/" & _
   "{00020329-0000-0000-c000-000000000046}/MyProp"" AS MAPIProp" & _
   " From scope('shallow traversal of """ & strURL & """')"

'Create a new Recordset object
Set rst = New Recordset
```

```
With rst
   'Open Recordset based on the SQL string
   .Open strSQL
   End With

If rst.BOF And rst.EOF Then
   End
End If

rst.MoveFirst
Do Until rst.EOF
   Debug.Print "MyProp: " & rst.Fields("MAPIProp").Value
   rst.MoveNext
Loop
```

Note that this format is not the only format you can use to query for MAPI properties. One format requires that you know the propset ID and the hexadecimal value for the property, and it is illustrated in the next bit of code. The other two formats are easier to use, but the one I just mentioned can be used to access properties in your own namespaces beyond the public strings namespace.

```
http://schemas.microsoft.com/mapi/id/{propset GUID}/value

Example:

http://schemas.microsoft.com/mapi/id/{3f0a69e0-7f56-11d2-b536-
00aa00bbb6e6}/0xfeedface
```

Creating New Items

Creating new items in Exchange Server using ADO is very similar to creating folders with ADO. The only difference is that you will pass as the *CreateOptions* parameter the value *adCreateNonCollection*. Other than that, it's pretty much the same code. The following code from the setup program of the Training application shows how to create a new item. The code creates e-mail messages for different notifications for the application, such as a new course or an authorization requirement. This makes the application fairly easy to configure, depending on which settings you specify. Note that I set the MAPI property tag for the message class. You could also set the property *http://schemas.microsoft.com/exchange/outlookmessageclass* to the correct message class for your item.

```
Private Sub CreateEmailTemplates()
   On Error GoTo errHandler:

   oConnection.BeginTrans
   'Open the Emails folder
   Dim oRec As New ADODB.Record
   oRec.Open strPath & "Emails", oConnection, adModeReadWrite, _
      adFailIfNotExists
```

(continued)

```
'Create a new post in the folder by using ADO
Dim oEmail As New ADODB.Record
oEmail.Open strPath & "Emails/New Discussion Group Email", _
   oConnection, adModeReadWrite, adCreateNonCollection

'Open the text file on the hard drive
'and read the data

Open App.Path & "\discussionemail.txt" For Input As #1
Dim strMessage
Do While Not EOF(1)
   Input #1, strLine
   strMessage = strMessage & strLine
Loop
Close #1

oEmail.Fields("urn:schemas:httpmail:textdescription").Value = _
   strMessage
oEmail.Fields("DAV:contentclass") = "urn:content-classes:message"
oEmail.Fields("http://schemas.microsoft.com/mapi/proptag/" & _
   PR_MESSAGE_CLASS).Value = "IPM.Post"
oEmail.Fields.Update
oEmail.Close

oEmail.Open strPath & "Emails/ " & _
   "Survey Email", oConnection, adModeReadWrite, adCreateNonCollection

'Open the text file on the hard drive
'and read the data

Open App.Path & "\surveyemail.txt" For Input As #1
strMessage = ""
Do While Not EOF(1)
   Input #1, strLine
   strMessage = strMessage & strLine
Loop
Close #1

oEmail.Fields("urn:schemas:httpmail:textdescription").Value = _
   strMessage
oEmail.Fields("DAV:contentclass") = "urn:content-classes:message"
oEmail.Fields("http://schemas.microsoft.com/mapi/proptag/" & _
   PR_MESSAGE_CLASS).Value = "IPM.Post"
oEmail.Fields.Update
oEmail.Close


oEmail.Open strPath & "Emails\New Course Email", oConnection, _
   adModeReadWrite, adCreateNonCollection

'Open the text file on the hard drive
'and read the data
```

```
Open App.Path & "\courseemail.txt" For Input As #1
strMessage = ""
Do While Not EOF(1)
   Input #1, strLine
   strMessage = strMessage & strLine
Loop
Close #1

oEmail.Fields("urn:schemas:httpmail:textdescription").Value = _
   strMessage
oEmail.Fields("DAV:contentclass") = "urn:content-classes:message"
oEmail.Fields("http://schemas.microsoft.com/mapi/proptag/" & _
   PR_MESSAGE_CLASS).Value = "IPM.Post"
oEmail.Fields.Update
oEmail.Close

'Create a new post in the folder by using ADO
oEmail.Open strPath & "Emails/WorkflowMessage", oConnection, _
   adModeReadWrite, adCreateNonCollection

'Open the text file on the hard drive
'and read the data

Open App.Path & "\workflowmessage.txt" For Input As #1
strMessage = ""
Do While Not EOF(1)
    Input #1, strLine
    strMessage = strMessage & strLine
Loop
Close #1

oEmail.Fields("urn:schemas:httpmail:textdescription").Value = _
   strMessage
oEmail.Fields("DAV:contentclass") = "urn:content-classes:message"
oEmail.Fields("http://schemas.microsoft.com/mapi/proptag/" & _
   PR_MESSAGE_CLASS).Value = "IPM.Post"
oEmail.Fields.Update
oEmail.Close



oRec.Close


If Err.Number = 0 Then
   oConnection.CommitTrans
End If

Exit Sub
```

(continued)

```
errHandler:
   MsgBox "Error in CreateEmailTemplates.  Error " & Err.Number & _
      " " & Err.Description
   oConnection.RollbackTrans
   End
End Sub
```

When creating items using ADO, you have to watch out for conflicting names. Check to see whether you receive an error when trying to create an item or folder. If so, pick a different name or add a random number to the end of the name. The following code shows you how to do this:

```
On Error Resume Next
Set rec = Server.CreateObject("ADODB.Record")
With rec
   .Open strStudentsFolderPath & "/" & strName, , 3, 0
   if err.number = &H80040e98 then
   'Already exists; try to open with a random name!
      err.clear
      Randomize
      iRandom = Int((50000 * Rnd)+1)
      .Open strScheduleFolderPath & "/" & strName & iRandom,,3,0

   end if
End With
```

Deleting Folders or Items

To delete a folder or an item, you can use the *DeleteRecord* method on an ADO Record object. If you delete a folder, all items in the folder also will be deleted. The following code sample shows you how to delete a folder called MyFolder:

```
Set Rec = CreateObject("ADODB.Record")
'This URL is for a public folder
strURL = "file://./backofficestorage/" & DomainName & "/" & strFolderPath

Rec.Open strURL
Rec.DeleteRecord
'Or you could do
'Rec.DeleteRecord strURL
Rec.Close
```

If you want to delete items using a Recordset instead of the Record object, you can use the *Delete* method of the Recordset object. The following example deletes all items in a folder:

```
'Create a query with a WHERE clause to delete only items
strQ = "SELECT * FROM scope('shallow traversal of " & Chr(34) & _
   strURL & Chr(34) & "')"
strQ = strQ & " WHERE ""DAV:isfolder"" = FALSE"
```

```
'Open the Recordset
Rs.Open strQ

Rs.MoveFirst

Do Until Rs.EOF

   'Delete current record (row); 1 is parameter for adAffectCurrent
   Rs.Delete 1
   Rs.MoveNext
Loop
```

Copying Folders or Items

To copy folders or items, use the *CopyRecord* method on the Record object. If you copy a folder, all items and subfolders will be copied as well. Note that you cannot copy folders or items between Exchange Server databases. Therefore, you cannot copy between private or public databases because they reside in different Exchange Server databases. However, you could create a new item in the separate database, copy the properties from the original item to the new item, and save the new item. The *creator*, *creation time*, and other properties on the item will not be the same as those of the original item, however. The following code copies an item named MyItem from one folder to another:

```
Set Rec = CreateObject("ADODB.Record")
strURL = "file://./backofficestorage/" & DomainName & "/" & _
   strPath & "/MyItem"
Rec.Open strURL
Rec.CopyRecord ,"file://./backofficestorage/" & DomainName & _
   "/" strDestination
```

Moving Folders or Items

To move items, you can call the *MoveRecord* method. Moving has the same restrictions as copying in that you cannot move between Exchange Server stores. Moving a folder moves all the items in the folder as well. The following code sample moves an item:

```
Set Rec = CreateObject("ADODB.Record")
Rec.Open URLFrom, , adModeReadWrite
NewURL = Rec.MoveRecord URLFrom, URLTo, "","", adMoveOverWrite
```

This is the syntax for the *MoveRecord* method:

```
URL = MoveRecord (Source, Destination, UserName, Password, Options, Async)
```

This method typically returns the *Destination* parameter, which is a string value of the destination the item was moved to. The *Options* parameter allows you to provide options for the move. The most common option you'll provide is the *adMoveOverWrite* option.

Using the Fields Collection

You've already seen how to get properties using the Fields collection in ADO. You might want to set properties on an item or folder using the Fields collection. If you do so, remember that the property is available only on the specified item rather than on every item in the folder. Adding new fields using the Fields collection is not like adding custom schema. When you add to the Fields collection, your properties are not included when a SELECT * statement is issued. Unlike schema, new properties added to the Fields collection cannot be shared among multiple items in a folder unless you explicitly create the property on every item. With schema, you can set the content class of your items to be of a particular type, and those items will implement the properties that you specified in your schema.

The Fields collection supports the *Append* method, which allows you to add custom fields to a Recordset or a Record object. Here is the syntax for the *Append* method:

```
fields.Append Name, Type, DefinedSize, Attrib, FieldValue
```

The *Type* parameter specifies the data type of the field. This parameter can have one of many different values; however, you'll probably use one of the following: *adWChar*, *adBStr*, *adDate*, *adInteger*, or *adBoolean*.

The *DefinedSize* parameter specifies the size of the new field. Normally, you would leave this parameter blank because ADO will base the size of the new field on the data type you specify.

The *Attrib* parameter allows you specify attributes of the field—for example, whether the field is nullable or contains fixed-length data. Normally, you would leave this parameter blank too, unless your field really needs special handling.

The final parameter is a value for the field. Instead of having to create the field and then use another line of code to set the value, ADO allows you to simply assign the value to the field right in its parameter.

The following code shows you how to add new fields to a record:

```
'Assumes oNewRecord is a new item in Exchange Server
oNewRecord.Fields.Append "MyNewProp",adInteger,,,12
oNewRecord.Fields.Update
```

Notice how the code calls the *Update* method on the Fields collection. If you do not call the *Update* method, your changes won't be saved to Exchange Server—meaning your new field won't be added to the item. Also, if you suspect that the values for your fields might have changed while you were working with your data, you can call the *ReSync* method to have ADO requery Exchange Server for the latest values for your data.

Working with Recordsets By Using ADO

One of the most common tasks you'll perform with ADO is working with Recordsets. To efficiently work with the Recordset object, you'll need to know about some of its

methods. For example, you'll need to know how to scroll through records in the recordset as well as detect the end of the recordset.

The first thing that you will want to know about your Recordset after you have Exchange perform a query for you is the number of items returned in the Recordset. ADO provides a *Recordcount* property on the Recordset object so that you know how many records are contained in the Recordset. This number depends on the cursor type you use for your Recordset. Table 18-2 outlines the different cursor types and how they affect the *Recordcount* property.

Cursor Type	*Description*	*Recordset Count Returned*
Dynamic	Dynamic collection of records that is the most flexible of all cursors. Additions, changes, and deletions by you or other users are immediately visible. You can scroll forwards or backwards through the records.	-1
Keyset	Like a Dynamic cursor except that you cannot see the records added by other users. You cannot access the records that other users delete.	-1
Static	A copy of the records. Additions, changes, and deletions by other users are not reflected, but any type of movement is allowed.	Actual number
Forward Only	Very similar to static except that you can only scroll forward through the records. This is the default type of cursor created.	Actual number

Table 18-2. *Cursor types and their effect on the* Recordcount *property.*

Using the *Recordcount* property's value, you can decide whether you need to scroll through the records if any were returned. To make scrolling the records easy in a Recordset, ADO provides the *BOF* and *EOF* properties on a Recordset object. *BOF* is a Boolean that, if *True*, indicates that the current record position is before the first record—in effect, it indicates that you are at the beginning of the Recordset. *EOF* is a Boolean that, if *True*, indicates that the current record position is after the last record in the Recordset. To ensure you do not read beyond the end or the beginning of your Recordset, you can use these two properties while scrolling through the Recordset either forwards or backwards.

To actually scroll through the recordset, you'll need to use the *MoveFirst*, *MoveNext*, *MoveLast*, and *MovePrevious* methods of the Recordset object. Depending

on the cursor type you specify, these methods will provide you with differing degrees of usability. The method names are pretty self-explanatory. *MoveFirst* moves you to the first record, *MoveNext* moves you to the next record if it exists (and returns an Error object if it doesn't exist), *MoveLast* moves you to the last record, and *MovePrevious* moves you to the previous record.

Handling Errors in ADO

You always have to be prepared to handle errors in your code. ADO provides built-in error-handling capabilities through its Errors collection on the Connection object. The Errors collection contains Error objects. These Error objects have three properties of interest to Exchange developers:

- *Description.* The default property for the Error object. This is a string representation of the error text.
- *Number.* A long integer that contains the number associated with the error. You can look at the *ErrorValueEnum* constants in ADO to see whether this number matches one of the ADO-defined errors.
- *Source.* A string that contains the name of the object that raised the error.

You can use the Errors collection in your error handling to scroll through all the Error objects contained in the collection. When you find the errors, you can then take action on them. For example, you can rollback a transaction, which we'll see shortly, or print an error message for the user.

Using OLE DB Transactions

The Exchange Server OLE DB provider supports local OLE DB transactions. This allows you to treat your ADO operations as tranasactions when working with Exchange Server. If for some reason an operation fails during the transaction, you can roll back the entire transaction without leaving your Exchange Server application in an inconsistent state. Since Exchange Server transactions are different than SQL Server ones, there are a few things you need to know.

First, as you would expect, for the transactions to work you must perform all your ADO operations on the same connection to the Exchange server. Therefore, when using the methods of the Record or Recordset object, always pass as a parameter the Connection object that you created at the beginning of the transaction.

Second, transactions in Exchange Server are not supported across Exchange Server stores. This means transactions will not work if performed on folders or items located in different Exchange Server stores. The most common examples of different Exchange Server stores are the private store and the public store.

Finally, Exchange Server does not support distributed transactions with SQL Server. You cannot wrap in a single transaction ADO commands between the two databases, nor can you use the distributed transaction coordinator with both databases. Therefore, you should use local transactions in each database and should perform all your SQL Server work before doing your Exchange Server work. That way, you guarantee that the SQL Server transactions go through, and you don't have to worry about rolling back your Exchange Server transactions.

You will work with a transaction by using one ADO Connection object. On this Connection object, you will first need to call the *BeginTrans* method, which starts a transaction. When your ADO methods are complete, you call the *CommitTrans* method, which attempts to commit the transaction to the Exchange database. If you find that an error occurred during your ADO calls, you can use the *RollBackTrans* method to roll back the entire transaction. The following example, taken from the Training application setup program, shows you how to use these methods:

```
Private Sub CreateCategoryMessage()
   On Error GoTo errHandler:

   oConnection.BeginTrans
   'Open the Categories folder
   Dim oRec As New ADODB.Record
   oRec.Open strPath & "Categories", oConnection, adModeReadWrite, _
      adFailIfNotExists

   'Create a new post in the folder by using ADO
   Dim oEmail As New ADODB.Record
   oEmail.Open strPath & "Categories\Categories", oConnection, _
      adModeReadWrite, adCreateNonCollection



   oEmail.Fields("urn:schemas:httpmail:textdescription").Value = _
      strCategories
   oEmail.Fields("http://schemas.microsoft.com/mapi/proptag/" & _
      PR_MESSAGE_CLASS).Value = "IPM.Post"
   oEmail.Fields.Update
   oEmail.Close

   oRec.Close


   If Err.Number = 0 Then
      oConnection.CommitTrans
   End If
```

(continued)

```
    Exit Sub
errHandler:
    MsgBox "Error in CreateEmailTemplates.  Error " & Err.Number & _
        " " & Err.Description
    oConnection.RollbackTrans
    End
End Sub
```

Best Practices for Using ADO

There are three best practices for working with ADO and Exchange Server 2000. First, learn your Exchange Server properties well. When working with ADO, you'll need to remember all the different properties in all the different name spaces that Exchange Server provides. The most common ones that you will work with are listed here. For the full list of properties, refer to the Exchange Server SDK included on the companion CD.

- *DAV:href*
- *DAV:contentclass*
- *DAV:displayname*
- *DAV:iscollection*
- *DAV:isfolder*
- *DAV:ishidden*
- *DAV:isreadonly*
- *urn:schemas-microsoft-com:office:office#Author*
- *urn:schemas-microsoft-com:office:office#Category*
- *urn:schemas-microsoft-com:office:office#Comments*
- *urn:schemas:httpmail:to*
- *urn:schemas:httpmail:subject*
- *urn:schemas:httpmail:cc*
- *urn:schemas:httpmail:bcc*
- *urn:schemas:httpmail:hasattachment*
- *urn:schemas:httpmail:from*

The second best practice is to use transactions wherever appropriate. There are a number of reasons for this. First, it allows you to roll back the Exchange Server database to the original state in case of an error. Second, using transactions is key

when working with events. If you create an item in the database using ADO, Exchange Server will fire an event. If you then update the fields after creating the item, Exchange Server will fire another event. You don't want this to happen; you want only one event to fire. (The problem of multiple events firing can be a tough one to debug.) When you use transactions, the information automatically is entered into the Exchange Server database, thereby causing only one event to fire.

The third best practice for working with ADO is to reuse a connection that already exists. Creating new connections to the Exchange Server database taxes your computer's resources because Exchange Server needs to keep open the connections you create. Therefore, you should reuse the same Connection object in a session variable—especially if you're writing ASP applications.

CDO FOR EXCHANGE SERVER 2000

Exchange Server 2000 contains a new version of CDO called CDO for Exchange 2000. This version of CDO builds on OLE DB, meaning that its object model is different from that of previous versions. For example, instead of having navigational objects, such as InfoStore, the new version of CDO relies on ADO for record navigation. Therefore, if you want to query Exchange Server for a specific set of records, you need to rely on ADO. However, if you want to perform collaborative functions on those records, you need to use CDO. You can retrieve items using ADO, and then open and process each item using CDO.

CDO adds collaborative functionality above and beyond that provided by ADO. If you need to create recurring meetings, ADO won't be very helpful—for example, you would have to determine which properties to set to make an appointment recur on the third Tuesday of every month. With CDO, however, you set some properties, save the appointment, and suddenly you have a recurring meeting. CDO does all the hard coding for you behind the scenes.

CDO and ADO were designed to work together. Both object models have the same Fields collection. Furthermore, CDO objects can be bound directly to ADO objects. This allows you to use the same connection with the Exchange server.

CDO Design Goals

Microsoft had a number of design goals in mind for the CDO object library. First, as just discussed, integrating with and extending ADO was key. Second, having learned from previous versions of the CDO object model, the CDO design team knew the model needed to be dual interfaced. Making the CDO object model dual interfaced meant that different development environments, such as Visual Basic, Microsoft Visual C++, and ASP, could take advantage of it. The third goal was that CDO adhere to Internet standards. Internet standards are now critical to CDO since it uses vCard, iCal,

Lightweight Directory Access Protocol (LDAP), Multipurpose Internet Mail Extensions (MIME), MIME Encapsulation of Aggregate HTML Documents (MHTML), Simple Mail Transfer Protocol (SMTP), and Network News Transfer Protocol (NNTP). The final and probably most important goal when designing CDO was to make the developer's job easier by providing a rich set of objects on top of OLE DB for building collaborative applications.

CDO FOR WINDOWS 2000

You might have seen CDO for Windows 2000. Consider this object model the little brother of CDO for Exchange Server 2000. CDO for Windows 2000 provides SMTP and NNTP support. CDO for Windows 2000 also provides support for protocol events such as SMTP events. However, CDO for Windows 2000 does not provide mailbox support or public folder support. CDO for Exchange Server 2000 provides these features and extends CDO for Windows 2000. If you get started with CDO for Windows 2000, you'll have a working knowledge of the basics of CDO for Exchange Server 2000.

CDO Object Model

The CDO object model consists of five main components. Notice I say *components* rather than *objects,* since the CDO object model contains dozens of objects. The five main components are messaging, calendaring, contacts, workflow, and management. We'll cover the most common tasks you'll perform with the CDO library in each of these areas, with the exception of management. I won't cover the specific properties and methods of these CDO object model components in great detail. Since the Exchange Server SDK provides extensive documentation on this subject, I'll refer you to the SDK on the companion CD for specific information.

> **NOTE** To see how you can use the CDO library to perform management tasks, such as setting storage limits and creating stores, look at the CDO for Exchange Management object model on the Exchange Server SDK, included on the companion CD.

Frequently Used Objects in CDO

You'll commonly work with two objects, the Configuration object and the DataSource object, on all CDO objects you use in your applications. Before we look at the most typical CDO functionality you'll use, let's examine these two objects.

Configuration Object

The Configuration object allows you to customize the parameters CDO uses and the way CDO works. For example, using the Configuration object, you could set the e-mail address of the message sender, set the proxy server to use, set the username and password if you require authentication via SMTP or NNTP, or select other configuration options. The following code, taken from the workflow process, sets the sender e-mail address for a meeting request to the notification address you specified in the setup program of the Training application:

```
'Create a throwaway appointment
   set oAppt = CreateObject("CDO.Appointment")
   set oConfig = CreateObject("CDO.Configuration")
   strNotificationAddress = GetWorkflowSessionField("notificationaddress")
   oConfig.Fields(http://schemas.microsoft.com/cdo/ & _
      "configuration/sendemailaddress") = strNotificationAddress
   oConfig.Fields.Update

   oAppt.Configuration = oConfig
```

As you can see in the code, you use the Fields collection on the CDO Configuration object to set your properties. All the properties you can set are contained in the *"http://schemas.microsoft.com/cdo/configuration"* name space.

Another common scenario that you'll use the Configuration object for is setting the time zone for viewing appointments from Exchange Server. Remember that Exchange Server stores dates in UTC. Any dates you retrieve through ADO will be returned in UTC. However, you can use CDO to change UTC dates to local time zone dates for the client. Sometimes you might want to use a different time zone in your application than the one originally detected by CDO. The Configuration object allows you to do this. The following code changes the time zone to Mountain:

```
Dim objAppt As New CDO.Appointment
Dim objConfig As New CDO.Configuration
objConfig.Fields(cdoTimeZoneID) =  cdoMountain
objAppt.Configuration = objConfig
```

DataSource Object

The DataSource object provides access from CDO objects to data sources, such as the Web Storage System or Active Directory. You can use the DataSource object to open or save items to data sources from CDO. There are six methods on the DataSource object that you should become familiar with: *Open*, *OpenObject*, *Save*, *SaveTo*, *SaveToContainer*, and *SaveToObject*.

Open method

The DataSource object's *Open* method is similar to the *Open* method on the ADO Record object. The only difference is that ADO can create items using the *Open* method, while CDO cannot. This is the syntax for the *Open* method:

```
Open(
  ByVal SourceURL as String,
  ByVal ActiveConnection as Object,
  [ByVal Mode as ConnectModeEnum],
  [ByVal CreateOptions as RecordCreateOptionsEnum],
  [ByVal Options as RecordOpenOptionsEnum],
  [ByVal UserName as String],
  [ByVal Password as String]
)
```

A code example that uses the *Open* method in conjunction with an ADO Recordset is shown here:

```
Dim rs as New Recordset
Dim msg as New Message

fldr = "file://./backofficestorage/domain/MBX/user/inbox"

rs.Open "Select * from " & _
   "scope('shallow traversal of " & _
   " & fldr & "')" & _
   "where urn:schemas:mailheader:subject = 'hello'"

rs.MoveFirst
msg.DataSource.Open rs("DAV:href"),rs.ActiveConnection
```

OpenObject method

You can use the *OpenObject* method to open data from another object rather than from a data source such as Exchange Server. A common use for *OpenObject* is to open an embedded message in another message. This is the syntax for *OpenObject*:

```
OpenObject(ByVal Source as Object, ByVal InterfaceName as String)
```

The following code sample opens an embedded message within another message. The code assumes you already retrieved the object that represents the embedded message in a variable named *oBodyPart*.

```
oDataSource.OpenObject oBodyPart, "IbodyPart"
```

Save method

The *Save* method writes back data to the currently opened data source. This method is so simple that we won't even look at a code sample. You should, however, call this method if you change any values that need to be written back. You should also

make sure you open the data source with the read/write flags; otherwise, you will receive an error.

SaveTo method

The *SaveTo* method allows you to save an item to a URL you specify. As you will see momentarily, this method differs from the *SaveToContainer* method, which doesn't let you specify a URL to the item that you want to create. Instead, the *SaveToContainer* method lets you specify the URL to the container where you want to save the item. When you use *SaveToContainer*, CDO generates a GUID to identify your item. This is actually quite useful because you do not have to worry about conflicting URLs when saving items. The syntax for the *SaveTo* method is shown here, along with a code sample:

```
SaveTo(
  ByVal SourceURL as String,
  ByVal ActiveConnection as Object,
  [ByVal Mode as ConnectModeEnum],
  [ByVal CreateOptions as RecordCreateOptionsEnum],
  [ByVal Options as RecordOpenOptionsEnum],
  [ByVal UserName as String],
  [ByVal Password as String]
)


'Assume oMsg is a valid message
Set oDataSource = oMsg

oDataSource.SaveTo "PATHTOFOLDER/myitem.eml", _
   MyCONN, _
   adModeReadWrite, _
   adCreateOverwrite
```

SaveToContainer method

The *SaveToContainer* method, as just discussed, saves the item to a container you specify and assigns a GUID as the identifier for the item. Here is the syntax for the *SaveToContainer* method, along with a code sample taken from the Training application that saves a new course to the schedule folder:

```
SaveToContainer(
  ByVal ContainerURL as String,
  ByVal ActiveConnection as Object,
  [ByVal Mode as ConnectModeEnum],
  [ByVal CreateOptions as RecordCreateOptionsEnum],
  [ByVal Options as RecordOpenOptionsEnum],
  [ByVal UserName as String],
  [ByVal Password as String]
)
```

(continued)

```
with iAppt
   .Fields("DAV:contentclass").Value = _
      "urn:content-classes:trainingevent"
   .Fields(strSchema & "instructoremail").Value = _
      Cstr(Request.Form("email"))
   .Fields(strSchema & "prereqs").Value = CStr(Request.Form("prereqs"))
   .Fields(strSchema & "seats").Value = CStr(Request.Form("seats"))
   .Fields(strSchema & "authorization").Value = _
      Cstr(Request.Form("authorization"))
   .Fields(strSchema & "category").Value = cStr(Request.Form("category"))

   .Fields("http://schemas.microsoft.com/mapi/proptag/" & _
      "x001A001E").Value = "IPM.Appointment"
   .Fields.Update
end with

iAppt.DataSource.SaveToContainer strScheduleFolderPath
```

SaveToObject method

The *SaveToObject* method allows you to save data to a run-time object rather than to a data source. *SaveToObject* works the same way as the *OpenObject* method, except that you're saving information rather than opening it.

CDO Messaging Tasks

In this section, we'll look at some of the most common CDO messaging tasks you can perform. But first you need to know how MIME works, since CDO leverages MIME to read and store content.

The MIME specification divides a message body into parts separated by boundary tags. This enables a mail reader to discern where the logical breaks between parts occur. The message body parts can contain child parts or data. MIME body parts can have one of two content types: singular parts or multiple parts. The nice thing about CDO is that, unless you really want to, you don't have to deal with the MIME stream itself; CDO provides an easy object model to manipulate MIME. By supporting MIME, CDO allows you to send complex messages, such as embedded messages, as well as messages that contain HTML pages. The following is an example of a MIME message:

```
From: "Thomas Rizzo" <thomriz@microsoft.com>
To: "Stacy" <stacy@test.com >
Subject: Text and HTML Message
Date: Tue, 7 Mar 2000 3:32:48 -0700
MIME-Version: 1.0
Content-Type: multipart/alternative; boundary="----=_123"
------=_ 123
Content-Type: text/plain; charset="iso-8859-1"
Content-Transfer-Encoding: quoted-printable
This is a multipart/alternative text & html message.
```

```
------=_ 123
Content-Type: text/html; charset="iso-8859-1"
Content-Transfer-Encoding: quoted-printable
<HTML>
<BODY>This is a multipart/alternative text &amp; html message.</FONT>
</BODY></HTML>
------=_123--
```

Now we're ready to discuss the various CDO messaging tasks.

Sending a Standard Message

Sending an e-mail using CDO is straightforward. You simply create the CDO message object, address the message, set the subject and body, and then send the message. We'll look at some of the more complex tasks you can perform with CDO messages in a moment. The code for sending a simple message is shown here:

```
set oMsg = createobject("CDO.message")
oMsg.To = "stacy@test.com"
oMsg.From = "bob@test.com"
oMsg.Subject = "Hello world!"
oMsg.AutoGenerateTextBody = True
oMsg.MimeFormatted = True
oMsg.HTMLBody = "<HTML><BODY>This is HTML!</BODY></HTML>
oMsg.Send
```

Sending an MHTML Message

In addition to sending a simple HTML message, CDO allows you to send a message with an entire Web page embedded in it. The MHTML standard enables you to take HTML content, convert it into MIME, and embed it in an e-mail message. In CDO, creating an MHTML message is as easy as calling a single method: *CreateMHTMLBody*. This method takes the following parameters: the URL of the Web resource you want to embed; flags that specify any content you don't want to embed, such as sounds or images; and if the Web site that you're embedding requires authentication, a username and password. The following code embeds the Microsoft Exchange Web site into an e-mail and mails it:

```
Set oMsg = CreateObject("CDO.Message")
oMsg.To = "test@test.com"
oMsg.Subject = "Exchange Web site"
oMsg.CreateMHTMLBody "http://www.microsoft.com/exchange"
oMsg.Send
```

Adding an Attachment

CDO makes adding attachments easy too. CDO supports an *Addattachment* method, which takes as a parameter a URL for the resource you want to add, and if that resource requires authentication, a username and password. If successful, CDO will

return to you the MIME body part that corresponds to the new attachment. Adding an attachment is shown here:

```
Dim oMsg as New CDO.Message
Dim oBp as CDO.IBodyPart
Set oBp = oMsg.AddAttachment("http://www.microsoft.com/myfile")
Set oBp = iMsg.AddAttachment("c:\docs\my.doc")
Set oBp = iMsg.AddAttachment("file://mypublicshare/docs/mydoc.doc")
iMsg.Send
```

Adding Mail Headers

You can now access mail headers directly from CDO. Most of the properties that you'll want to access in the mail header already are exposed as top-level properties in CDO. For example, you could look in the mail header to see who a message is from and who it will be sent to. However, CDO already has two properties that perform this service for you: *From* and *To*. You instead might want to access the mail headers for a message you're having a problem with, if CDO doesn't provide an object for the header you're interested in or if you want to get and set custom headers. The following code sets some built-in and custom headers in an e-mail message:

```
Dim oMsg as New CDO.Message
Dim oFlds as ADODB.Fields
Set oFlds = oMsg.Fields

With oFlds
   .Item("urn:schemas:httpmail:to") = "test@test.com"
   .Item("urn:schemas:httpmail:from") = "stacy@test.com"

   .Item("urn:schemas:mailheader:mycustomheader") = "test"

   .Update
End With
```

Resolving Addresses

Before adding addresses to the *To*, *CC*, or *BCC* properties on a CDO Message object, you might want to resolve the address against a directory or the Contacts folder to make sure the address is correct. The way to resolve addresses is to use the CDO Addressee object. This object has a number of properties such as the *DisplayName* or *EmailAddress* properties that you can fill out to try and resolve an address. The names of these properties clue you in to what values should go in them. Once you create the Addressee object and fill in one of the two properties described above, you can call the *CheckName* method, which takes either an LDAP path to your Active Directory or the file path to a Contacts folder contained in Exchange Server 2000.

If the address can be resolved without ambiguous names, you can use some other properties on the Addressee object to get more information about the person. For example, the *DirURL* property returns either the Active Directory path or the File

path to a contact, depending on whether you are validating the address against the Active Directory or the Contacts folder. You can also call the *ResolvedStatus* property, which will contain a *0* for unresolved, *1* for resolved, and *2* for ambiguous. If you get an ambiguous address, you can use the *AmbiguousNames* property to return the Addressees collection, which is made of Addressee objects that are ambiguous with respect to your current Addressee object.

The following sample shows you how to use the properties and methods just described:

```
Dim oAddressee as New CDO.Addressee

'Use the Display Name
oAddressee.DisplayName = "Thomas Rizzo"

'Use the Active Directory

oAddressee.CheckName "LDAP://thomriznt5srv"
if oAddressee.ResolvedStatus = 1 then
   'Resolved
   msgbox oAddressee.DirURL & vblf & oAddressee.EmailAddress
end if

'Use the E-mail Address

Dim oAddressee2 as New CDO.Addressee
oAddressee2.EmailAddress = "thomriz@thomriznt5dom.microsoft.com"

'Use a Contact folder

oAddressee2.CheckName "file://./backofficestorage/" & _
   "thomriznt5dom.microsoft.com/public folders/contacts"
if oAddressee2.ResolvedStatus = 2 then
   'Ambiguous
   set oAddressees = oAddressee2.AmbiguousNames
   for each otmpAddressee in oAddressees
      'Scroll through the ambiguous addressees
   msgbox otmpAddressee.EmailAddress
      next
   end if
```

CDO Calendaring Tasks

The new version of CDO provides some invaluable calendaring features. For example, if you have the correct permissions, you can open other users' calendars and perform operations on those calendars. Also, you can use public folder calendars. Plus, CDO directly supports iCalendar, which allows you to send meeting requests in a standard format that other clients can understand. The following sections show you the most common tasks you'll perform with the CDO calendaring component.

Creating Appointments

Creating appointments by using CDO is as easy as creating a new CDO Appointment object, specifying some properties, and using the *Datasource* interface on the Appointment object to save the appointment into a folder. This process is shown in the following sample:

```
Dim oAppt As New Appointment
sCalendarURL = "file://./backofficestorage/thomriz.com/MBX/User/Calendar/"

'Set the appointment properties
oAppt.StartTime = #2/14/2000 12:30:00 PM#
oAppt.EndTime = #2/14/2000 1:30:00 PM#
oAppt.Subject = "Shop for Valentine's Day present!"
oAppt.Location = "Gift Store"
oAppt.TextBody = "Don't forget this year!!!!"

'Save the appointment
oAppt.DataSource.SaveToContainer sCalendarURL
```

Creating Meeting Requests and Checking Free/Busy Information

In addition to creating appointments in a calendar, you can create meeting requests. To do so, create an appointment and then add the names of the desired attendees. When generating a meeting request, you can also retrieve the free/busy information for all attendees. The following code creates a meeting request, finds the first available slot of free/busy time for an attendee, and sends the meeting request to the attendee:

```
Dim oAppt As New CDO.Appointment
Dim oAttendee As CDO.Attendee
Dim oAddressee As New CDO.Addressee

oAddressee.EmailAddress = "bobw@thomriznt5dom.extest.microsoft.com"
'Check the address against the local directory
'You could also specify any LDAP path
oAddressee.CheckName ("LDAP://thomriznt5dom.extest.microsoft.com")

If oAddressee.ResolvedStatus = cdoResolved Then
   'Create an hour meeting sometime on March 7th
   'Get the free/busy information.
   strFB = oAddressee.GetFreeBusy(CDate("3/7/2000 9:00:00 AM"), _
      CDate("3/7/2000 5:00:00 PM"), 60)
   'Returns a string of 0,1,2,3
   '0 - Free, 1 - Tenative, 2 - Busy, 3 - OOF, 4 - No F/B data
   'Find the first free spot by looking for 0
   bFoundFB = False
   For i = 1 To 8
      'Look for a free spot
      If Mid(strFB, i, 1) = 0 Then
```

```
            dStart = DateAdd("h", i, CDate("3/7/2000 8:00 AM"))
            dEnd = DateAdd("h", 1, dStart)
            bFoundFB = True
            Exit For
         End If
      Next

      If bFoundFB = True Then
         Dim oConfig As New CDO.Configuration
         oConfig.Fields("http://schemas.microsoft.com/cdo/" & _
            "configuration/sendemailaddress") = _
            "thomriz@thomriznt5dom.extest.microsoft.com"
         oConfig.Fields("http://schemas.microsoft.com/cdo/" & _
            "configuration/calendarlocation") = _
            "file://./backofficestorage/" & _
            "thomriznt5dom.extest.microsoft.com/MBX/thomriz/calendar"
         oConfig.Fields.Update

         oAppt.Configuration = oConfig
         oAppt.StartTime = dStart
         oAppt.EndTime = dEnd
         oAppt.Subject = "Meeting"
         oAppt.Location = "Your office"
         oAppt.TextBody = "Meeting with you!"
         Set oAttendee = oAppt.Attendees.Add
         oAttendee.Address = oAddressee.EmailAddress
         oAttendee.Role = cdoRequiredParticipant
         Set oCalMsg = oAppt.CreateRequest
         oCalMsg.Message.Send

         strCalendarURL = _
            "file://./backofficestorage/" & _
            "thomriznt5dom.extest.microsoft.com/MBX/thomriz/calendar"
         'Save to organizer calendar
         oAppt.DataSource.SaveToContainer strCalendarURL
      End If
   End If
```

Notice that the code uses the Addressee object. This object allows you to create and resolve addresses using LDAP with a directory server. Once the attendee is resolved, the free/busy information is retrieved for the attendee by using the *GetFreeBusy* method. This method takes the start time, end time, and the interval in minutes that you want to break the free/busy information into.

The code then checks for the first time when the attendee is free. Next, the code creates an appointment, fills out the appropriate fields, and then calls the *CreateRequest* method. This method returns a CalendarMessage object. You can then call the contained Message object's methods, such as *Send*. Finally, the code then saves the appointment to the calendar.

Creating Recurring Meeting Requests

To create a recurring meeting request, you simply add a RecurrencePattern object to the appointment and set some properties on it. If you want to get more complex, you can add an Exception object to the RecurrencePattern object to specify that a certain date should be excluded from the meeting request. The following code shows how to create a simple recurring meeting request:

```
Dim oAppt As New CDO.Appointment
Dim oAttendee As CDO.Attendee
Dim oAddressee As New CDO.Addressee
Dim oRP As CDO.IRecurrencePattern

oAddressee.EmailAddress = "bobw@thomriznt5dom.extest.microsoft.com"
'Check the address against the local directory.
'You could also specify any LDAP path.
oAddressee.CheckName ("LDAP://thomriznt5dom.extest.microsoft.com")


If oAddressee.ResolvedStatus = cdoResolved Then
   Dim oConfig As New CDO.Configuration
   oConfig.Fields("http://schemas.microsoft.com/cdo/" & _
      "configuration/sendemailaddress") = _
      "thomriz@thomriznt5dom.extest.microsoft.com"
   oConfig.Fields("http://schemas.microsoft.com/cdo/" & _
      "configuration/calendarlocation") = _
      "file://./backofficestorage/" & _
      "thomriznt5dom.extest.microsoft.com/MBX/thomriz/calendar"
   oConfig.Fields.Update

   oAppt.Configuration = oConfig
   oAppt.StartTime = dStart
   oAppt.EndTime = dEnd
   oAppt.Subject = "Meeting"
   oAppt.Location = "Your office"
   oAppt.TextBody = "Meeting with you!"
   Set oRP = oAppt.RecurrencePatterns.Add("ADD")
   'Make it weekly
   oRP.Frequency = cdoWeekly
   oRP.Interval = 1
   oRP.Instances = 10

   Set oAttendee = oAppt.Attendees.Add
   oAttendee.Address = oAddressee.EmailAddress
   oAttendee.Role = cdoRequiredParticipant
   Set oCalMsg = oAppt.CreateRequest
   oCalMsg.Message.Send
```

```
   strCalendarURL = _
      "file://./backofficestorage/" & _
      "thomriznt5dom.extest.microsoft.com/MBX/thomriz/calendar"
   'Save to organizer calendar
   oAppt.DataSource.SaveToContainer strCalendarURL
End If
```

The code creates the RecurrencePattern object by using the RecurrencePatterns collection on the Appointment object. The *Add* method of this collection has only two values that you can pass to its parameter: *ADD* and *DELETE*. Once the new RecurrencePattern object has been added to the collection, the code sets the properties on the object to make the recurrence weekly, specifying only 10 recurrences. If you do not specify the instances, CDO will make the meeting occur indefinitely.

Creating Exceptions

You can create exceptions to your recurring meetings or appointments by using the CDO Exceptions and the Exception object. I'll show you a simple example here of adding a single Exception to a recurring appointment, but you can do more complex operations with the CDO Exception object. The following code sample uses the CDO Exceptions collection to add a new exception to a recurring meeting. You can also delete and modify exceptions using the CDO Exceptions collection.

```
Set oException = oAppt.Exceptions.Add("Add")
oException.StartTime = "5/2/2000 10:00AM"
oException.EndTime = "5/2/2000 11:00 AM"
```

Responding to a Meeting Request

You can use CDO calendaring to accept, decline, or mark tentative a meeting request. Since this operation is quite easy, I'll just show you the code. This code sample assumes that the meeting request you want to process is in your Inbox and is already assigned to the oAppt object:

```
Set oMsg = oAppt.Accept
'You can then modify the response message by modifying oMsg
oMsg.Message.Send
'Save the appointment to your calendar
oAppt.DataSource.SaveTo URLTOYOURCALENDAR
```

Opening Other User's Mailbox Folders

To open another user's folder or folders, all you need to do is pass in the path to that folder either to a Record object or to a query that returns an ADO Recordset. The following example opens up Neil Charney's Calendar folder. To open another user's folder, you must have permissions on that user's folders.

```
Dim oRecord as New ADODB.Record
oRecord.Open "file://./backofficestorage/domain/MBX/neilc/calendar/"
```

Working with Time Zones

One the first issues you have to deal with when working with CDO calendaring functionality, and even when working with ADO, is the storage of dates in UTC in Exchange. CDO will offset the UTC date to the local time zone specified in the CDO Configuration object. If you do not explicitly set a time zone, CDO will use the time zone of the machine. This functionality is different from the way in which ADO handles time zones; ADO does no offset at all and will return the date in UTC. You can run into some very strange debugging issues if you forget this fact—if you create event registrations or calendar appointments using ADO and then look at those items through CDO, you will see different dates.

Time zones play an important role in the Training application when setup creates event registrations for the survey and creating course notification events. To use ADO to create the event registrations so that the events fire daily at 10 PM local time on the server, the application has to figure out the offset from 10 PM local time to UTC time. The following code does this by leveraging CDO to create an appointment at 10 PM local time. The code then retrieves that appointment using ADO, which returns the start time for the appointment at UTC, not at 10 PM local time. The code then figures out the offset between UTC and local time on the server by using the *DateDiff* function. This information is used by the application event handlers to make sure that the ADO query to find all items created in the last 24 hours finds those items with the correct date and time. You need to figure out the UTC offset because Exchange stores the creation dates of items in UTC. Otherwise, you will not really query for items created in the past 24 hours in local time.

```
'Figure out start time for timer events.
'Since the start time needs to be UTC, we need to figure
'out the local server time and then figure out the UTC
'time to tell the event to fire so that  it is 10 PM.

Dim oAppt As CDO.Appointment
Set oAppt = CreateObject("CDO.Appointment")
oAppt.Subject = "Delete me"
If DateDiff("n", Now, Date & " 10:00 PM") < 60 Then
   If DateDiff("n", Now, DateAdd("h", 1, Date & " 10:00 PM")) <= 60 Then
      'We're passing 10 PM for the current day
      dDate = DateAdd("d", 1, Date)
   Else
      dDate = Date
   End If
Else
   dDate = Date
End If

oAppt.StartTime = DateValue(dDate) & " 10:00 PM"
oAppt.EndTime = DateAdd("n", 60, oAppt.StartTime)
strTime = TimeValue(oAppt.StartTime)
```

```
'Dump the appt into the root folder
oAppt.DataSource.SaveToContainer strPath
strdeletehref = oAppt.Fields("DAV:href").Value
Set oAppt = Nothing
'Get it back
Dim oDeleteRecord As New ADODB.Record
oDeleteRecord.Open strdeletehref, , adModeReadWrite
strNow = oDeleteRecord.Fields("urn:schemas:calendar:dtstart").Value

'Figure out the offset from UTC to the local time zone.
'This is needed for the survey notification event handler
'so that it knows how much to offset its query for items created
'during the current day.
strDate = oDeleteRecord.Fields("urn:schemas:calendar:dtstart").Value

strLocDate = dDate & " " & strTime
iDiff = DateDiff("h", strDate, strLocDate)

'Delete it
oDeleteRecord.DeleteRecord
```

CDO Contact Tasks

CDO implements a Person object for working with contacts in both the Exchange Server store and Active Directory. Using this object, you can create and query contact objects in Exchange Server and Active Directory. CDO will handle all the property mapping between Exchange Server and Active Directory. The CDO Person object is very straightforward, so instead of telling you all the properties you can set on this object, I'll just show you some code samples

Creating a Contact in Exchange Server

Creating a contact in Exchange Server is as easy as creating a CDO Person object, setting several properties, and then calling *SaveToContainer*. The following code shows these steps:

```
Dim oPerson As New CDO.Person

strContactURL = _
   "file://./backofficestorage/thomriznt5dom.extest.microsoft.com/" & _
   "public folders/group contacts/"
oPerson.FirstName = "Thomas"
oPerson.LastName = "Rizzo"
oPerson.Company = "Microsoft"
oPerson.WorkStreet = "1 Microsoft Way"
oPerson.WorkCity = "Redmond"
oPerson.WorkState = "WA"
oPerson.WorkPostalCode = "98052"
oPerson.Email = "thomriz@microsoft.com"
```

(continued)

```
oPerson.WorkPhone = "425 555 1212"
oPerson.Email2 = "test@test.com"

'Save the Person objectto the folder
oPerson.DataSource.SaveToContainer strContactURL
```

Creating a Contact in Active Directory

Creating a contact in Active Directory is similar to creating a contact in an Exchange Server folder. The only difference is that you provide an LDAP URL rather than a file URL, as shown here:

```
strContactURL = "LDAP://thomriznt5srv/cn=tomrizzo,cn=users," & _
   "dc=extest,dc=microsoft,dc=com"
oPerson.FirstName = "Thomas"
oPerson.LastName = "Rizzo"
oPerson.Company = "Microsoft"
oPerson.WorkStreet = "1 Microsoft Way"
oPerson.WorkCity = "Redmond"
oPerson.WorkState = "WA"
oPerson.WorkPostalCode = "98052"
oPerson.Email = "thomriz@microsoft.com"
oPerson.WorkPhone = "425 555 1212"
oPerson.Email2 = "test@test.com"

'Save the Person object to the folder
oPerson.DataSource.SaveToContainer strContactURL
```

Saving Your Contact as a vCard

CDO allows you to access or create vCard information for your contacts. vCard is a standard way of describing contact information on the Internet. CDO vCard support allows you to send or save your contacts to any compliant vCard system. You get the vCard information by using the *GetVCardStream* method, which returns the information about a contact in vCard format to an ADO Stream object. The following code uses the *GetVCardStream* method to retrieve the vCard information and print it to the screen for a contact:

```
Dim oPerson as New CDO.Person
oPerson.DataSource.Open "file://./backofficestorage/domain/folder/name.eml"
oPerson.GetVCardStream.SaveToFile "c:\vcard.txt"
```

The output from running the preceding code, which is contained in the text file vcard.txt, is shown here:

```
BEGIN:VCARD
VERSION:2.1
N:Rizzo;Thomas
FN:Thomas Rizzo
ORG:Microsoft Corporation
```

```
TITLE:Product Manager
NOTE;ENCODING=QUOTED-PRINTABLE:=0D=0A
TEL;WORK;VOICE:(425) 555-1212
ADR;WORK:;;1 Microsoft Way;Redmond;WA;98052;United States of America
LABEL;WORK;ENCODING=QUOTED-
PRINTABLE:1 Microsoft Way=0D=0ARedmond, WA 98052=0D=0AUnited States of America
URL:
URL:http://www.microsoft.com/exchange
EMAIL;PREF;INTERNET:thomriz@microsoft.com
REV:20000410T033803Z
END:VCARD
```

CDO Folder Tasks

CDO provides a Folder object that allows you to create or access folders contained in the Exchange database. Using the Folder object, you can retrieve the number of read and unread items contained in the folder. You will interact most with the Folder object's properties. These properties include the following: *Configuration*, *ContentClass*, *DataSource*, *Description*, *DisplayName*, *EmailAddress*, *Fields*, *HasSubFolders*, *ItemCount*, *UnreadItemCount*, and *VisibleCount*.

Most of these properties are straightforward. The only one that really needs explaining is the *VisibleCount* property. This property is the total number of hidden items in the folder. Hidden items are created in the associated Contents table. These items can be created by setting the *DAV:ishidden* property to *True*.

Creating a Folder

To create a folder using the CDO Folder object, you need to first create a CDO Folder object and then use the *Datasource* property to save the folder. The following example shows you how to create a folder using CDO:

```
Dim oFolder As New CDO.Folder

oFolder.Description = "This is my folder"
oFolder.ContentClass = "urn:content-classes:contactfolder"
   oFolder.Fields("http://schemas.microsoft.com/exchange"  & _
   "/outlookfolderclass") = "IPF.Contact"
oFolder.Fields.Update
oFolder.DataSource.SaveTo "file://./backofficestorage/"  & _
   "thomriznt5dom.extest.microsoft.com/public folders/my contact folder"
```

Mail-Enabling Folders

With the addition of new top-level hierarchies, Exchange Server 2000 does not, by default, mail-enable folders in the new hierarchies. Therefore, you will want to mail-enable the folders either through the administrative UI or programmatically. To programmatically mail-enable a folder is quite easy. All you need to do is use the

IMailRecipient interface from the CDO for Exchange Management library. To retrieve the interface, set a variable to the *IMailRecipient* interface. Then set the value of that variable to be your CDO folder. Call the *MailEnable* method on the folder. Make the changes to the properties on the *IMailRecipient* interface. For example, set up the alias, establish the smtp e-mail address, and decide whether the new address should be hidden from the address book. Once you are finished setting the properties, just save the folder back to the Exchange database. Remember that you will need to open the folder in the read/write mode using the CDO Folder object, as shown in the following code:

```
Dim oFolder As New CDO.Folder
Dim oRecip As CDOEXM.IMailRecipient

oFolder.DataSource.Open "file://./backofficestorage/" & _
   "thomriznt5dom.extest.microsoft.com/public folders/my contact folder", _
   , adModeReadWrite
Set oRecip = oFolder
oRecip.MailEnable
oRecip.SMTPEmail = "contacts@domain.com"
oRecip.HideFromAddressBook = False
oRecip.Alias = "My Contacts Folder"
oFolder.DataSource.Save
```

Chapter 19

Putting It All Together

Using XML with Exchange Server

One of the biggest buzzwords in the industry has been XML. XML provides an easy way to describe data and share it among applications. Microsoft Exchange Server 2000 directly supports XML, making it a great Web development platform. In other words, you can use XML to get and set data directly in Exchange Server.

I couldn't possibly attempt to cover all aspects of XML in this chapter. However, many good books on the topic exist. Instead, I'll discuss how to get, set, search, and format using XSL, XML data in Exchange Server 2000. Since you'll just be querying for data, I won't cover how to create an XML document from scratch.

XMLHTTP Component

You can retrieve XML data from Exchange 2000 in a number of different ways. The technique you'll probably use the most is to employ the Web Distributed Authoring and Versioning (WebDAV) protocol. WebDAV is an extension to the http protocol that specifies how to perform file processing, thus making the Web extremely readable and writable. Using WebDAV commands, you can lock a resource, get a property, or change a property. This is a powerful capability—WebDAV can work through firewalls and proxy servers because it piggybacks on http.

You might be wondering how to use to a protocol such as WebDAV within your Web applications. Microsoft Internet Explorer 5 ships with a component called XMLHTTP, to work with the WebDAV protocol. You will still have to send correctly formatted WebDAV requests, but the XMLHTTP component simplifies this. Be aware that you should use the XMLHTTP component only on the client side since the component was built for that environment. If you're writing code on the Exchange server, you should write to OLE DB or ADO.

We'll take a look at the most important properties and methods of the XMLHTTP component so that you can use them in your applications. These properties and methods are *Abort*, *OnReadyStateChange*, *Open*, *ReadyState*, *responseBody*, *responseStream*, *responseText*, *responseXML*, *Send*, *SetRequestHeader*, *Status*, and *StatusText*. All other properties and methods of XMLHTTP are fully documented on the Microsoft Developer's Network (MSDN), *http://msdn.microsoft.com*.

Open Method

The first method that you'll probably use with the XMLHTTP component is the *Open* method. This method takes five parameters: *Method*, *URL*, *Async*, *User*, and *Password*. The *Method* parameter specifies which http method to use: *GET*, *POST*, *PROPPATH*, *SEARCH*, or another method. The *URL* parameter is the absolute URL to the resource—for example, *http://myserver/documents*. The *Async* parameter is a Boolean that specifies whether the call is asynchronous. If set to *True*, which is the default, the call returns immediately. Finally, you can use the *User* and *Password* parameters to pass the username and password with which you want the component to access secured sites. The following code example, taken from the coursexml.asp file in the Training application, shows you how to create an XMLHTTP object and use the *Open* method:

```
request = new ActiveXObject("microsoft.xmlhttp");

//Example propfind
//request.open("PROPFIND", URLSchedule, true);

request.open("SEARCH", URLSchedule, true);
```

SetRequestHeader Method

The *SetRequestHeader* method allows you to specify the name and value for an http header. The most common headers that you will set are depth and content-type. A depth header specifies how deep in a hierarchy the request applies. For example, setting the depth header to <;$QD>*1, noroot*<;$QD> means that the method will be applied to all the children of the specified URL, but not to the URL itself. If you do not specify a depth, Exchange will default to an infinite depth. A content-type header specifies the content type you plan to send to the server. The following code snippet shows you how to use this method:

```
//For propfind you can select a depth
//request.setRequestHeader("depth", "1,noroot");
request.setRequestHeader("Content-type", "text/xml");
```

ReadyState Property

The *ReadyState* property contains the state of the request object. This property can contain one of the five values shown in Table 19-1.

Value	*Description*
UNINITIALIZED	The object has been created, but the *Open* method has not been called.
LOADING	The object has been created, but the *Send* method has not been called.
LOADED	The *Send* method has been called, but the response is not yet available.
INTERACTIVE	The object is not fully loaded yet, but the user can already interact with it. Partial results can be viewed in the browser.
COMPLETED	All data has been received and can be viewed in the browser.

Table 19-1. *Values of the* ReadyState *property.*

OnReadyStateChange Property

This property allows you to specify which event handler to call when the *ReadyState* property changes. The following code sets the function to be called to *dostatechange*. We'll see what *dostatechange* does in a moment, when we look at the *responseXML* property.

```
request.onreadystatechange = dostatechange;
```

Send Method

As you would guess, this method sends the request to the server. You specify the request as a string parameter to this method. The following code sends the body of our request to the server. We'll see what this body looks like later in the chapter, when we examine WebDAV requests.

```
request.send(body);
```

responseBody, *responseStream*, *responseText*, and *responseXML* Properties

All of these properties allow you to retrieve the response in each format specified. You'll most commonly use *responseText* and *responseXML* in your applications. The *responseText* property returns the XML response as a text string. The *responseXML* property returns the response as an XML document parsed by the XML Document Object Model (XMLDOM), meaning you'll receive an object that you can format using

XSL or you can call methods on the XMLDOM to interact with the data. You'll use these properties in the event handler you specified for the *onreadystatechange* property. The following code gets the XML response and sends it to be formatted by XSL:

```
thexml = request.responseXML;
   datediv.innerHTML = "<B>Showing Courses from <%=dDateStart%>
      to <%=dDateEnd%></B><BR>"
   //Check for empty body
   if (thexml.selectSingleNode("a:multistatus/a:response") == null) {
      msgdiv.innerHTML = "<B>No courses found.</B>";
      request = null;
      return;
   }
   //For debugging purposes
   //alert(request.responseText);
   msgdiv.innerHTML = thexml.transformNode(reportXSL.documentElement);
```

Status and *StatusText* Properties

Both of these properties contain status information about the request. *Status* contains the http status returned by the request. This is an integer that corresponds to one of the http status codes. *StatusText* returns a string that represents the status returned by the request. The following code checks to see whether the *Status* property has a value of 207, which indicates that the request was successful. If *Status* does not have a value of 207, the code prints out the values of the *Status* and *StatusText* properties as an error.

```
if(request.status != 207) {
   msgdiv.innerText = "Error, status = " + request.status + " " + request.statusText;
   msgdiv.style.fontFamily = "verdana"
```

Abort Method

The *Abort* method will cancel the current http request and restore the XMLHTTP component to the *UNINITIALIZED* state.

WebDAV Commands

Now that you know about the object model of XMLHTTP, which allows you to send commands to the Exchange server, you're probably wondering what these commands are. WebDAV supports a number of commands, including MKCOL, PROPPATCH, PROPFIND, DELETE, MOVE, COPY, SEARCH, LOCK, UNLOCK, SUBSCRIBE, and POLL. Each of these commands serves a distinctive purpose in your applications. For example, MKCOL allows you to create a collection (or a folder) on your server. PROPPATCH and PROPFIND allow you to set and get properties on resources. Let's look at some typical tasks you'll perform with WebDAV so that you can see each of these commands in action.

Creating Folders

To create a folder using WebDAV, you need to issue the MKCOL command and pass the URL of the new folder that you want to create. If the creation is successful, you'll receive the Status of *201* and the StatusText of *Created*. The following code uses XMLHTTP to create a new folder:

```
<HTML>
<BODY>

<SCRIPT LANGUAGE=javascript>
<!--
   var strURL = "http://localhost/public/my new folder";
   var request = new ActiveXObject("Microsoft.XMLHTTP");
   request.open("MKCOL",strURL,false);
   request.send();
   alert(request.status + " " + request.statustext);
//-->
</SCRIPT>


</BODY>
</HTML>
```

Creating Items

To create an item, you'll use the http *PUT* method and pass to it the URL of the new item that you want to create. If successful, you'll get the Status of *201* and the StatusText of *Created*. The following code creates a new post in the Public Folder we created earlier:

```
<HTML>
<BODY>

<SCRIPT LANGUAGE=javascript>
<!--
   var strURL = "http://localhost/public/my new folder/my new item.eml/";
   var request = new ActiveXObject("Microsoft.XMLHTTP");
   request.open("PUT",strURL,false);
   request.setRequestHeader("Translate","f");
   request.send();
   alert(request.status + " " + request.statustext);
//-->
</SCRIPT>


</BODY>
</HTML>
```

Copying Folders and Items

To copy both items and folders, use the COPY command. This command takes as a parameter the source URL of the copied item. As part of your request headers, you need to specify the destination URL for the copied item. If you are copying a folder, all the contents of that folder will be copied as well. The following code copies the newly created folder and item from the earlier example into another folder. If successful, you should receive the Status value of *201*.

```
<HTML>
<BODY>

<SCRIPT LANGUAGE=javascript>
<!--
   var strSourceURL = "http://localhost/public/my new folder/";
   var strDestURL = "http://localhost/public/my new folder 2/";
   var request = new ActiveXObject("Microsoft.XMLHTTP");
   request.open("COPY",strSourceURL,false);
   request.setRequestHeader("Destination",strDestURL);
   request.send();
   alert(request.status + " " + request.statustext);
//-->
</SCRIPT>


</BODY>
</HTML>
```

Moving Folders and Items

Moving folders is as easy as copying them. All you need to do is change the command from COPY to MOVE. The following code shows you how to move a folder:

```
<HTML>
<BODY>

<SCRIPT LANGUAGE=javascript>
<!--
   var strSourceURL = "http://localhost/public/my new folder/";
   var strDestURL = "http://localhost/public/my new folder 3/";
   var request = new ActiveXObject("Microsoft.XMLHTTP");
   request.open("MOVE",strSourceURL,false);
   request.setRequestHeader("Destination",strDestURL);
   request.send();
   alert(request.status + " " + request.statustext);
//-->
</SCRIPT>


</BODY>
</HTML>
```

Deleting Items and Folders

To delete items or folders, you use the DELETE command and pass to it the URL of the item to be deleted. If successful, you'll receive the Status of *200* and a StatusText of *OK*. The following code deletes a folder:

```
<HTML>
<BODY>

<SCRIPT LANGUAGE=javascript>
<!--

   var strURL = "http://localhost/public/my new folder 3/";
   var request = new ActiveXObject("Microsoft.XMLHTTP");
   request.open("DELETE",strURL,false);
   request.send();
   alert(request.status + " " + request.statustext);
//-->
</SCRIPT>


</BODY>
</HTML>
```

Setting Properties

To set properties, you'll need to use the PROPPATCH command. To use this command, you must create an XML document to send that will list the properties you want updated. You can generate this XML document in a number of ways. The two easiest ways are to use the XMLDOM, or to simply generate the XML yourself by using JavaScript code. The following example generates the XML directly and sends it to the server to update the properties:

```
<HTML>
<BODY>

<SCRIPT LANGUAGE=javascript>
<!--

   var strURL = "http://localhost/public/my new folder2/my new item.eml/";
   request = new ActiveXObject("microsoft.xmlhttp");
   request.open("PROPPATCH", strURL, false);
   request.setRequestHeader("Content-type", "text/xml");

   proplist  = "<M:subject>My New Message</M:subject>";
   proplist = "<M:textdescription>This is my body</M:textdescription>";

   body = "<?xml version='1.0'?>";
   body += "<D:propertyupdate xmlns:D='DAV:' xmlns:M='urn:schemas:httpmail'>";
```

(continued)

```
    body += "<D:set><D:prop>";
    body += proplist;
    body += "</D:prop></D:set>";
    body += "</D:propertyupdate>";

    request.send(body);
    alert(request.status + " " + request.statustext);

//-->
</SCRIPT>

</BODY>
</HTML>
```

Retrieving Properties

To retrieve properties, you need to use the PROPFIND command. PROPFIND can request a single property, all properties, or all property names. When working with PROPFIND, you need to set the depth header value to the depth you want the request's scope to be. The value for depth can be *0*, which indicates only the entity in the URL specified; *1*, which signifies the target URL and any of its immediate children; or *Infinity*, which indicates the target URL, its children, and its children's children, all the way down to the leaves of the tree. Also, you need to send an XML document that specifies the property list you want to retrieve from the resource. The following code requests some DAV properties and Microsoft Office properties from a Microsoft Word document in a Public Folder:

```
<HTML>
<BODY>
<TEXTAREA rows=10 cols=20 id=textarea1 name=textarea1>

</TEXTAREA>
<SCRIPT LANGUAGE=javascript>
<!--
    var strURL = "http://localhost/public/my new folder/my word doc.doc/";

    request = new ActiveXObject("microsoft.xmlhttp");
    request.open("PROPFIND", strURL, false);
    request.setRequestHeader("depth", "1,noroot");
    request.setRequestHeader("Content-type", "text/xml");

    proplist = "<D:iscollection/><D:displayname/><D:getlastmodified/>
       <D:creationdate/><O:Author/><O:Manager/><O:Title/>";

    body = "<?xml version='1.0'?>";
    body += "<D:propfind xmlns:D='DAV:'
       xmlns:O='urn:schemas-microsoft-com:office:office'>";
    body += "<D:prop>";
```

```
    body += proplist;
    body += "</D:prop>";
    body += "</D:propfind>";


    request.send(body);

    alert(request.status + " " + request.statustext);
    textarea1.value = request.responseText;

//-->
</SCRIPT>


</BODY>
</HTML>
```

The returned XML stream that is placed into the TEXTAREA follows. Notice that it is a multistatus response.

```
<?xml version="1.0"?><a:multistatus xmlns:b="urn:uuid:c2f41010-65b3-
11d1-a29f-00aa00c14882/" xmlns:c="xml:" xmlns:d="urn:schemas-microsoft-
com:office:office" xmlns:a="DAV:"><a:response><a:href>http://localhost/
public/my%20new%20folder/my%20word%20doc.doc</a:href><a:propstat>
<a:status>HTTP/1.1 200 OK</a:status><a:prop><a:iscollection b:dt="boolean">
0</a:iscollection><a:displayname>my word doc.doc</a:displayname>
<a:getlastmodified b:dt="dateTime.tz">2000-03-09T10:33:04.532Z
</a:getlastmodified><a:creationdate b:dt="dateTime.tz">2000-03-
09T10:33:04.372Z</a:creationdate><d:Author>Thomas Rizzo</d:Author>
<d:Manager>Some Manager</d:Manager><d:Title>This is my word
doc</d:Title></a:prop></a:propstat></a:response></a:multistatus>
```

To retrieve all properties in the DAV namespace, you would issue the following WebDAV command:

```
<?xml version="1.0" ?>
<D:propfind xmlns:D="DAV:">
   <D:allprop/>
</D:propfind>
```

Locking a Resource

You might want to lock a resource, such as a file or collection so that no other WebDAV requests can access it. You can acquire an exclusive or shared lock on a resource. Each lock has a timeout, affording you a precise window of opportunity to make your changes before the lock expires. If a lock does expire, you can always request another lock on the resource from the server. The following code requests an exclusive write lock on a Word document in a Public Folder. As you'll see in the code, the lock remains in effect for 3,000 seconds, and the XML body holds an *Owner*

property. If another application tries to modify the Word document, it will receive an XML document that contains a *lockdiscovery* property, which contains the *Owner* property of the person who currently has a lock on the resource. The application can use this property to request that the current lock owner disable the lock or notify the application when the lock is released.

```
<HTML>
<BODY>
<TEXTAREA rows=10 cols=80 id=textarea1 name=textarea1>

</TEXTAREA>
<SCRIPT LANGUAGE=javascript>
<!--

   var strURL = "http://localhost/public/my new folder/my word doc.doc/";
   request = new ActiveXObject("microsoft.xmlhttp");
   request.open("LOCK", strURL, false);
   request.setRequestHeader("Content-type", "text/xml");
   request.setRequestHeader("timeout", "Second-3000");


   body = "<?xml version='1.0'?>";
   body += " <a:lockinfo xmlns:a='DAV:'>"
   body += "<a:lockscope><a:exclusive /></a:lockscope>";
   body += "<a:locktype><a:write /></a:locktype>";
   body += "<a:owner><a:href>mailto:thomriz</a:href>";
   body += "</a:owner></a:lockinfo>";

   request.send(body);
   alert(request.status + " " + request.statustext);
   textarea1.value = request.responseText;




//-->
</SCRIPT>

</BODY>
</HTML>
```

After this request is sent to the server, the following response will be received:

```
<?xml version="1.0"?><a:prop xmlns:a="DAV:"><a:lockdiscovery>
<a:activelock><a:locktype><a:write/></a:locktype><a:lockscope>
<a:exclusive/></a:lockscope><a:owner xmlns:a="DAV:"><a:href>
mailto:thomriz</a:href></a:owner><a:locktoken><a:href>
opaquelocktoken:9641CB50-729A-4966-B904-6F55773AA5B7:
106544051121029120O1</a:href></a:locktoken><a:depth>infinity
```

```
</a:depth><a:timeout>Second-3000</a:timeout></a:activelock>
</a:lockdiscovery></a:prop>
```

If the lock is successful, you'll receive the Status of *200 OK*, along with the XML just shown. The most important property to be aware of is the *locktoken* property. The *locktoken* property uniquely identifies your lock and must be used in future requests. Because http is stateless, you need to pass the *locktoken* with your future requests so that the server knows who is attempting to write to the resource. You'll also need this property for the PUT, PROPPATCH, and other requests you send, and to unlock the file.

> **NOTE** If the resource is already locked when you try to lock it, you will receive the Status of *423 Locked.*

Unlocking a Resource

Unlocking a resource is easy if you have a unique lock token. Simply send the UNLOCK command to the server and add a header that contains your lock token. If successful, the server will return the Status of *204 No Content.* This means the command completed successfully but the server had no text to return except for the status. The following code uses a specific lock token to unlock a resource:

```
<HTML>
<BODY>
<SCRIPT LANGUAGE=javascript>
<!--

   var strURL = "http://localhost/public/my new folder/my word doc.doc/";
   request = new ActiveXObject("microsoft.xmlhttp");
   request.open("UNLOCK", strURL, false);
   request.setRequestHeader("Lock-Token",
      "<opaquelocktoken:9641CB50-729A-4966-B904-
      6F55773AA5B7:10582347518064984065>");
   request.send();
   alert(request.status + " " + request.statustext);
//-->
</SCRIPT>

</BODY>
</HTML>
```

Subscribing to a Resource

You can use WebDAV to subscribe to a resource. As a subscriber, you can receive notification about changes to a resource in one of two ways: you can have the server inform you when the resource changes, or you can poll the server for any changes to the resource. To create a subscription, use the SUBSCRIBE command and pass the URL you want to subscribe to, as in the following example. This example also sets the timeout for a subscription.

```
<HTML>
<BODY>
<SCRIPT LANGUAGE=javascript>
<!--

   var strURL = "http://localhost/public/my new folder/my word doc.doc/";
   request = new ActiveXObject("microsoft.xmlhttp");
   request.open("SUBSCRIBE", strURL, false);
   request.setRequestHeader("Subscription-lifetime", 1000);

   request.send();
   alert(request.status + " " + request.statustext);

//-->
</SCRIPT>

</BODY>
</HTML>
```

If your subscription is successful, the server will return a Subscription-id, which you should keep because you'll need to pass it when you later poll the server or unsubscribe from the resource.

Polling the Server

To poll the server to see whether any of the resources you've subscribed to have changed, you need to use the POLL command. This command requires that you pass the Subscription-id that the SUBSCRIBE command gave you when you subscribed to the resource. The following code checks whether anything has changed on a resource. If no changes were made to the resource, the server will return a 204 status code. In addition to polling, you can listen on a TCP/IP port for a UDP notification of changes. See the sample included on the companion CD to learn how to use this technique.

```
<HTML>
<BODY>
<SCRIPT LANGUAGE=javascript>
<!--

   var strURL = "http://localhost/public/my new folder/my word doc.doc/";
   request = new ActiveXObject("microsoft.xmlhttp");
   request.open("POLL", strURL, false);
   request.setRequestHeader("Subscription-id", "SomeID");

   request.send();
   alert(request.status + " " + request.statustext);

//-->
</SCRIPT>

</BODY>
</HTML>
```

Querying with WebDAV SEARCH

One of the neat things that you can do with WebDAV is perform SQL syntax queries against Exchange Server and have your results formatted as XML. Having your query returned to the client as XML is a very powerful capability because it allows you to perform client-side formatting with XSL. Furthermore, since the XML is already on the client, you can re-sort the data very quickly. The following code is taken from the coursexml.asp file in the Training application. This code sends a SEARCH request to the Exchange server and receives the data from the server as XML.

```
request = new ActiveXObject("microsoft.xmlhttp");

//Example propfind
//request.open("PROPFIND", URLSchedule, true);

request.open("SEARCH", URLSchedule, true);

//For propfind you can select a depth
//request.setRequestHeader("depth", "1,noroot");
request.setRequestHeader("Content-type", "text/xml");

proplist = "<D:iscollection/><D:displayname/>
   <D:getlastmodified/><D:creationdate/>
   <C:instructoremail/><CAL:location/><O:Author/>
   <O:Manager/><O:Title/><H:subject/>";

//You can also do a propfind to find specific properties
//body = "<?xml version='1.0'?>";
//body += "<D:propfind xmlns:D='DAV:'
//xmlns:O='urn:schemas-microsoft-com:office:office'
//xmlns:C='<%=strSchema%>' xmlns:CAL='urn:schemas:calendar:'
//xmlns:H='urn:schemas:httpmail:'>";

body = "<searchrequest xmlns='DAV:'>";
body += "<sql>";
body += "SELECT \"<%=strSchema%>materialshttppath\" as materialshttppath,"
body += "\"<%=strSchema%>overallscore\" as
   overallscore,\"<%=strSchema%>rating\" as rating,";
body += "\"<%=strSchema%>materialsfilepath\" as
   materialsfilepath,\"<%=strSchema%>surveycount\" as surveycount,";
body += " \"<%=strSchema%>discussionurl\" as
   discussionurl,\"<%=strSchema%>prereqs\" as prereqs,";
body += "\"urn:schemas:httpmail:textdescription\" as
   description,\"<%=strSchema%>category\" as category,";
body += "\"urn:schemas:calendar:dtstart\" as
   starttime, \"urn:schemas:calendar:dtend\" as endtime,";
body += "\"DAV:iscollection\" as iscollection,\"DAV:href\" as
   href,";
```

(continued)

```
body += "\"urn:schemas:httpmail:subject\" as
   subject,\"urn:schemas:calendar:location\" as location,
   \"<%=strSchema%>instructoremail\" as instructoremail
   FROM scope('shallow traversal of \"<%=strURLToSchedule%>\"')
   where \"DAV:ishidden\" = false AND \"DAV:isfolder\" = false";
//Add date restriction
body += " AND \"urn:schemas:calendar:dtstart\" &gt;&eq;
   CAST(\"<%=dISODateStart%>\" as 'dateTime')";
body += " AND \"urn:schemas:calendar:dtstart\" &lt;&eq;
   CAST(\"<%=dISODateEnd%>\" as 'dateTime')";

body += "</sql>";
body += "</searchrequest>";
//For debugging
//alert(body);

//Propfind example
//body += "<D:prop>";
//body += proplist;
//body += "</D:prop>";
//body += "</D:propfind>";

request.onreadystatechange = dostatechange;
msgdiv.innerHTML = "<font face='verdana' size='+1'>Loading...</font>";
request.send(body);
```

The SELECT statement in this code uses column aliasing, which will make it easier to format the data using XSL. You'll see how to use XSL to format the XML data returned later in this section. The following code sample shows the raw XML data returned from this query, illustrating how custom and built-in schema can be queried and returned with XML:

```
<?xml version="1.0"?><a:multistatus xmlns:b="urn:uuid:c2f41010-65b3-11d1-
a29f-00aa00c14882/" xmlns:c="xml:" xmlns:d="urn:schemas-microsoft-com:
office:office" xmlns:a="DAV:"><a:response><a:href>http://thomriznt5srv/
public/140/Training/Schedule/{8C35C44B-68EB-4651-AC3E-5C475923A7A1}
.EML</a:href><a:propstat><a:status>HTTP/1.1 200 OK</a:status><a:prop>
<materialshttppath>http://thomriznt5srv/public/140/Training/Course
Materials/Leveraging XML in Exchange 2000/?Cmd=contents&amp;View=Messages
</materialshttppath><materialsfilepath>file://THOMRIZNT5SRV/Course
Materials140/Leveraging XML in Exchange 2000</materialsfilepath>
<discussionurl>http://thomriznt5srv/public/140/Training/Discussions/
Leveraging XML in Exchange 2000/?Cmd=contents&amp;View=By Conversation
Topic</discussionurl><prereqs>fejio</prereqs><description>fjeoj
</description><category>dev</category><starttime b:dt="dateTime.tz">
2000-03-08T21:00:00.000Z</starttime><endtime b:dt="dateTime.tz">2000-03-
```

```
08T23:00:00.000Z</endtime><iscollection b:dt="boolean">0</iscollection>
<href>http://thomriznt5srv/public/140/Training/Schedule/{8C35C44B-68EB-
4651-AC3E-5C475923A7A1}.EML</href><subject>Leveraging XML in Exchange
2000</subject><location>43</location><instructoremail>thomriz@thomriznt
5dom.extest.microsoft.com</instructoremail></a:prop></a:propstat>
<a:propstat><a:status>HTTP/1.1 404 Resource Not Found</a:status><a:prop>
<overallscore/><rating/><surveycount/></a:prop></a:propstat></a:response>
<a:response><a:href>http://thomriznt5srv/public/140/Training/Schedule/
{75BD5A83-09E7-47B7-A9F1-A75DD62F5BA7}.EML</a:href><a:propstat>
<a:status>HTTP/1.1 200 OK</a:status><a:prop><prereqs>fjoi</prereqs>
<description>fei</description><category>dev</category><starttime
b:dt="dateTime.tz">2000-03-08T18:00:00.000Z</starttime><endtime
b:dt="dateTime.tz">2000-03-08T19:00:00.000Z</endtime><iscollection
b:dt="boolean">0</iscollection><href>http://thomriznt5srv/public/140/
Training/Schedule/{75BD5A83-09E7-47B7-A9F1-A75DD62F5BA7}.EML
</href><subject>CDO and You</subject><location>43</location>
<instructoremail>thomriz@thomriznt5dom.extest.microsoft.com</
instructoremail></a:prop></a:propstat><a:propstat><a:status>HTTP/
1.1 404 Resource Not Found</a:status><a:prop><materialshttppath/>
<overallscore/><rating/><materialsfilepath/><surveycount/><discussionurl/>
</a:prop></a:propstat></a:response></a:multistatus>
```

Persisted Search Folders

When using WebDAV, you can use the WebDAV search methods that we looked at earlier. Exchange 2000 provides the capability to create persisted search folders when using WebDAV. These search folders are like standard folders in that you can use a URL to access them and query them. Search folders can be created in any of your application hierarchies. You cannot, however, create search folders in the MAPI All Public Folders hierarchy. Also, you cannot create search folders using ADO.

Search folders allow you to offload to the server the task of finding new items that meet your SQL search criteria. For example, imagine you have an application that spans 10 folders under the root folder. In each folder, you need to find all the items with a specific property, such as items whose content classes are a certain type—say, *urn:content-classes:mycc*. Rather than querying Exchange every time you need to find items that meet this criterion, you could create a top-level search folder. This search folder would asynchronously add links to new items that meet the criterion (or criteria) you specified in the search folder. Your application could query the search folder rather than perform a deep traversal of all the application folders. Plus, search results are stored and dynamically updated by Exchange without requiring the clients to be connected or having to requery the Exchange database. Search performance should be much greater with a search folder.

Creating a Search Folder

To create a search folder, all you need to do is issue an MKCOL command. The MKCOL command also specifies a *DAV:searchrequest* property that contains the SQL statement you want the search folder to perform. The following example shows how to create a search folder:

```
function SearchFolderCreate( folderURL, SQLQuery ) {
   var oXMLHTTP;
   oXMLHTTP = new ActiveXObject("Microsoft.XMLHTTP");

   oXMLHTTP.Open("MKCOL", folderURL, false);
   strR = "<?xml version='1.0'?>";
   strR += "<d:propertyupdate xmlns:d='DAV:'>";
   strR += "<d:set><d:prop><d:searchrequest><d:sql>" + SQLQuery+ "</d:sql>";
   strR += "</d:searchrequest></d:prop></d:set></d:propertyupdate>";
   oXMLHTTP.SetRequestHeader("Content-type:", "text/xml");

   oXMLHTTP.send(strR);

   if(! Req.Status == "207") { // Multistatus response
      alert("An error has occurred!!");
   }
}
```

If the command is successful, the server returns a 207 Multistatus response.

Search folders are just like regular folders in that they contain properties, but they are different in that they contain properties unique to them. Table 19-2 outlines the special properties for search folders.

Property	*Description*
DAV:resourcetype	If the folder is a search folder, the value of this property will be *<DAV:collection/><DAV:searchresults/>*.
DAV:searchrequest	This property contains the original SQL query for the persisted search. You cannot change this property. If you need to modify your search, you must delete your search folder and re-create it, or create a new search folder.
DAV:searchtype	This property is set to *dynamic* by the Web Storage System.

Table 19-2. *Properties unique to search folders.*

Searching a Persisted Search Folder

To query your search folder, all you need to do is use either the WebDAV Search or *PropFind* methods that we looked at earlier, and then specify the search folder URL.

Optionally, you can specify a Range header in your Search queries to return a certain number of rows. For example, you can specify that you want only the first 10 rows or the last 10 rows. The following examples show you how to use the Range header with XMLHTTP.

```
'Rows 10-20 and 40-50
Req.setRequestHeader "Range", "rows=10-20,40-50"
'Last 10 rows
Req.setRequestHeader "Range", "rows=-10"
'From Row 10 to the end of the resultset
Req.setRequestHeader "Range", "rows=10-"
'Rows 1-10 and the last 10 rows
Req.setRequestHeader "Range", "rows=1-10,-10"
```

Using ADO to Retrieve XML Data from Exchange Server

There are two other ways you can retrieve XML data from Exchange Server. First, you can generate the XML data yourself by using ADO. For example, you could generate an XML document for your data and simply plug the values in this document for the properties from the ADO Fields collection. Not too pretty or easy a technique, although it is functional.

The second way you can retrieve XML data from Exchange Server is to leverage the XML persistence feature in ADO. ADO allows you to both load and save data in an XML format. The XML format must, however, adhere to the structure expected by ADO. To save data as XML, you just call the *Save* method of the Recordset object and pass in a location and *adPersistXML (1)*.

The cool thing about using ADO with XML is that ADO can persist to the file system or directly to the ASP Response object. This means you can either save your Recordset to an XML file or blast the data to the browser in ASP applications. You can also reload a Recordset from a correctly formatted XML document. We'll cover that feature later in the chapter when we talk about deploying the workflow portion of the Training application. The following XML document comes from the Training application data saved by ADO's XML features:

```
<xml xmlns:s='uuid:BDC6E3F0-6DA3-11d1-A2A3-00AA00C14882'
   xmlns:dt='uuid:C2F41010-65B3-11d1-A29F-00AA00C14882'
   xmlns:rs='urn:schemas-microsoft-com:rowset'
   xmlns:z='#RowsetSchema'>
<s:Schema id='RowsetSchema'>
   <s:ElementType name='row' content='eltOnly' rs:updatable='true'>
      <s:AttributeType name='c0' rs:name='urn:schemas:mailheader:subject'
         rs:number='1' rs:nullable='true' rs:write='true'>
         <s:datatype dt:type='string' dt:maxLength='32768'/>
```

(continued)

```
      </s:AttributeType>
      <s:AttributeType name='c1' rs:name='DAV:href' rs:number='2'
         rs:nullable='true'>
         <s:datatype dt:type='string' dt:maxLength='32768'/>
      </s:AttributeType>
      <s:AttributeType name='c2' rs:name='urn:schemas:calendar:dtstart'
         rs:number='3' rs:nullable='true' rs:write='true'>
         <s:datatype dt:type='dateTime' rs:dbtype='filetime'
            dt:maxLength='16' rs:precision='19' rs:fixedlength='true'/>
      </s:AttributeType>
      <s:AttributeType name='c3' rs:name='urn:schemas:calendar:dtend'
         rs:number='4' rs:nullable='true' rs:write='true'>
         <s:datatype dt:type='dateTime' rs:dbtype='filetime'
            dt:maxLength='16' rs:precision='19' rs:fixedlength='true'/>
      </s:AttributeType>
      <s:extends type='rs:rowbase'/>
   </s:ElementType>
</s:Schema>
<rs:data>
   <z:row c0='CDO and You' c1='file://./backofficestorage/
      thomriznt5dom.extest.microsoft.com/Public Folders/140/Training/
      Schedule/{75BD5A83-09E7-47B7-A9F1-A75DD62F5BA7}.EML'
      c2='2000-03-08T18:00:00' c3='2000-03-08T19:00:00'/>
   <z:row c0='Leveraging XML in Exchange 2000' c1=
      'file://./backofficestorage/thomriznt5dom.extest.microsoft.com/
         Public Folders/140/Training/Schedule/{8C35C44B-68EB-4651-AC3E-
         5C475923A7A1}.EML'
       c2='2000-03-08T21:00:00' c3='2000-03-08T23:00:00'/>
</rs:data>
</xml>
```

Using XSL to Format XML

Now that you've retrieved your XML data from Exchange Server, you're probably wondering how to display this data in your application. This is where XSL comes in. While XML provides a great way to *describe* data, it doesn't provide a way to *display* data. And while HTML provides a great way to *display* data, it doesn't provide a good method for *describing* data. XSL bridges the gap between XML and HTML so that you can support rich descriptions of data, while also supporting rich viewing of that data.

The following code, taken from coursexml.asp in the Training application, shows you how to use XSL to format the XML data returned from Exchange Server. Although I don't have room to cover everything XSL allows you to do, I will point out the major tasks you can perform with XSL. This code should help you get started:

```
<SCRIPT LANGUAGE="javascript">
   var thexml;
   function Resort(){
```

```
      var strProp = document.all.SortByProp.innerText;
      //Call sortfield
      sortfield(strProp);
   }

</SCRIPT>
<LABEL ID="SortByProp" style="display:none"
   onpropertychange="javascript:Resort()"></LABEL>
<DIV id="datediv"></DIV>
<BR>
<div id=msgdiv>
</div>

<xml id=reportXSL>
<xsl:template
   xmlns:xsl="uri:xsl"
   xmlns:d="DAV:"
   xmlns:o="urn:schemas-microsoft-com:office:office"
   xmlns:c="<%=strSchema%>"
   xmlns:h="urn:schemas:httpmail:"
   xmlns:cal="urn:schemas:calendar:"
>

<xsl:script>
function getMyDate(objThis, szDateFormatString, szTimeFormatSTring)
{
   var m_objDate = new Date();
   var m_x=0;
   var gszDateString = "";

   var szDate   = objThis.text;
   var szSubStr = szDate.substring(5,7);

   if(szSubStr.charAt(0) == "0")
   {
      szSubStr = szSubStr.charAt(1);
   }
   m_objDate.setUTCFullYear(szDate.substring(0,4));     //Set Year
   m_objDate.setUTCMonth(Number(szSubStr)-1);           //Set Month
   m_objDate.setUTCDate(szDate.substring(8,10));        //Set Date
   m_objDate.setUTCHours(szDate.substring(11,13));      //Set Hours
   m_objDate.setUTCMinutes(szDate.substring(14,16));    //Set Minutes
   m_objDate.setUTCSeconds(szDate.substring(17,19));      //Set Seconds

      var iNumHours = m_objDate.getHours();
     var szFormattedTime = formatTime(m_objDate.getVarDate(), szTimeFormatSTring);
     var szFormattedDate = formatDate(m_objDate.getVarDate(), szDateFormatString);
```

(continued)

```
        gszDateString =
           szFormattedDate.substring(0,szFormattedDate.length-1) + " " +
           szFormattedTime;
        return (gszDateString);

        }


</xsl:script>

   <table id="XMLTable">
   <TBODY>
   xsl:for-each select="d:multistatus/d:response"
      order-by="<%=Request.QueryString("SortBy")%>">
   <xsl:if test="d:propstat/d:prop/href[.!='']">
   <xsl:if test="d:propstat/d:prop/iscollection[.='0']">
   <xsl:if test="d:propstat/d:prop/subject[.!='']">
   <TR><TD>
      <LI>
         <A HREF='detaildrop.asp' title='Click to view more information
            about this course' onclick="vbscript:ExpandCollapse()">
         <xsl:attribute name="ID"><xsl:value-of
            select="d:propstat/d:prop/href" /></xsl:attribute>
         <xsl:value-of select="d:propstat/d:prop/subject" />
         </A>
      </LI>
   </TD></TR>
      <TABLE style="display: none"><xsl:attribute name=
         "ID">Details<xsl:value-of select="d:propstat/d:prop/href"
         /></xsl:attribute>

         <TR><TD>
            <B>Location:</B></TD><TD>
            <xsl:value-of select="d:propstat/d:prop/location" />
               </TD></TR>

               <TR><TD><B>Instructor:</B></TD><TD>
               <!--turn into mailto-->
               <A><xsl:attribute name="href">mailto:<xsl:value-of select=
                  "d:propstat/d:prop/instructoremail" /></xsl:attribute>
               <xsl:value-of select="d:propstat/d:prop/instructoremail" />
               </A>
               </TD></TR>
               <TR><TD><B>Category:</B></TD><TD>
               <xsl:value-of select="d:propstat/d:prop/category" />
```

```
</TD></TR>

<!--convert to the correct timezone -->
<TR><TD><B>Start Time:</B></TD><TD>

<xsl:for-each select="d:propstat/d:prop/starttime">
   <xsl:eval>getMyDate(this,"MM-dd-yyyy",
   "h:mm tt")</xsl:eval></xsl:for-each>

</TD></TR>
<TR><TD><B>End Time:</B></TD><TD>
<xsl:for-each select="d:propstat/d:prop/endtime">
   <xsl:eval>getMyDate(this,"MM-dd-yyyy","h:mm
   tt")</xsl:eval></xsl:for-each>

</TD></TR>
<TR><TD><B>Prerequisites:</B></TD><TD>
<xsl:value-of select="d:propstat/d:prop/prereqs" />

</TD></TR>
<TR><TD><B>Description:</B></TD><TD>
<xsl:value-of select="d:propstat/d:prop/description" />

</TD></TR>
<xsl:if test="d:propstat/d:prop/materialsfilepath[.!='']">

<TR><TD><B>Course Materials:</B></TD>
      <TD>

      <A href=""><xsl:attribute name="onclick"
         >javascript:window.open('<xsl:value-of select=
         "d:propstat/d:prop/materialsfilepath" />');
         window.event.returnValue=false;</xsl:attribute>
      File link to materials
      </A>
       

      <A href=""><xsl:attribute name="onclick"
         >javascript:window.open('<xsl:value-of select=
         "d:propstat/d:prop/materialshttppath" />');
         window.event.returnValue=false;</xsl:attribute>
      HTTP link to materials
      </A>

      </TD></TR>
</xsl:if>
```

(continued)

```
            <xsl:if test="d:propstat/d:prop/discussionurl[.!='']">
                    <TR><TD>


                    <A style="color: olive" href="" title="Click here to
                       view the discussion for this course.">
                       <xsl:attribute name="onclick"
                       >javascript:window.open('<xsl:value-of select=
                       "d:propstat/d:prop/discussionurl" />');
                       window.event.returnValue=false;</xsl:attribute>
                    View Discussion Group
                    </A>

                    </TD></TR>
            </xsl:if>

               <TR><TD>
                    <xsl:choose>
                    <xsl:when test="d:propstat/d:prop/starttime[. &lt;
                       '<%=TurnIntoIso(Date(),"end")%>']">
                    <!--Course has already taken place work -->
                       <B>This course has already taken place.</B>
                    </xsl:when>
                    <xsl:otherwise>
                       <A style="color: olive" href="" title="Click here
                          to register for this course."><xsl:attribute
                          name="onclick">javascript:window.
                          open('register.asp?FullCourseURL=<xsl:value-of
                          select="d:propstat/d:prop/href" />');
                          window.event.returnValue=false;</xsl:attribute>
                       Register for this course
                       </A>

                    </xsl:otherwise>
                    </xsl:choose>

                    </TD></TR>


               <TR><TD><BR></BR></TD></TR>

        </TABLE>


    </xsl:if>
    </xsl:if>
    </xsl:if>
```

```
  </xsl:for-each>
</TBODY>
   </table>
</xsl:template>
</xml>


<script language="javascript">

var URLSchedule ="<%=strURLToSchedule%>";



function sortfield(sortby)
{
   thenode =
      reportXSL.selectSingleNode("xsl:template/table/TBODY/xsl:for-each");
   thenode.setAttribute("order-by", sortby);
   if (thexml.selectSingleNode("a:multistatus/a:response") == null) {
       msgdiv.innerHTML = "<B>No courses found.</B>";
    }
    else{
      msgdiv.innerHTML = thexml.transformNode(reportXSL.documentElement);
   }

}
```

You'll notice that the code contains an XSL template that combines XSL commands with HTML. The template also contains embedded JavaScript that allows the XSL to transform the XML data into correctly formatted HTML.

XSL Elements

Let's take a look at some of the most frequently used XSL elements.

XSL Value-of

Value-of will be one of the XSL elements you use the most. This element places the value of the node you specify as part of the element. Here is an example of this element, taken from the code in the previous section:

```
<xsl:value-of select="d:propstat/d:prop/category" />
```

XSL If

As you can guess by the name, the If element implements simple, conditional logic. You can pass in the *Language* parameter for this element indicating the scripting language you want to use to evaluate script in your condition testing. The *Test*

parameter is the actual condition you want to test for. The following code checks to see whether the subject of the item returned is not empty. You can test for multiple conditions by using the Choose, When, and Otherwise elements discussed next.

```
<xsl:if test="d:propstat/d:prop/subject[.!='']">
```

XSL Choose, When, and Otherwise

Use these three elements together when you require more complex conditional testing. You can use these three elements to implement an If...ElseIf...Else structure. The following code checks to see whether the start time of the course is less than the current time, which would mean the course has already taken place. If the start time is after the current time, a registration link is created for the course.

```
<xsl:choose>
<xsl:when test="d:propstat/d:prop/starttime[. &lt;
   '<%=TurnIntoIso(Date(),"end")%>']">
   <!--Course has already taken place work -->
   <B>This course has already taken place.</B>
</xsl:when>

//Another xsl:when could go here

<xsl:otherwise>
   <A style="color: olive" href=""
   title="Click here to register for this course.">
   <xsl:attribute name="onclick">
   javascript:window.open('register.asp?FullCourseURL=<xsl:value-of
   select="d:propstat/d:prop/href" />');
   window.event.returnValue=false;</xsl:attribute>
   Register for this course
</A>

</xsl:otherwise>
</xsl:choose>
```

XSL Attribute

Notice in the previous code example the use of the XSL Attribute element. This element allows you to put an attribute on an HTML element inside your XSL template. You should do this if, as part of your HTML element, you want to evaluate another XSL element as the HTML element's value. In the previous example, the onclick attribute is added to the A element in the HTML. The href to the item is added as the value for the onclick attribute by using the Value-of XSL element.

XSL For-Each

The XSL For-Each element is similar to Visual Basic's For Each...Next loop. The For-Each element allows you to apply a template to an element. The best example of how you can use For-Each can be found in the first For-Each element that appears in the previous section's code. This For-Each element uses a Select clause and pattern matching to select only the response nodes in the XML. Furthermore, this example uses the Order-by criteria to support sorting the data by a specific node in the XML. The code example from the previous section follows:

```
<xsl:for-each select="d:multistatus/d:response"
   order-by="<%=Request.QueryString("SortBy")%>">
```

XSL Script and XSL Eval

You'll probably want to use the XSL Script and Eval elements together in your template. The Script element allows you to specify a global script that the rest of your XSL template can call. You can pass the Script element a *Language* parameter that specifies the scripting language, such as JavaScript, for your script code.

The Eval element evaluates a script expression and generates a text string. You'll usually need to call a script you defined by using the Script element in your Eval element and have that script return a text value. You can, however, place in-line script in the Eval element as well. The following code gets the date of the course and correctly formats it using the Script and Eval elements:

```
<xsl:script>
function getMyDate(objThis, szDateFormatString, szTimeFormatSTring)
   {
      var m_objDate = new Date();
      var m_x=0;
      var gszDateString = "";

      var szDate   = objThis.text;
      var szSubStr = szDate.substring(5,7);

      if(szSubStr.charAt(0) == "0")
      {
         szSubStr = szSubStr.charAt(1);
      }
      m_objDate.setUTCFullYear(szDate.substring(0,4));   //Set Year
      m_objDate.setUTCMonth(Number(szSubStr)-1);         //Set Month
      m_objDate.setUTCDate(szDate.substring(8,10));      //Set Date
      m_objDate.setUTCHours(szDate.substring(11,13));    //Set Hours
      m_objDate.setUTCMinutes(szDate.substring(14,16));  //Set Minutes
      m_objDate.setUTCSeconds(szDate.substring(17,19));  //Set Seconds
```

(continued)

```
        var iNumHours = m_objDate.getHours();
        var szFormattedTime = formatTime(m_objDate.getVarDate(),
           szTimeFormatSTring);
        var szFormattedDate = formatDate(m_objDate.getVarDate(),
           szDateFormatString);

        gszDateString =
           szFormattedDate.substring(0,szFormattedDate.length-1)
           + " " + szFormattedTime;
        return (gszDateString);

        }


</xsl:script>
. . .
<!--convert to the correct time zone -->
                <TR><TD><B>Start Time:</B></TD><TD>

                <xsl:for-each select="d:propstat/d:prop/starttime">
                   <xsl:eval>getMyDate(this,"MM-dd-yyyy","h:mm tt")
                   </xsl:eval></xsl:for-each>

                </TD></TR>
                <TR><TD><B>End Time:</B></TD><TD>
                <xsl:for-each select="d:propstat/d:prop/endtime">
                   <xsl:eval>getMyDate(this,"MM-dd-yyyy","h:mm tt")
                   </xsl:eval></xsl:for-each>
```

While I've barely begun to scratch the surface of XSL, this overview should help you get started in transforming your XML using XSL. The best resource I have found on XSL is the MSDN library at *http://msdn.microsoft.com/*. Check out the Web Workshop section in the Platform SDK. Not only does it include lots of documentation on XML, it includes a wealth of information on XSL.

THE XML DOCUMENT OBJECT MODEL

I won't cover the XML Document Object Model here, but it's important for you to know about it. The XMLDOM provides a programmatic way for you to get, change, and create XML nodes in an XML document. Using the XMLDOM, you can display the data returned to you from WebDAV and avoid using XSL. However, the XMLDOM will most likely be slower than XSL because you will have to traverse through all the elements in the XML document and print them out using script. XSL is implemented by Internet Explorer natively, so you pass your XSL template to Internet Explorer and it transforms the XML document using your XSL template.

REUSING OUTLOOK WEB ACCESS

One of the great things about the new version of Outlook Web Access is that it lets you customize the client by either using Web Storage System forms, which we'll look at momentarily, or simply adding some parameters to the URL you pass to OWA. In this section, we'll take a quick look at the parameters you can pass to OWA to make it perform the functionality you want. Table 19-3 shows these parameters.

Command	*Supported Parameters*	*Description*
Cmd=	*Navbar*	Displays only the left-hand navigation of OWA, including the Outlook bar, Outlook icons, and the folder list.
	Contents	Displays only the right-hand contents of OWA without navigation bar.
	New	Creates a new item in the folder.
	Options	Displays the Options page.
	Open	Opens an item for reading. Be careful because e-mail messages end with *.EML*. If you pass just the subject of the e-mail message without the trailing .EML, OWA will not be able to find the item.
	Edit	Opens an item for editing.
	Reply, ReplyAll, Forward	Performs the specific operation on the item.
View=	A string that specifies a view in the folder—for example, "*By Conversation Topic*", "*Daily*", "*Monthly*"	Shows the items in the folder using the view you specify.
M=	A number that corresponds to a month	The month value you want to display.
D=	A number that corresponds to a day	The day value you want to display.
Y=	A number that corresponds to a year	The year value you want to display.

Table 19-3. *The parameters you can use to customize Outlook Web Access.*

The Training application takes advantage of the cmd=Contents, the view=, and the date semantics, such as m=, d=, and y=. For example, this URL from the Training application shows a public folder calendar with a monthly view, without the navigation bar:

http://server/pfpath/?cmd=contents&view=monthly&d=15&m=4&y=2000

You can take these URL parameters and use them as hyperlinks or even as sources for frames within your applications. This extensibility of OWA allows you to quickly add Web services, such as calendaring, without having to write a single line of code.

WEB STORAGE SYSTEM FORMS

While OWA provides an extensibility model in the form of the URL string, one of the key requirements for customizing OWA is the ability to replace the default forms that it displays for items contained in Exchange Server. Imagine that you created an ASP application that stored information in Exchange Server. Now suppose that, when using OWA to access your application's content, your users got the default OWA forms instead of getting your application's forms. This wouldn't make for a great user experience. That's why Exchange Server supports Web Storage System forms and a Web Storage System forms registry.

Creating a Web Storage System Form

Since Web Storage System forms can be ASP forms or simply HTML forms, there is no custom tool you need to learn to create them. Instead, you can create your forms by using common HTML development tools, such as Microsoft FrontPage or Microsoft Visual InterDev.

When creating Web Storage System forms, you have two choices for rendering. First, you can simply use an ASP file and generate all the code and HTML yourself without getting any help from Exchange Server. Second, you can use HTML forms and the Web Storage System Forms Renderer. This option does not require you to write any code.

Using ASP Forms

The Training application uses the first approach of creating a Web Storage System form because of the flexibility that ASP provides. When a user chooses OWA as her interface to the Events Calendar page and opens an item from the calendar, Exchange Server checks the Web Storage System forms registry and sees that an ASP page is registered for the calendar. Instead of displaying the default calendar form, Exchange Server hands over execution of the application to the ASP application and passes it

some parameters within the URL. As part of the URL, Exchange Server passes the *dataurl* parameter. The *dataurl* parameter is the full path to the item that the user requested from Exchange Server. By using the value in this *dataurl* parameter, you can open the item via ADO and perform the necessary operations for your application. The following code is taken from eventwsform.asp, which is the ASP file called in the Training application:

```
'Figure out display from a querystring variable
set rec = Server.CreateObject("ADODB.Record")

strFullURL = request.querystring("dataurl")
set oConnection = Server.CreateObject("ADODB.Connection")
oConnection.ConnectionString = strFullURL
oConnection.Provider = "ExOledb.Datasource"
oConnection.Open
rec.Open strFullURL,oConnection

Response.Write "<B>Course Details:</B><BR><BR>"
'Open the connection
set iAppt = Server.CreateObject("CDO.Appointment")

i=1

Dim strhref
'Load the appointment into CDO
'Response.Write strhref
strhref = rec.Fields("DAV:href")
iAppt.DataSource.Open strhref,oConnection,1
. . . .
```

Using HTML Forms

Creating an HTML form that supports data binding is easy with Exchange Server 2000 and Web Storage System forms. By using some special HTML markup, you can turn ordinary HTML elements into data-bound elements. The markup that you'll need to implement consists of two attributes to set. First, you must set the name attribute of the HTML element as the schema name that you want to bind to the element. An example of this is urn:schemas:httpmail:subject. The second attribute that you need to set on the HTML element is named class. You always set this attribute to a value of *field* to tell the Web Storage System Form Renderer that this is a data-bound field. The following HTML sets a text box to suddenly be data bound to a Web Storage System field named myprop:

```
My Prop: <input type="text"
   name="urn:schemas:myschema:myprop" class="field">
```

You also receive the *dataurl* parameter as you would in an ASP Web Storage System form. Using *dataurl*, you can set the action of your form to post the data back to the URL from which the data is retrieved. The following code permits a user submitting a form to have that form post back to Exchange Server and then redirect the Web page to the application's default page:

```
<form class="form" method="post"
   actionspec="%dataurl%?Cmd=saveitem&redir=%dataurl%/../">
```

That's it. As you can see, creating a Web Storage System data-bound HTML form is pretty simple.

Registering a Web Storage System Form

Now that you've created a Web Storage System form, you need to tell Exchange Server what to do with it. For example, you might want your form to replace another form. Or, you might want your form to process commands sent to a specific server. In order to tell Exchange Server how to use your form, you must add a form registration to your application. A form registration is just another item in the Exchange Server database; however, it contains some special properties, which we'll examine momentarily. Normally, you will want to save your form registrations in the schema folder for your applications in the same place you store your custom content classes and properties.

The following code shows you how to register a form using ADO. This particular registration will display the custom form whenever a user browses to the folder over http, either by typing a URL in Internet Explorer or by browsing through the folders of OWA and selecting a folder.

```
Set oRec = CreateObject("ADODB.Record")
oRec.Open "default.reg", oCon, 3, 0

oRec.Fields("DAV:contentclass") = _
   "urn:schemas-microsoft-com:office:forms#registration"
oRec.Fields("urn:schemas-microsoft-com:office:forms#contentclass") = _
   "urn:content-classes:folder"
oRec.Fields("urn:schemas-microsoft-com:office:forms#request")="GET"
oRec.Fields("urn:schemas-microsoft-com:office:forms#cmd") = "*"
oRec.Fields("urn:schemas-microsoft-com:office:forms#formurl") = _
   "default.htm"
oRec.Fields("urn:schemas-microsoft-com:office:forms#executeurl") = _
   "/exchweb/bin/exwform.dll"
oRec.Fields_
   ("urn:schemas-microsoft-com:office:forms#executeparameters") = ""
oRec.Fields("urn:schemas-microsoft-com:office:forms#contentstate") = "*"
oRec.Fields("urn:schemas-microsoft-com:office:forms#platform") = "WINNT"
oRec.Fields("urn:schemas-microsoft-com:office:forms#browser") = "*"
oRec.Fields("urn:schemas-microsoft-com:office:forms#majorver") = "*"
```

```
oRec.Fields("urn:schemas-microsoft-com:office:forms#minorver") = "*"
oRec.Fields("urn:schemas-microsoft-com:office:forms#version") = ""
oRec.Fields("urn:schemas-microsoft-com:office:forms#messagestate") = "*"
oRec.Fields("urn:schemas-microsoft-com:office:forms#language") = "*"
oRec.Fields.Update

oRec.Close
```

The code simply creates a new item in Exchange Server. It sets the content class of the item as *urn:schemas-microsoft-com:office:forms#registration*, which tells Exchange Server that the item is a form registration. Then the code sets some properties that tell Exchange Server which commands and requests the form should handle, what languages the form should be used for, and which type the form is. You can have multiple form registrations in a single folder that target similar types of requests but display different forms based on the criteria you set, such as browser, language, or content state. Table 19-4 describes each of these properties in more detail. Be aware that, when appropriate, properties can support wildcards that allow all values.

Property Name	***Description***
urn:schemas-microsoft-com:office:forms#browser	Indicates the type of browser. This property is useful if you want different forms for different browsers. For example, you can detect a cell phone microbrowser making a request and return an appropriate version of the form for that type of browser.
urn:schemas-microsoft-com:office:forms#cmd	This corresponds to a URL query string Cmd parameter, such as *Myurl/?cmd=contents*. You can also set this property to custom commands that you implement and register. For example, you could have a parameter named *NewEvent*, which you register for when the url contains *Myurl/?cmd=NewEvent*.
urn:schemas-microsoft-com:office:forms#contentclass	The content class the form is registered for. You can specify a built-in content class or a custom one.
urn:schemas:microsoft-com:office:forms#contentstate	This checks the form against the *urn:schemas:microsoft-com:office:forms#contentstate* property. You can set this property to any value you want. For example, you could set *contentstate* to *approved*, and have a form display the item differently depending on whether the *contentstate* property is *approved* or *not approved*.

Table 19-4. *Form registration properties.* *(continued)*

Table 19-4 *continued*

Property Name	*Description*
urn:schemas-microsoft-com:office:forms#executeparameters	Parameters to pass to the form rendering engine.
urn:schemas-microsoft-com:office:forms#executeurl	The URL from which to execute the form. If you're using the built-in renderer, this URL will be */exchweb/bin/exuform.dll*. However, if you're using ASP, you'll want to use the URL of your ASP page as this value.
urn:schemas-microsoft-com:office:forms#formurl	The URL to the form that should be used to render the data.
urn:schemas-microsoft-com:office:forms#language	Allows you to specify the language of the browser client for which the form should be rendered. This property can be any language code that is valid for http headers.
urn:schemas-microsoft-com:office:forms#majorver	The browser major version.
urn:schemas-microsoft-com:office:forms#messagestate	The message state. An example of this value can be *read* or *submitted*.
urn:schemas-microsoft-com:office:forms#minorver	The browser minor version.
urn:schemas-microsoft-com:office:forms#platform	Indicates the platform of the browser, such as WINNT or UNIX.
urn:schemas-microsoft-com:office:forms#request	Specifies whether the form is the default for a GET or POST request.

Since I showed you an example of registering a form by using the built-in forms renderer, I thought I should show you an example of a registration that uses an ASP page. This code is taken from the setup program for the Training application. Notice how the *formurl* and *executeurl* parameters differ from when we registered to use the built-in forms renderer.

```
'Create the registration in the schema folder
   Set oRec = CreateObject("ADODB.Record")
   oRec.Open strPath & "schema/webstoreform.reg", oConnection, 3, 0

   oRec.Fields("DAV:contentclass") = _
      "urn:schemas-microsoft-com:office:forms#registration"
   oRec.Fields("urn:schemas-microsoft-com:office:forms#contentclass") = _
      "urn:content-classes:appointment"
   oRec.Fields("urn:schemas-microsoft-com:office:forms#request") = "GET"
   oRec.Fields("urn:schemas-microsoft-com:office:forms#cmd") = "*"
```

```
'Put full URL to Web Store forms
oRec.Fields("urn:schemas-microsoft-com:office:forms#formurl") = _
   strHTTPURL & "/eventwsform.asp"
oRec.Fields("urn:schemas-microsoft-com:office:forms#executeurl") = _
   strHTTPURL & "/eventwsform.asp"
oRec.Fields _
   ("urn:schemas-microsoft-com:office:forms#executeparameters") = ""
oRec.Fields("urn:schemas-microsoft-com:office:forms#contentstate") = "*"
oRec.Fields("urn:schemas-microsoft-com:office:forms#platform") = "WINNT"
oRec.Fields("urn:schemas-microsoft-com:office:forms#browser") = "*"
oRec.Fields("urn:schemas-microsoft-com:office:forms#majorver") = "*"
oRec.Fields("urn:schemas-microsoft-com:office:forms#minorver") = "*"
oRec.Fields("urn:schemas-microsoft-com:office:forms#version") = ""
oRec.Fields("urn:schemas-microsoft-com:office:forms#messagestate") = _
   "*"
oRec.Fields("urn:schemas-microsoft-com:office:forms#language") = "*"
oRec.Fields.Update
oRec.Close
```

Another way you can register a new form for a folder is by using the *DAV:defaultdocument* property. This property takes a string that specifies the Web page to open when the user browses to the folder by using OWA. Instead of displaying the contents of the folder when this property is set, OWA displays the custom form. You can easily set this property by using ADO or WebDAV.

FrontPage 2000 Support

Although you can create and register your forms manually, you can also take advantage of the support Exchange Server 2000 provides for FrontPage 2000. You can create and run FrontPage Web pages inside Exchange Server 2000 folders. Since Exchange 2000 has some FrontPage tools that don't require you to do any development (they're pretty much point and click), I won't cover them at length. However, I will show you what the tools look like so that you can understand their capabilities.

First, let's examine the Schema Picker, shown in Figure 19-1. You can drop any HTML element on your Web page into FrontPage and use the schema picker to data bind the element to Exchange Server. If you only name your element and don't use the schema picker, the FrontPage add-ins will generate the schema on the fly when you publish your form.

You can also use the tools in FrontPage to set the properties for your form. In the Choose Content Class dialog box, shown in Figure 19-2, you can change the content class that your application's custom schema uses. If you don't use any custom schema but instead use one from built-in properties, you can simply select an existing content class as your default.

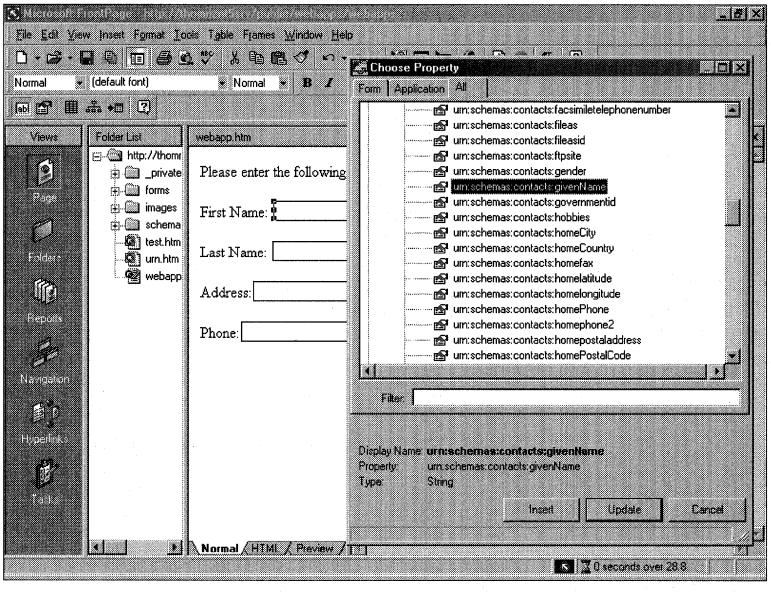


Figure 19-1. *Selecting the property to data bind to an element from the Schema Picker.*

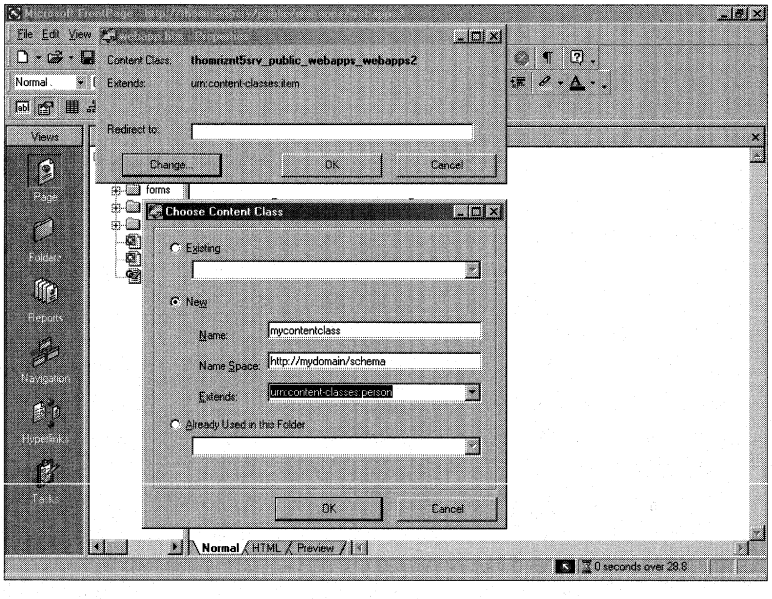


Figure 19-2. *The Form Properties dialog box in FrontPage.*

Another tool offered in FrontPage is the View Design-Time Control (DTC), shown in Figure 19-3. This DTC makes it easy for you to reuse the OWA view capa-

bilities in your application without having to write code. The View DTC also gives you all the features of the OWA View control, such as sorting and grouping.

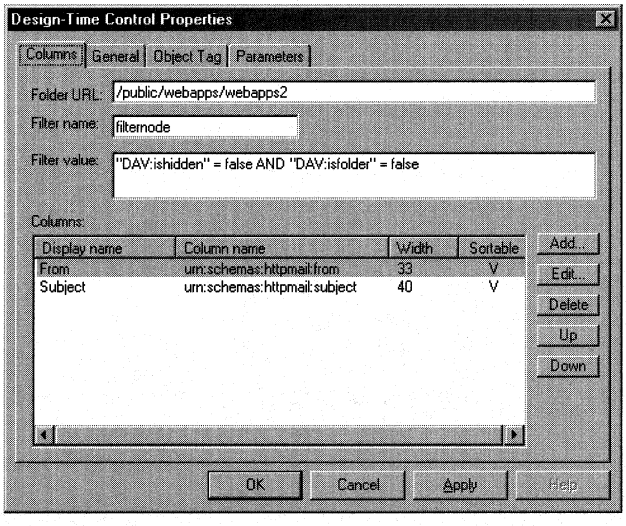


Figure 19-3. *The View DTC in FrontPage.*

After you specify your options for the DTC, it generates some HTML and an XML data island on your Web page. An XML data island describes the view that you want to use with the OWA View control. The values in the XML data island determine which columns and formats will be displayed on your page. You also can generate your own HTML and XML to perform the functionality of the DTC in your applications. The HTML and XML generated by the DTC is shown here:

```
<SPAN id="View1" class="view"
   style="width:544;height:296"
   URL="/public/webapps/webapps2"
   storeType="1"
   RowsPerPage="5"
   viewDescriptor="View1_XML"
   viewClass="/exchweb/views/standardview.xsl"
   linkspec="%DataUrl%"
>
   <XML id="View1_XML">
      <view
         xmlns:v="http://schemas.microsoft.com/schemas/view"
         xmlns:d="urn:uuid:c2f41010-65b3-11d1-a29f-00aa00c14882">
         <baseroot>/exchweb/img/</baseroot>
         <name>Normal</name>
         <column><heading>Subject</heading>
            <prop d:dt="string">urn:schemas:httpmail:subject</prop>
```

(continued)

```
            <width>40</width>
            <sortable>1</sortable>
            <bitmap>0</bitmap><multivalued>0</multivalued>
            <style></style>
         </column>
         <filter id="filternode">
            "DAV:ishidden" = false AND "DAV:isfolder" = false
         </filter>
         <groupby>
            <order>
               <heading>From</heading>
               <prop d:dt="string">urn:schemas:httpmail:from</prop>
               <sort>ASC</sort>
               <style></style>
            </order>
         </groupby>
         <headerstyle>background-color:#c0c0c0</headerstyle>
         <rowstyle>background-color:#c0c0c0</rowstyle>
      </view>
   </XML>
</SPAN>
```

FrontPage also provides the Folder Tree DTC, shown in Figure 19-4. Think of this DTC as the Web equivalent of the Outlook folder picker. Using the Folder Tree DTC, you can select the root folder that the folder picker will start at in your hierarchy.

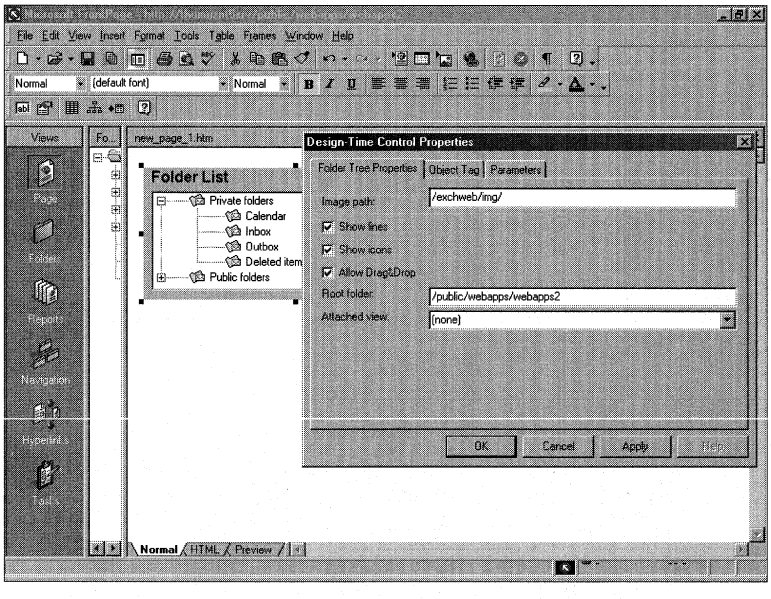


Figure 19-4. *The Folder Tree DTC in FrontPage.*

FrontPage provides one more DTC, the Outlook Web Access DTC, which is shown in Figure 19-5. This DTC lets you set some properties, and then it provides a link to Outlook Web Access using the correct URL parameters. Again, you can achieve the same result yourself, but the DTC makes it easier.

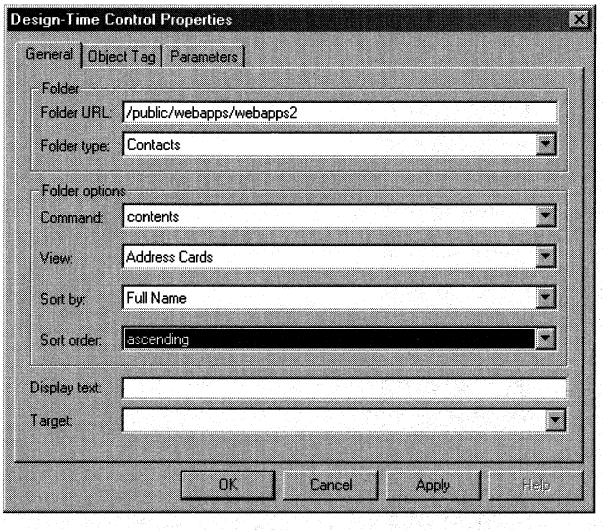


Figure 19-5. *The Outlook Web Access DTC in FrontPage.*

CONTENT INDEXING

Exchange Server 2000 provides built-in support for content indexing. If you plan to use Exchange Server as a repository for a large amount of information, or if you think that your users will require extensive search features, you should consider using content indexing. You can also use content indexing to search for text in attachments on items. Content indexing supports indexing Office and HTML documents as well as standard text attachments. My only caveat for content indexing is that you make sure the requirements for processor and disk resources don't affect the performance of your server. You can schedule incremental crawls of the data sources for content indexing through the Exchange System Manager, shown in Figure 19-6.

Content indexing allows you to perform quick queries against a full-text index inside your Exchange Server applications. Using the full-text index is as easy as generating ADO code that uses the CONTAINS or FREETEXT predicate. You cannot use these predicates until you turn on content indexing for your Exchange server. Be aware that content indexing operates on a per-store basis, meaning there is no top-level index that allows you to search across stores. Let's see how a query containing the CONTAINS or FREETEXT predicate looks.

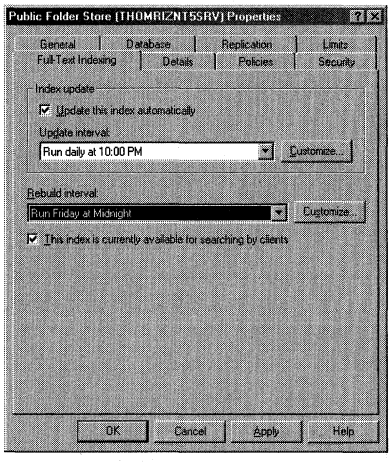


Figure 19-6. *Using the Exchange System Manager for content indexing.*

CONTAINS Predicate

The CONTAINS predicate allows you to perform text-matching operations against the full-text index. With CONTAINS, you can perform simple queries, such as "Show me all items that contain the word *Bob* in the subject," as well as complex queries with weighting on the terms they contain. (You can use a weighted query to indicate relative importance of the terms you search for.) The following code snippet shows several ways you can use the CONTAINS predicate. The code includes a simple version of CONTAINS; it also shows you how to use the NEAR keyword, prefix matching, linguistic matching (such as drive, driving, and so on), and weighted queries.

```
'Contains Bob
   strSQL = "Select ""urn:schemas:httpmail:subject"" From " & _
      "scope('shallow traversal of """ & strURL & _
      """') WHERE CONTAINS(""urn:schemas:httpmail:subject"",' ""Bob"" ')"

   'Contains Bob AND Cool, could also be OR
   strSQL = "Select ""urn:schemas:httpmail:subject"" From " & _
      "scope('shallow traversal of """ & strURL & _
      """') WHERE CONTAINS(""urn:schemas:httpmail:subject"",' ""Bob"" " & _
      "AND ""Cool""')"

   'Word prefix match
   strSQL = "Select ""urn:schemas:httpmail:subject"" From " & _
      "scope('shallow traversal of """ & strURL & _
      """') WHERE CONTAINS(""urn:schemas:httpmail:subject"",' ""*Bob*"" ')"

   'Linguistic matching
   strSQL = "Select ""urn:schemas:httpmail:subject"" From " & _
      scope('shallow traversal of """ & strURL & _
      """') WHERE CONTAINS('FORMSOF(INFLECTIONAL,""drive"") ')"
```

```
'Bob NEAR cool, where ~ is same as NEAR (within 50 words)
strSQL = "Select ""urn:schemas:httpmail:subject"" From " & _
   "scope('shallow traversal of """ & strURL & _
   """') WHERE CONTAINS(""urn:schemas:httpmail:subject""," & _
   "'""Bob"" ~ ""cool""')"

'Weighted Match
strSQL = "Select ""urn:schemas:httpmail:subject"" From " & _
   "scope('shallow traversal of """ & strURL & _
   """') WHERE CONTAINS(""urn:schemas:httpmail:subject""," & _
   "'ISABOUT (""Bob"" WEIGHT(0.9), ""Cool"" WEIGHT(0.1))')"
```

FREETEXT Predicate

You can use the FREETEXT predicate to search columns based on the meaning of the search words rather than the exact wording. When you use FREETEXT, the query engine breaks the string you specify into a number of search terms, assigns weights to the terms, and then attempts to find a match. The following code performs a search for the meaning "best server on the planet." Since I have in my Inbox a message with the subject "Exchange Server is the best server in the world," the search finds that message even though the wording is slightly different.

NOTE You can use the AND, OR, and FREETEXT predicates together.

```
'Freetext
strSQL = "Select ""urn:schemas:httpmail:subject"" From " & _
   "scope('shallow traversal of """ & strURL & _
   """') WHERE FREETEXT(""urn:schemas:httpmail:subject""," & _
   "'best server on the planet')"
```

Working with Ranking

When using content indexing with a query, you might want to retrieve the rank value of a document as compared to the search terms in your query. The search engine will assign a rank of 0–1000 to your items depending on how well they match the query. In your ADO SELECT statement, you can request the rank property by adding the property *urn:schemas.microsoft.com:fulltextqueryinfo:rank*. You can use the ORDER BY predicate with this property to sort the items returned from the query based on their relevance to the search terms. Also, you can force Exchange to coerce the values for the rank either by using clause weighting or by using rank coercion.

The idea of clause weighting is similar to the idea used in the weighted column example we examined earlier. The difference is that instead of applying the weight to only the column, you apply the weight to the entire search term using the RANK BY predicate. This predicate takes a number of options such as WEIGHT and COERCION.

The next example uses clause ranking, so the WEIGHT option is used. This option takes a decimal value from *0* through *1* plus up to three digits past the decimal, such as 0.832. Using the technique of clause weighting, you can assign certain search terms a fraction of the weight that other search terms have. The following example searches all properties on the items for the term *transportation* and the terms *heavy* and *trains*. As you can see by the ranking, if the search engine finds only *transportation*, it should rank that item at one-quarter the value of an item containing *heavy trains*. When using weighting in this manner, the search engine applies weighting to the terms in the preprocessing stage.

```
StrSQL = "Select ""urn:schemas:httpmail:subject"", " _ &
   """urn:schemas.microsoft.com:fulltextqueryinfo:rank"" FROM " _ &
   "scope('shallow traversal of "file://./backofficestorage/ _ &
   "thomriznt5dom.extest.microsoft.com/apps/items/""') WHERE " _ &
   "CONTAINS(*,'""transportation""') RANK BY WEIGHT(0.25) OR " _ &
   "CONTAINS(*,'""heavy trains""') RANK BY WEIGHT(1.0)"
```

Coercion, especially rank coercion, is a post-processing concept in which after the search engine finds matches for the search terms, your application can tell the search to recalculate the rank according to your specifications. Coercion is best illustrated with an example. Suppose you are searching for a document that contains the word *Exchange*. If the word *Exchange* is in a particular property that you think will make the item containing the word very relevant, such as the *subject* property, you can coerce the search engine into ranking the item you are searching for very high. If the search engine finds an item with the word *Exchange* in another property (that is, a property other than the *subject* property), you can have the search engine readjust the ranking so that the items containing *Exchange* found in the other properties are ranked lower than those found in the *subject* property.

You can perform the coercion using one of two approaches. One approach is absolute coercion, in which you assign an absolute value such as *500* to the items that meet your criteria for coercion. But what if you have more complex scenarios and absolute coercion will not meet your needs? For example, you want items with the word *Exchange* in the *subject* property to be ranked from 900 through 1000. (Remember that rank can range from 0 through 1000.) Using a coercion formula—the second approach—you can tell the search engine to make the coerced rank of these items according to this formula: 900 plus the uncoerced rank multiplied by 0.1. For the items containing the word *Exchange* in a property different from *subject*, you can coerce the rank to be in the range of 0 through 900 by making the coerced rank equal to the uncoerced rank multipled by 0.9.

Using a coercion formula requires that your users know which columns they should have the search engine rank higher when their search criterion involves those columns. You could implement some logic in your application to take a shot at defining which columns should be coerced as ranking higher if search terms are found in those columns. The following code shows both absolute coercion and using a coercion formula for the example we just looked at:

```
'Use absolute coercion
'1000 - Exchange in Subject
'500 - Exchange anywhere else
StrSQL = "Select ""urn:schemas:httpmail:subject""," _ &
   """urn:schemas.microsoft.com:fulltextqueryinfo:rank"" FROM " _ &
   "scope('shallow traversal of "file://./backofficestorage/" _ &
   "thomriznt5dom.extest.microsoft.com/apps/items/""') WHERE " _ &
   "CONTAINS(""urn:schemas:httpmail:subject"",'""Exchange""') " _ &
   "RANK BY COERCION(ABSOLUTE,1000) OR CONTAINS(*,'""Exchange""') " _ &
   "RANK BY COERCION(ABSOLUTE,500)"

'Use coercion formula
'900 - 1000 - Exchange in Subject using MULTIPLY and ADD
'0 - 900 - Exchange anywhere else using MULTIPLY
'MULTIPLY takes a decimal number from 0 to 1, with 3 digits after
'the decimal
'ADD takes an integer
'You cannot go above 1000
StrSQL = "Select ""urn:schemas:httpmail:subject""," _ &
   """urn:schemas.microsoft.com:fulltextqueryinfo:rank"" FROM " _ &
   "scope('shallow traversal of "file://./backofficestorage/ _ &
   "thomriznt5dom.extest.microsoft.com/apps/items/""') WHERE " _ &
   "(CONTAINS(""urn:schemas:httpmail:subject"",'""Exchange""') " _ &
   "RANK BY COERCION(MULTIPLY,0.1)) RANK BY COERCION(ADD,900) OR " _ &
   "CONTAINS(*,'""Exchange""') RANK BY COERCION(MULTIPLY,0.9)"
```

Indexing Default Properties

The content indexing engine by default indexes a certain set of built-in properties, which are listed in Table 19-5. Note that at this time, there is no simple way to tell the engine to index your custom properties, such as setting a *fulltextindexed* property in your schema. The only way to ensure that your custom properties are full-text indexed is to create a text file, such as *http://thomriz.com/schema/myprop*, that contains the fully qualified names for your properties on separate lines. You then need to set the following registry key to point at that text file:

```
HKLM\Software\Microsoft\Search\1.0\ExchangeParameters\SchemaTextFilePathName
```

Note that if any of the keys do not exist, you must create them.

MAPI Property	urn:schemas:httpmail *Property*
PR_SUBJECT, PR_SUBJECT_W	*urn:schemas:httpmail:subject*
PR_BODY, PR_BODY_W	*urn:schemas:httpmail:textdescription*
PR_SENDER_NAME, PR_SENDER_NAME_W	*urn:schemas:httpmail:textdescription*
PR_SENDER_NAME_W	*urn:schemas:httpmail:sendername*
PR_SENT_REPRESENTING_NAME, PR_SENT_REPRESENTING_NAME_W	*urn:schemas:httpmail:fromname*
PR_DISPLAY_TO, PR_DISPLAY_TO_W	*urn:schemas:httpmail:displayto*
PR_DISPLAY_CC, PR_DISPLAY_CC_W	*urn:schemas:httpmail:displaycc*
PR_DISPLAY_BCC, PR_DISPLAY_BCC_W	*urn:schemas:httpmail:displaybcc*
PR_SENDER_EMAIL_ADDRESS, PR_SENDER_EMAIL_ADDRESS_W	*urn:schemas:httpmail:senderemail*

Table 19-5. *The set of built-in properties indexed by the content indexing engine by default.*

The following code, taken from the Training application, can be used when you turn on content indexing for the application. For right now, I've commented out this code.

```
'Enable this if you have content indexing enabled on your system
'Uses CONTAINS instead of LIKE
'***************************** BEGIN
'strCategoryText = "CONTAINS (""" & strSchema & "category"","

'if strCategories = "all" then
'   arrCategories = Session("arrCategories")
   'Select the first one
   'Generate the rest
   'strCategoriesSQL = strCategoryText
'   for i=lbound(arrCategories) to UBound(arrCategories)
'      if i=LBound(arrCategories) then
        'First one, start the '
'        strCategoriesSQL = strCategoryText & "'""" & arrCategories(i)
'        if lbound(arrCategories) = UBound(arrCategories) then
           'Only one, end the statement
'           strCategoriesSQL = strCategoriesSQL & """')"
'        else
'           strCategoriesSQL = strCategoriesSQL & """ OR "
'        end if
```

```
'      elseif (i<UBound(arrCategories) AND i>LBound(arrCategories)) then
'         strCategoriesSQL = strCategoriesSQL & """" & _
'            arrCategories(i) & """ OR "
'      else
         'It's the last one, drop the OR
'         if Right(arrCategories(i),1) = chr(10) then
            'Must be a carriage return/linefeed
'            arrCategories(i) = Mid(arrCategories(i), _
               1,(len(arrCategories(i))-2))
'         end if
'         strCategoriesSQL = strCategoriesSQL & """" & _
            Trim(Cstr(arrCategories(i))) & """')"
'      end if
'   next
'else
   'Need to create the category search string
   'Grab the querystring value which should be separated by $
'   strCats = Request.QueryString("Categories")
'   arrCats = Split(strCats,"$")
   'Always going to be at least one
'   for i=lbound(arrCats) to UBound(arrCats)
'      if i=LBound(arrCats) then
         'First one, start the '
'         strCategoriesSQL = strCategoryText & "'""" & arrCats(i)
'         if lbound(arrCats) = UBound(arrCats) then
            'Only one, end the statement
'            strCategoriesSQL = strCategoriesSQL & """')"
'         else
'            strCategoriesSQL = strCategoriesSQL & """ OR "
'         end if
'      elseif (i<UBound(arrCats) AND i>LBound(arrCats)) then
'         strCategoriesSQL = strCategoriesSQL & """" & arrCats(i) & _
            """ OR "
'      else
         'It's the last one, drop the OR
'         if Right(arrCats(i),1) = chr(10) then
            'Must be a carriage return/linefeed
'            arrCats(i) = Mid(arrCats(i),1,(len(arrCats(i))-2))
'         end if
'         strCategoriesSQL = strCategoriesSQL & """" & _
            Trim(Cstr(arrCats(i))) & """')"
'      end if
'   next
'end if
'********************************** END
```

EXCHANGE SERVER EVENTS

Being able to capture and program events in Exchange Server 2000 opens up a world of possibilities when writing applications. Because the events are server side, it doesn't matter which client puts items into the database; no matter what the client, the events will fire and your code will run. This means that if a user drags and drops a Word document into Exchange Server using the new file system capabilities, your events will fire.

Exchange Server 5.5 took an impressive first step toward utilizing events. As you've seen in this book, the Event Scripting Agent provides countless possibilities for Exchange Server 5.5 applications to add workflow, data validation on the server, and other server-side programming capabilities. However, the Exchange Server 5.5 Event Scripting Agent supports only asynchronous events, meaning events are fired after the item has been committed into the Exchange Server database. This makes it hard to prohibit users from moving, copying, deleting, or modifying items since you're notified after they have already performed these activities. Fortunately, any applications you've written using the Exchange Server 5.5 event architecture will run in Exchange Server 2000.

Besides supporting asynchronous events, Exchange Server 2000 adds synchronous events, which are called before the item is committed to the database. Your application can look at the item, and then either accept it or prevent it from being committed into Exchange Server.

Be aware that events are fired when items are moved, copied, created, or deleted in Exchange Server. The scope of these events is scoped to a folder, and you can write event handlers that are passed ADO or OLE DB interfaces, depending on the programming language you use. You can write your event handlers in script, Microsoft Visual Basic, or Microsoft Visual C++. I'll concentrate on Visual Basic in this discussion. I recommend that you write your event handlers in this language, unless of course your language of choice is Visual C++.

I don't recommend writing the event handlers in script because it's harder to write and debug. However, if you don't have permissions on the server to install your event handlers, using scripts can be helpful. Since your event handler is a script that you can store directly in Exchange and not a COM component you need to install on the Exchange server, as long as you have permissions to register for events, your scripts can fire.

Firing Order of Events

When an item is saved into Exchange Server, events are fired in a fixed order in the system. This firing order is important to understand because multiple entities can be working on the items in your folders, which can make it look as though your application

isn't working. Synchronous events fire first, followed by any folder rules, and finally by asynchronous events. As you might have guessed, these events build on one another. For instance, if your synchronous event aborts the item from being saved, then the rules and asynchronous events will not be notified of the item. Furthermore, if any events that are higher up in the chain move or delete an item, the other events should be prepared to not have access to the item.

Security Requirements

There are some security requirements that you should know about before you write your event handlers. First, you must be a folder owner to register an event handler in a folder. Plus, Exchange Server provides an extra security precaution: if you're registering someone else's event handler, he can implement the *ICreateRegistration* interface, which is called when you attempt to register his component as the object to handle events in your folder. If the component writer wants, he can prohibit you from registering his event handler.

Second, if you're writing COM components to implement your event handlers, you must have access to install components on the Exchange server. Exchange Server does not support instantiating and running remote components via DCOM as event handlers. Although there is nothing stopping you from doing this in your event handler, you must remember that the EXOLEDB provider is not remotable. Therefore, you should run your code on the same server the data is on. You can, however, use DCOM to connect to another component residing on another server that accesses data on that server. Getting the security contexts to support this will be the hard part. If you use COM+ applications (which we'll talk about later in this section), you can have your components run in a specific security context, making this easier.

Supported Events

As mentioned earlier, Exchange 2000 supports a wide range of events, including synchronous events, asynchronous events, and system events. We will cover these in this section.

Synchronous Events

Synchronous event handlers are called twice by Exchange Server. The first time, the event handler is called before the item is committed into the Exchange Server database. The second time the event handler is called, the item either has been committed or aborted. On the first pass, the item is read/write. You can modify properties or copy the item somewhere else. However, on the second pass, after the transaction has been committed, the item is read-only. Be aware that the item is not a true item in the database on the first pass. Since the item is not yet committed, you

shouldn't grab any properties that could change in the future—between the time you access the item and the time it is committed. For example, the URL to the item is not valid on the first pass since the item is not yet in the database. Since many factors could change the URL, you shouldn't query or save it during the first pass.

In synchronous events, your event handler runs in the context of a local OLE DB transaction. Therefore, any changes that you make to the item will not trigger other events. However, you also must realize that the work performed in your event handler can be undone if another event handler rejects the item from being committed. All the event handlers that act on an item in a folder must commit for the action to occur. If any event handler rejects the transaction, the action will not occur. If your event handler has already run, it will be called a second time and will be notified that the action has been aborted. Your event handler can then perform any necessary cleanup.

For example, let's say you have two event handlers registered in a folder for the OnSyncSave event. One of them opens the item and saves attachments to another location. The other validates data in the item before allowing it to commit. Based on the priority you set for your event handlers, if the validation occurs after copying the attachments, the validation could fail and the transaction for saving the item could be aborted. Now, any good developer obviously would make validation precede the other events. However, since multiple developers can register event handlers (as long as they meet the security requirements mentioned earlier), you should be aware that other event handlers can abort transactions. In this case, your event handler for copying the attachments will be notified that the transaction was aborted, and you should clean up your work by deleting the attachments from the other location.

You also should be aware that synchronous event handlers, while running, are the only process that can access an item. Exchange Server will block any other threads, processes, or applications from accessing that item while your event handler is running and working on it. This is critical because if you write an inefficient event handler, you can degrade the performance of other applications and Exchange Server. For example, if your event handler takes 10 seconds to run, each time an Outlook user saves an item that triggers your event handler, Outlook will show an hourglass for 10 seconds. Therefore, if you can use asynchronous events to implement your functionality, you should do so. If the Outlook scenario had been with an asynchronous event, Outlook would have returned immediately and allowed the user to continue working.

Synchronous events are also expensive for Exchange Server to perform. The server needs to stop its processing on the item, call your event handler, wait, and then figure out whether to commit or abort the transaction for the item based on your event handler. All this creates temporary copies of the item before committing and forcing the Exchange Server threads to wait, which affects performance.

Continuing with the Outlook scenario, if you abort the transaction, different clients will display different error messages. Outlook will probably display an error message stating that the item couldn't be saved. You cannot show user interface elements in your event handlers because they are running on the server. Instead, you'll have to find a way to notify your users that they submitted the item incorrectly, or that the action they're trying to perform is not allowed. You can perform this notification via e-mail or another method.

Exchange Server 2000 supports two synchronous events: OnSyncSave and OnSyncDelete. As you can guess from their names, the OnSyncSave and OnSyncDelete events support save and delete operations. However, both events are called as part of move and copy operations. For example, if you move an item from one folder to another, a save event will be fired in the new folder and a delete event will be fired in the old folder. If either is aborted, the move will not occur. With a copy, you will get a save event in the location where the copy is supposed to be placed. You should know that the OnSyncDelete event is not called on an item when the item's parent folder is moved or deleted. Also, OnSyncDelete can distinguish between hard and soft deletes. A hard delete is a deletion in which the item is completely removed from the Exchange database. A soft delete is a deletion in which the item is moved into the dumpster. (An analogy would be the deleted items subfolder for every folder.) A user can recover an item from the dumpster. You are notified of the deletion type by the flags that are passed to the OnSyncDelete event.

OnSyncSave is not called for items in a folder when the parent folder for the items is being moved or copied. The event will be called, however, for the parent folder; you can abort the transaction if you don't want the items moved or copied.

You might be asking yourself, "If only Save and Delete are supported, how do I get notified of a change?" If a user makes a change to an item (such as modifying the subject or any property) and saves the item, you will receive the OnSyncSave event. The flags that are passed to the event will notify you that the item has been modified. However, you will not receive notification of which property the user changed in the item. You will have to scan the item to see what changed. To do this, you must have an original copy of the item, which you can obtain by copying the item in your event handler to another folder.

When you register your event handler, if you do not specify criteria for the types of items for which you want to receive events, Exchange Server will notify you of all new items being put into the folder. Folders store some surprising items that you might not expect to handle in your event handler. For example, when someone publishes a form or adds a field to a folder in Outlook, a hidden item is added to the folder. This will trigger an event. You should set the criteria for your event registration, which we'll learn about in a little while, so that only the items your event handler is interested in can trigger events.

Asynchronous Events

Exchange Server 2000 supports two asynchronous events, OnSave and OnDelete. These asynchronous events are called after an item has been committed to the database, and they fire in no particular order. Although these events are guaranteed to be called, another process or user could delete or move the item before the event handler even sees it. Exchange Server doesn't guarantee when it will call your event handler, but usually your event handler will be called as soon as the item is committed to the Exchange Server database. Furthermore, if multiple asynchronous event handlers are registered for a single folder, Exchange does not guarantee in which order the event handlers will be called. To set the firing order for synchronous event handlers, you can use the *priority* property, which we will discuss in a little while. Again, you should use asynchronous events rather than synchronous events whenever possible.

System Events

The three system events of Exchange Server 2000 are OnMDBStartup, OnMDBShutdown, and OnTimer. OnMDBStartup and OnMDBShutdown are called whenever an Exchange Server store starts or shuts down. This is useful for event handlers that want to scan the database or perform some sort of activity whenever the database starts or shuts down. (An event handler is an object whose methods Microsoft Windows uses to notify an application about events). Because these two events are asynchronous, Exchange Server won't wait for your event handler to finish before continuing execution of the item in question.

The OnTimer event fires according to your configured parameters. For example, you can have a timer event fire every five minutes, daily, weekly or monthly. It all depends on the requirements of your application. We'll see how the Training application uses timer events for notification about new courses and student surveys for courses that already have taken place.

Registering an Event Handler

I'm going to do this a little bit backwards. Since writing event handlers involves working with registration parameters, I'll discuss the registration process first. That way, things will be clearer to you when we talk about writing actual event handlers later in this section.

Registering an event handler is quite easy. Exchange Server 2000 provides a script program called regevent.vbs, which allows you to pass some parameters to the program and regevent.vbs registers for all the event types you specify. Besides using regevent.vbs, you can create event registration items for your applications just by creating new items in Exchange using ADO. In the setup program for the Training application, the three event handlers for the application are registered automatically using the ADO method. We'll look at the code for this registration at the end of this section.

Event Registration Properties

When registering events, you need to set some criteria to tell Exchange Server what events you're interested in, what the ProgID or script location of the event handler is, and so on. Table 19-6 shows the criteria required to register an event handler. All these properties are contained in the *http://schemas.microsoft.com/exchange/events/* name space.

Property Name	*Required*
criteria	No
enabled	No
eventmethod	Yes
matchscope	No
priority	No
scope	Yes (Note that this property is in RC1 but will not be in the RTM product. Therefore, if you are working with RC1 of Exchange 2000, you will have to set this property. In the final version of Exchange 2000, this property will no longer exist.)
scripturl	Yes (for script event handlers only)
sinkclass	Yes
timerexpirytime	No (for timer events only)
timerinterval	Yes (for timer events only)
timerstarttime	Yes (for timer events only)

Table 19-6. *Criteria required by Exchange Server 2000 to register an event handler.*

criteria property

The *criteria* property allows you to specify a SQL WHERE clause that will act as a filter for your event handler so that the handler is called only when items meet your criteria. This property allows you to avoid being called for items that you're not interested in. For example, the Training application uses the following criteria so that it doesn't get called when hidden items or folders are created:

```
WHERE "DAV:ishidden" = false and "DAV:isfolder" = false
```

You can use AND, OR, NOT, or EXISTS as part of your WHERE clause. CONTAINS is not supported, however. Also, if you plan to check custom schema, you must explicitly cast your custom property to the right data type. For example, if you want to make sure that your event handler is called only in an application in which a property on items submitted is greater than 100, you would set the *criteria* property for your event registration to the value at the top of the next page.

```
"WHERE cast($"MySchema/MyNumber"$ as 'i4')>100"
```

Notice how the $ character is used to avoid using double quotation marks.

***enabled* property**

This Boolean property allows you to specify whether your event handler is enabled. Rather than deleting an event registration, if you plan to reuse it in the future, you can set this property to *False*.

***eventmethod* property**

This property is a multivalued string that allows you to specify the types of events you are interested in receiving, such as OnSyncSave and OnDelete. You can register for event methods of the same type, within the same event registration. For example, one event registration can be used for OnSyncSave, OnSyncDelete, OnSave, or OnDelete but cannot include OnTimer. You must register OnTimer and the other system events separately. However, your event handler COM component could implement the interfaces for all the events.

***matchscope* property**

This property allows you to specify the scope of the event. The values for this property can be *any*, *fldonly*, *deep*, *exact*, and *shallow*. You'll use only the value *any* with database-wide events. The scope of the *exact* value is a specific item. This is similar to *shallow*, which fires for items only in the exact folder you specify. The *fldonly* value will notify you only of changes to the folder itself, such as modifications to a property on the folder. The *deep* value notifies you of changes in the current folder as well as any items in subfolders. By setting the property to *deep*, even new subfolders and items created in them will trigger your event handler. This capability is an improvement from Exchange Server 5.5, in which there was no concept of a deep scope. If new folders were created in Exchange 5.5, you would have to explicitly register new event handlers in them. Exchange Server 2000 system events do not support this property.

***priority* property**

This integer property allows you to specify a number that indicates the priority of your event handler, compared to other event handlers. The number for this property can range from *0* to *FFFFFFFF*. By default, your event handler is registered with a value of *65,535* (*0x0000FFFF*). This property is valid only for synchronous events and tells the system what order you want these events to fire in. If you give two synchronous events the same priority, it is undetermined which one will get called first. When registering your event handlers, you might want to check to see whether other event registrations exist in the folder. If they do, check their priority before registering your event handler.

scope property

The *scope* property provides a URL to the folder where the event handler wants to be notified of events. This can be in the form of a *file://* or *http://* URL. This property is valid only for store events, not for system events. Please note that this property, while required in RC1 of Exchange 2000, will be removed from the final product.

scripturl property

When you write script for your event handler, this property holds the URL to the script file. Exchange Server supports the *file://* and *http://* URL formats in this property. The script file can live in Exchange Server or in another location accessible via a URL, as long as the location is on the same machine as the Exchange server. When using script handlers, be aware that you must specify *ExOleDB.ScriptEventSink* for the *sinkclass* property in addition to filling out the *scripturl* property.

sinkclass property

This property holds the CLSID or the ProgID of your event handler. Exchange Server will then instantiate the object when an event is triggered. By default, Exchange Server will cache your object so that it doesn't have to instantiate the object multiple times.

timerexpirytime property

This integer property specifies the number of minutes after the *timerstarttime* property that the event handler should stop receiving OnTimer notifications. If you don't specify this property, your event handler will never stop receiving notifications. This property is valid only for OnTimer event registrations.

timerinterval property

This integer property specifies the amount of minutes to wait to notify your event handler of another OnTimer event. If you do not set this property, Exchange Server will call your event handler only once after the creation of your registration item for the OnTimer event.

timerstarttime property

This property allows you to specify the date and time to start notifying your event handler of OnTimer events. If you do not specify this property, Exchange Server will start notifying your handler immediately. Note that this value can be affected by Exchange storing date and time values as UTC values. Therefore, you're going to want to set the UTC time, not the local time, when you want your timer event to start firing. For example, the Training application setup program needs to create the event registration for survey notifications. The survey notification event handler should be called at 10 p.m. local time every night. To create the correct *timerstarttime* property

value, the setup program has to figure out what 10 p.m. local time is in UTC. Then it registers to be notified at the correct UTC time, which will be converted by Exchange to 10 p.m. local time depending on the server machine's time zone.

Creating an Event Registration Item

The easiest way to register your event handler with Exchange Server is to use ADO to create an event registration item in the Exchange Server database and set the properties we just discussed on that item. Be aware that when you create your event registration item, you should do it in the context of an OLE DB transaction. Why? Because Exchange Server utilizes its own events to provide an event registration event handler, which is installed by default. This event handler looks for items with a special content class, *urn:content-class:storeeventreg*. The event handler then takes those items, scans the properties, and performs its magic, making the items valid event registrations. This magic includes turning the item into an invisible message in the folder by making it part of the associated contents table. This table is the same place that views, forms, and rules are stored.

If you don't use OLE DB transactions, you might trigger the event handler when you first attempt to create your registration item using ADO and set the *DAV:contentclass* property to *urn:content-class:storeeventreg*. If this happens and you haven't set the properties for your registration item yet, the event handler will think that your event registration item is invalid. By using an OLE DB transaction and atomically creating and setting your properties at the same time, you avoid this problem.

The following code, taken from the setup program for the Training application, shows you how easy it is to create an event registration item:

```
...
'Create the Survey Notification Event Registration
   'Timer event
   strNow = Now
   arrRequired = GenerateRequiredEventArray("", "ontimer", _
      "EventSink.SurveyNotify", "", "")
   arrOptional = GenerateOptionalEventArray("", "", "", "", _
      1440, strNow, "")
   CreateEvtRegistration oConnection, strPath & _
      "Schedule/surveynotification", arrRequired, arrOptional, False

...


'Event Registration Helper Sub
Sub CreateEvtRegistration(cn, strEventRegPath, arrRequiredParameters, _
   Optional arrOptionalParameters, Optional bWorkflow)
   On Error GoTo errHandler
```

```
'Create the event registration.
'cn - Connection to Exchange Server database for transaction purposes
'strEventRegPath - Full file path to the event item
'arrRequiredParameters - Required parameters for all event registrations
'arrOptionalParameters - Optional parameters such as criteria

Const propcontentclass = "DAV:contentclass"
Const propScope = "http://schemas.microsoft.com/exchange/events/Scope"

Dim rEvent As New ADODB.Record

cn.BeginTrans
rEvent.Open strEventRegPath, cn, 3, _
   adReadWrite + adCreateOverwrite + adCreateNonCollection

'Set the properties in the item
With rEvent.Fields
   .Item(propcontentclass) = "urn:content-class:storeeventreg"

   'Scroll through and commit required parameters.
   'Scroll through and commit optional parameters.
   If IsArray(arrRequiredParameters) Then
      For i = LBound(arrRequiredParameters, 1) To _
         UBound(arrRequiredParameters, 1)
         'Use Dimension 1 since the second dimension should
         'always be the same
         If Not (IsEmpty(arrRequiredParameters(i, 0))) Then
            .Item(arrRequiredParameters(i, 0)) = _
               arrRequiredParameters(i, 1)
         End If
      Next
   End If

   If IsArray(arrOptionalParameters) Then
      For i = LBound(arrOptionalParameters, 1) To _
         UBound(arrOptionalParameters, 1)
         'Use Dimension 1 since the second dimension should always
         'be the same
         If Not (IsEmpty(arrOptionalParameters(i, 0))) Then
            .Item(arrOptionalParameters(i, 0)) = _
               arrOptionalParameters(i, 1)
         End If
      Next
   End If


   'Add custom properties that the event sink can use to
   'determine the context of this event registration
```

(continued)

```
        If bWorkflow = False Then
            strConfigurationFolderPath = strPath & "Configuration/"
            Dim propname As String
            'Hard-code this one property so that we can always find it!
            propname = "http://thomriz.com/schema/configurationfolderpath"
            'For a case switch in the event sink
            .Append propname, adVariant, , , strConfigurationFolderPath
        End If

        .Update  'Get the ADO object current

    End With
    cn.CommitTrans


    Exit Sub

errHandler:
    MsgBox "Error in CreateEvtRegistration.  Error " & Err.Number & " " & _
        Err.Description
    End
End Sub

Function GenerateRequiredEventArray(strCriteria, strEventMethod, _
    strSinkClass, strScriptURL, strSinkList)

    Const propCriteria = _
        "http://schemas.microsoft.com/exchange/events/Criteria"
    Const propEventMethod = _
        "http://schemas.microsoft.com/exchange/events/EventMethod"
    Const propSinkClass = _
        "http://schemas.microsoft.com/exchange/events/SinkClass"

    Const propScriptURL = _
        "http://schemas.microsoft.com/exchange/events/ScriptUrl"
    Const propSinkList = _
        "http://schemas.microsoft.com/exchange/events/SinkList"

    'Generate the array by checking the passed arguments.
    'Note dynamic arrays only support redimensioning the last dimension.
    'This causes a problem, so dimension an array and fill in blanks if
    'necessary.
    'Flip-flop value - propname
    Dim arrRequired(4, 1)
    iArrayCount = 0
    If strCriteria <> "" Then
        arrRequired(iArrayCount, 0) = propCriteria
        arrRequired(iArrayCount, 1) = strCriteria
```

```
      iArrayCount = iArrayCount + 1
   End If
   If strEventMethod <> "" Then
      arrRequired(iArrayCount, 0) = propEventMethod
      arrRequired(iArrayCount, 1) = strEventMethod
      iArrayCount = iArrayCount + 1
   End If
   If strSinkClass <> "" Then
      arrRequired(iArrayCount, 0) = propSinkClass
      arrRequired(iArrayCount, 1) = strSinkClass
      iArrayCount = iArrayCount + 1
   End If
   If strScriptURL <> "" Then
      arrRequired(iArrayCount, 0) = propScriptURL
      arrRequired(iArrayCount, 1) = strScriptURL
      iArrayCount = iArrayCount + 1
   End If
   If strSinkList <> "" Then
      arrRequired(iArrayCount, 0) = propSinkList
      arrRequired(iArrayCount, 1) = strSinkList
      iArrayCount = iArrayCount + 1
   End If
   GenerateRequiredEventArray = arrRequired

End Function

Function GenerateOptionalEventArray(bEnabled, strMatchScope, lPriority, _
   bReplicateReg, iTimerInterval, iTimerStart, iTimerStop)
   Const propEnabled = _
      "http://schemas.microsoft.com/exchange/events/Enabled"
   Const propMatchScope = _
      "http://schemas.microsoft.com/exchange/events/MatchScope"
   Const propPriority = _
      "http://schemas.microsoft.com/exchange/events/Priority"
   Const propReplicateEventReg = _
      "http://schemas.microsoft.com/exchange/events/ReplicateEventReg"
   Const propTimerInterval = _
      "http://schemas.microsoft.com/exchange/events/TimerInterval"
   Const propTimerStartTime = _
      "http://schemas.microsoft.com/exchange/events/TimerStartTime"
   Const propTimerExpiryTime = _
      "http://schemas.microsoft.com/exchange/events/TimerExpiryTime"

   Dim arrOptional(6, 1)
   iArrayCount = 0
   If bEnabled <> "" Then
      arrOptional(iArrayCount, 0) = propEnabled
      arrOptional(iArrayCount, 1) = bEnabled
```

(continued)

```
        iArrayCount = iArrayCount + 1
    End If
    If strMatchScope <> "" Then
        arrOptional(iArrayCount, 0) = propMatchScope
        arrOptional(iArrayCount, 1) = strMatchScope
        iArrayCount = iArrayCount + 1
    End If
    If lPriority <> "" Then
        arrOptional(iArrayCount, 0) = propPriority
        arrOptional(iArrayCount, 1) = lPriority
        iArrayCount = iArrayCount + 1
    End If
    If bReplicateReg <> "" Then
        arrOptional(iArrayCount, 0) = propReplicateEventReg
        arrOptional(iArrayCount, 1) = bReplicateReg
        iArrayCount = iArrayCount + 1
    End If
    If iTimerInterval <> "" Then
        arrOptional(iArrayCount, 0) = propTimerInterval
        arrOptional(iArrayCount, 1) = iTimerInterval
        iArrayCount = iArrayCount + 1
    End If
    If iTimerStart <> "" Then
        arrOptional(iArrayCount, 0) = propTimerStartTime
        arrOptional(iArrayCount, 1) = iTimerStart
        iArrayCount = iArrayCount + 1
    End If
    If iTimerStop <> "" Then
        arrOptional(iArrayCount, 0) = propTimerStopTime
        arrOptional(iArrayCount, 1) = iTimerStop
        iArrayCount = iArrayCount + 1
    End If


    GenerateOptionalEventArray = arrOptional

End Function
```

As you can see in the code, ADO is used to create an event registration item. Using the Fields collection, the properties needed to make the item a valid event registration are filled in, and the item is saved into Exchange Server. Although this particular application uses events in public folders, you can register and fire events from private folders as well.

Registering a Database-Wide Event

In addition to scoping your event handlers to just one folder, you can scope them to the entire store or MDB. However, you cannot scope your event handler to the entire server. All types of event registrations are per individual MDB (database) only.

To create an MDB-wide event registration, you need to change the value for the *matchscope* property and the location in which you put your registration item.

You should specify *any* as the value of the *matchscope* property. This indicates that any scope is valid for notifying your event handler. All other properties can stay the same, based on the type of event you're registering for.

As just mentioned, the location in which you put the registration item will change. Instead of throwing the item into the folder where you want the scope of the event notifications to begin, you need to place the registration item in the Globalevents folder. This folder is located in Public Folders in the non-ipm subtree in a folder called StoreEvents{MDBGUID}, where MDBGUID is the unique identifier for the MDB. The easiest way to retrieve the MDBGUID is to use the *StoreGUIDFromURL* method in EXOLEDB. The following code sample shows you how to use this method:

```
set oStoreGUID=CreateObject("Exoledb.StoreGuidFromUrl")
'Get the GUID
strguid = oStoreGUID.StoreGuidFromUrl("file://./backofficestorage/" & _
   somestoreitem)
```

A valid path to save your registration item to would look something like this:

```
file://./backofficestorage/ADMIN/domain/public
folders/non_ipm_sub
   tree/StoreEvents{GUID}/globalevents/

'For a mailbox database
File://./backofficestorage/ADMIN/domain/MBX/SystemMailbox{GUID}/_
   Storeevents/GlobalEvents
```

Only the Administrator account, which is a member of the Domain Administrators group, or users in the Exchange Administrators role can register global events. It is not enough to be a member of the Administrators group or the Exchange Servers group.

Using the *ICreateRegistration* Interface

If you're writing global event handlers that you think other developers will register for, or if you want to protect your application event handlers, you can implement the *ICreateRegistration* interface. This interface will be called whenever a user tries to register for your event handler. *ICreateRegistration* passes you the event registration information, including the registration item for the particular user. You can grab information from this item, or you can also retrieve from the item information about the user who's trying to register for your event handler. You can then decide whether to permit or reject the user's registration.

To implement this interface, you need to add an *Implements ICreateRegistration* line to your Visual Basic code and put your validation code in the *ICreateRegistration_Register* function.

Writing an Event Handler

The code samples containing event handlers that you'll see in this section are taken from the Training application. All these event handlers are written in Visual Basic. When writing Visual Basic event handlers, you first should create an ActiveX DLL. In addition, make sure to add references to the various object libraries your application might need to access. You definitely will need a reference to the EXOLEDB type library; libraries such as Microsoft ActiveX Data Objects 2.5 and Microsoft CDO for Exchange Server 2000 also might come in handy.

Once you've added the references, you need to use the Implements keyword. Depending on the type of event handler that you plan to write, you'll need to implement the *IExStoreSyncEvents*, *IExStoreAsyncEvents*, or *IExStoreSystemEvents* interface. You can implement all three in a single DLL if you want.

Next, you must implement the subroutines for the events you're interested in. The following code, taken from the course notification event handler, shows the OnSave and OnTimer events being implemented. I won't list all the code for the implementation of these two events because I mainly want to show you the parameters that are passed your functions.

```
Implements IExStoreAsyncEvents
Implements IExStoreSystemEvents
Const strHTMLMsgSubject = "New Course Email"

Dim oEventRegistration        'Global that holds the event
                              'registration record
Dim bShowPreviousDay          'Global that holds whether to show
                              'new courses only
                              'From previous day, just in case timer
                              'event runs after midnight
Dim oRecord As ADODB.Record   'Global record to hold item that
                              'triggered event
Dim strHTMLBody               'Global string that holds HTML message body
'Add Discussion group, file, and http link notification as part of
'the message.
'Update HTML message to incorporate file and http link, as well as
'discussion group.


Private Sub IExStoreAsyncEvents_OnDelete(ByVal pEventInfo As _
   Exoledb.IExStoreEventInfo, ByVal bstrURLItem As String, _
   ByVal lFlags As Long)
   'Not implemented
End Sub
```

```
Private Sub IExStoreAsyncEvents_OnSave(ByVal pEventInfo As _
   Exoledb.IExStoreEventInfo, ByVal bstrURLItem As String, _
   ByVal lFlags As Long)
   If (lFlags And EVT_NEW_ITEM) > 0 Then
      'New item put in.
      'Get the ADO Record object for the item.
      'Set oRecord = dispInfo.EventRecord
      Set oRecord = CreateObject("ADODB.Record")
      oRecord.Open bstrURLItem
      'Check to see whether the item happened very recently.
      'If it did, just exit since the training event is
      'probably having its survey information updated by
      'a user. Don't notify people of old training events.
      If DateDiff("d", Now, _
         oRecord.Fields("urn:schemas:calendar:dtstart").Value) > 0 Then
         'It's OK; event happens in future.
         'Load the global settings.
         LoadAppSettings oRecord, pEventInfo
         If bEventLogging Then
            App.LogEvent "Event Notification OnSave event called for " & _
               "training event.  Path: " & bstrURLItem
         End If
         strCategory = GetCourseCategory(oRecord)
         QueryPreferences strCategory
      End If
   End If
End Sub


Private Sub IExStoreSystemEvents_OnMDBShutDown _
   (ByVal bstrMDBGuid As String, ByVal lFlags As Long)
   'Not implemented
End Sub

Private Sub IExStoreSystemEvents_OnMDBStartUp _
   (ByVal bstrMDBGuid As String, ByVal bstrMDBName As String, _
   ByVal lFlags As Long)
   'Not implemented
End Sub

Private Sub IExStoreSystemEvents_OnTimer(ByVal bstrURLItem As String, _
   ByVal lFlags As Long)
   Dim rec As ADODB.Record
   Dim rst As ADODB.Recordset
   Dim conn As ADODB.Connection
```

(continued)

```
Set oBindingRecord = Nothing
'Get the registration
On Error Resume Next
Set oBindingRecord = CreateObject("ADODB.Record")
oBindingRecord.Open bstrURLItem
On Error GoTo 0
If Err.Number = 0 Then
   'Could retrieve the item
   LoadAppSettings oBindingRecord

   If bEventLogging Then
      App.LogEvent "Event Notification: OnTimer event called at " & Now
   End If
   'Get the folder in which the timer is running.
   'This is never used since we know the folder already.
   'However, this shows you how to retrieve the folder if you need to.
   strFolder = oBindingRecord.Fields.Item("DAV:parentname")
   'Figure out all the courses created in the current 24 hours or
   'previous 24 hours
   curDate = Date
   If bShowPreviousDay Then
      'Subtract a day
      curDate = DateAdd("d", -1, curDate)
   End If
   dISODateStart = TurnintoISO(curDate, "Start")
   dISODateEnd = TurnintoISO(curDate, "End")

   strSQL = "Select ""urn:schemas:mailheader:subject"","" & _
      ""DAV:href"",""urn:schemas:calendar:dtstart"","" & _
      ""urn:schemas:calendar:dtend"" FROM scope('shallow " & _
      "traversal of """ & strScheduleFolderPath & _
      """') WHERE (""DAV:iscollection"" = false) AND " _ &
      "(""DAV:ishidden"" = false) " & _
      "AND (""urn:schemas:calendar:dtstart"" >= CAST(""" & _
      dISODateStart & """ as 'dateTime'))" & _
      "AND (""urn:schemas:calendar:dtstart"" <= CAST(""" & _
      dISODateEnd & """ as 'dateTime'))"

   Set conn = CreateObject("ADODB.Connection")
   Set rst = CreateObject("ADODB.Recordset")
   Set oRecord = CreateObject("ADODB.Record")

   With conn
      .Provider = "exoledb.datasource"
```

```
            .Open strScheduleFolderPath
        End With

        'Create a new Recordset object

        With rst
            'Open Recordset based on the SQL string
            .Open strSQL, conn, adOpenKeyset
        End With

        Dim iAppt As CDO.Appointment

        If Not (rst.BOF And rst.EOF) Then
            Set iAppt = CreateObject("CDO.Appointment")
            'On Error Resume Next
            rst.MoveFirst
            Do Until rst.EOF
                'Set oRecord to the current item in the Recordset
                On Error Resume Next
                oRecord.Close
                Err.Clear
                On Error GoTo 0
                oRecord.Open rst.Fields("DAV:href").Value, conn
                strCategory = GetCourseCategory(oRecord)
                QueryPreferences strCategory

                rst.MoveNext
            Loop

            rst.Close

            Set iAppt = Nothing
            Set rst = Nothing
            Set rec = Nothing
            Set conn = Nothing
        Else
            If bEventLogging Then
                App.LogEvent "Event Notification: No students need to be " & _
                    "notified of training event."
            End If
        End If
    End If
End Sub
```

As you can see in the code, your application is passed different parameters for the different events. However, all events have some common parameters. For example, you are always passed a URL to the item that triggered the event. For system events such as OnTimer, this is the event registration item itself. If you need to, you can add custom properties to the event registration item so that when your OnTimer event handler is called, you can retrieve that custom property.

I use this trick in the Training application. When I register the OnTimer event handler for the application's workflow process, I add a custom property that is the full URL to the application's configuration folder. Since you can customize where the application is installed, you need to tell the workflow process where to look for the customization information that you select during setup. Because I added an extra property, I can grab it in my workflow code, get the configuration message contained in the folder the property specifies, and determine the value for the customized fields in the application, such as which e-mail address to send notification messages from.

For nontimer events, the URL you receive is the path to the item that's triggering the event. You can then retrieve the item and look at its properties.

An event handler also receives a parameter called *lFlags*. This parameter corresponds to flags that tell you exactly what's happening to the item. For example, one of the flags, EVT_NEW_ITEM, tells you that the item triggering the event is a new item rather than an item that already existed in the folder and had some properties changed. You should use a bitwise AND with the *lFlags* parameter and one of the identifiers from Table 19-7 to determine the values of flags in the *lflags* parameter. The flag names are included in the EXOLEDB type library.

> **NOTE** Not all flags are supported by all events. For example, the delete flags are supported only by the delete events.

When using server events, in addition to the flags and the URL, you will receive a *pEventInfo* variable of type *Exoledb.IExStoreEventInfo*. You should set this variable to another variable of type *Exoledb.IExStoreDispEventInfo*, which is the *IDispatch* version of the interface. From this interface, you can control the transaction state for the event handler for synchronous events and get more information about the context of the event. Table 19-8 outlines the elements you retrieve and call on the *IExStoreDispEventInfo* interface.

Flag	*Description*
EVT_NEW_ITEM	The item being saved is new rather than an existing, modified item.
EVT_IS_COLLECTION	The item being saved is a collection (folder).
EVT_REPLICATED_ITEM	The item being saved is the result of a replication event, which occurs when the item is replicated using Exchange Server replication.
EVT_IS_DELIVERED	The item being saved is the result of a message delivery.
EVT_SOFTDELETE	A soft delete has occurred, meaning the item has been moved to the dumpster.
EVT_HARDDELETE	A hard delete has occurred, meaning the item has been removed from the store.
EVT_INITNEW	This is the first time that the event handler has been called. This allows you to perform any necessary initialization procedures. Your event handler is passed this flag only once during its lifetime.
EVT_MOVE	The event is the result of a move operation.
EVT_COPY	The event is the result of a copy operation.
EVT_SYNC_BEGIN	The event handler is being called in the begin phase of a synchronous event. This is the point at which you can abort the item being saved into the database. Also, the URL you receive for the item is invalid at this point, since the item is not in the database yet.
EVT_SYNC_COMMITTED	The event handler is being called in the commit phase of a synchronous event, after the transaction has been committed to the database.
EVT_SYNC_ABORTED	A synchronous event has been aborted.
EVT_INVALID_URL	This tells the event handler that the URL passed to it is invalid.
EVT_INVALID_SOURCE_URL	The source URL could not be obtained during a move operation.
EVT_ERROR	Some error occurred in the event.

Table 19-7. *Flag values for the* lFlags *parameter.*

Element	*Description*
AbortChange	Aborts the transaction in which the event is currently executing. Exchange Server will not commit the item to its database. You pass a long value, which specifies the error code you want returned.
EventBinding	Returns as an object the registration item for the event handler. You can use the returned object to retrieve the custom properties that you set on the event registration item.
EventConnection	The ADO Connection object under which the event is executing.
EventRecord	The ADO Record object bound to the item that triggered the event.
SourceURL	The original source URL in an OnSyncSave event. This property is valid only for a move operation.
StoreGUID	The GUID for the MDB where the item that triggered the event is located.
UserGUID	The GUID for the user that triggered the event. You can use this in conjunction with the Win32 APIs to look up the user's name.
UserSID	The Security Identifier (SID) of the user who triggered the event.
Data	This property allows you to save in-memory data between the begin and commit or abort phase of a synchronous event. This property is useful for passing to your event sink the variables between the different calls in a transaction; you don't have to save those variables to Exchange or on disk.

Table 19-8. *Elements of the* IExStoreDispEventInfo *interface.*

CREATING COM+ APPLICATIONS

If you plan to write your event handlers in Visual Basic, you should definitely install your DLLs as COM+ applications. There are a number of reasons for doing this. First, Exchange Server won't let you try to run an event handler for synchronous events written in VB that are not COM+ enabled. Second, if your event handler has problems, making it into a COM+ application isolates it and allows you to shut it down without affecting other parts of the system. Finally, by making your DLLs COM+ applications, you can take advantage of COM+ roles and security.

This last point is key—it allows you to retrieve the SID of the user who's triggering the event. Using the COM+ Services Type Library and the SID, you can employ COM+ roles-based security. For example, you can use the *IsSecurityEnabled* function to check whether COM+ security is enabled. You can also use the *IsUserInRole* function to see whether the user is in a particular role. For more information on COM+ security issues, refer to the Platform SDK on MSDN at *http://msdn.microsoft.com*.

The Training application places its event handlers into a COM+ application. The Training application's setup program deploys the COM+ application with the built-in COM+ deployment tools. When you export your COM+ application from the COM+ user interface, as shown in Figure 19-7, COM+ will create a Windows Installer file. You can then shell out to this file or run it directly if you want to install the COM+ application on your computer.

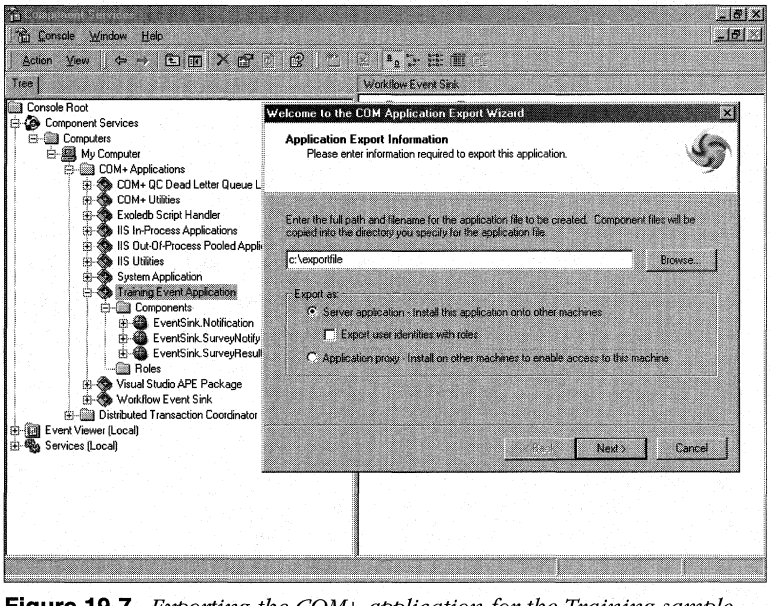


Figure 19-7. *Exporting the COM+ application for the Training sample.*

Debugging an Event Handler

When you set up the Training application, it asks if you want to enable event handler debugging. Debugging your event handler is easy if you use COM+ applications. The only three things you need to do to debug your event handler is set the identity of your COM+ application so that it uses the interactive user, add some breakpoints to your VB event handler, and then put the Visual Basic event handler in run mode. At that point, you'll be able to take advantage of all the great VB debugging. Figure 19-8 shows an example of an event handler in the Visual Basic debugger.

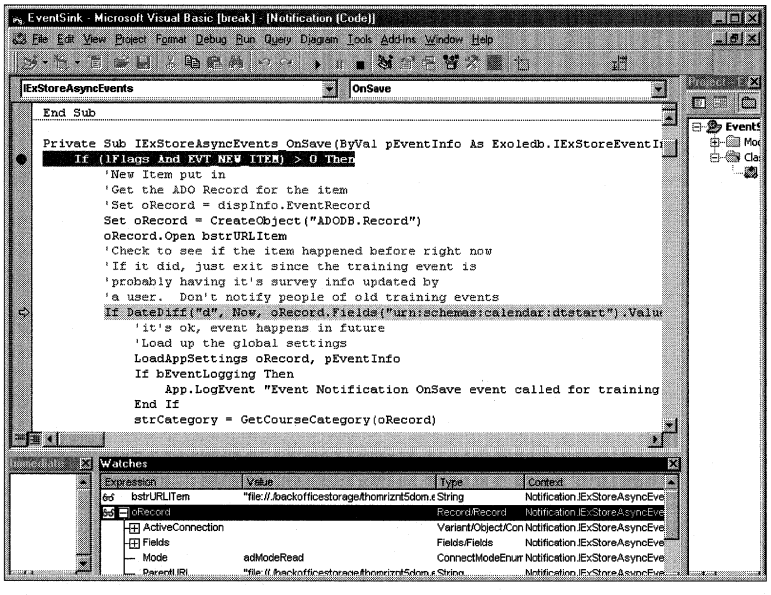


Figure 19-8. *One of the Training application event handlers in the Visual Basic debugger.*

In addition to using the debugger, you can also leverage VB's *LogEvent* method on the App object. Using this method, you can have VB place entries into the Windows 2000 event log. The Training application, when you set it up, asks if you want to enable event handler debugging. This is the sort of debugging that it uses. Figure 19-9 shows an example from the event log of an event handler from one of the Training application's event handlers.

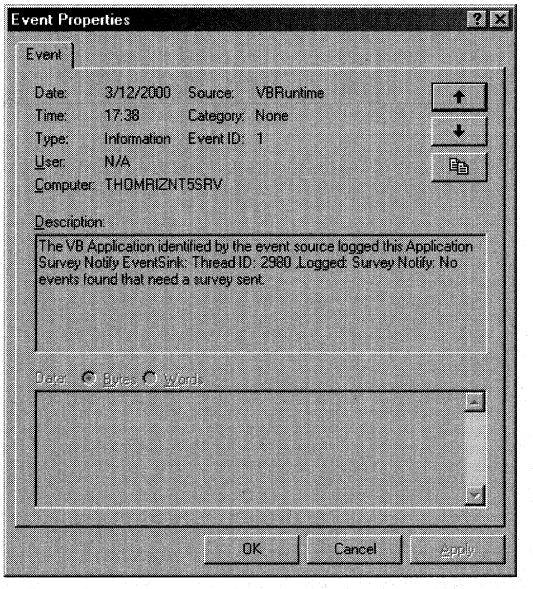


Figure 19-9. *A message posted by Visual Basic into an event handler's log.*

WORKFLOW CAPABILITIES

Exchange Server 2000 allows you to build applications that model and automate business processes. You could hard-code a workflow process by using server events and Visual Basic, but the built-in workflow engine—Workflow Designer—and CDO workflow objects of Exchange Server 2000 simplify this development task by providing specialized workflow tools.

With these tools, you can build both database workflows and e-mail workflows on Exchange Server 2000. You can also enable the user to connect directly to the data, or to an application that directly touches the data to update state information in the workflow. The Training application is a database workflow. When a manager approves a user to take a course, an ASP page updates a property on the pending approval message to the student, which in turn triggers the workflow engine to change the state of the pending approval message to *Approved*. This change of state causes the application to register the student for the course and then notify the student. Database workflows are great if your users have direct access to the process instances of the workflow, and, in effect, to the items undergoing the workflow process, so that they can update the state.

In e-mail workflows, instead of having the user directly interact with the data, notifications and data copies are sent to the user via e-mail. The user can then perform an action, such as approve or edit the data, and return it to the application via e-mail. The Exchange Server workflow engine directs the e-mail to the correct process instance in your workflow folder, and you can update your process instance accordingly. This style of workflow is useful if you can't guarantee that your users have direct access to your data, or if you want to provide an easy transmission method for data and approval. Since most users can access some form of e-mail, this solution works well for business processes that extend beyond a corporation and over the Internet. Please note that to use e-mail–style workflow, you must be running Microsoft Outlook on your client.

How Is Workflow Implemented in Exchange Server?

Exchange Server 2000 uses four main components to implement workflow: CDO for Workflow (CDOWF), server event sinks (handlers), action tables, and script files. The workflow engine in Exchange Server 2000 is stored in CDOWF. This engine works in conjunction with the other components to evaluate and maintain the workflow state, such as *Approved* or *Rejected*. Plus, CDOWF provides the object model you can use to interact with the engine to change states or validate workflow properties.

The workflow engine works in conjunction with the action table, which is basically a finite state machine that describes transitions between states in the workflow. Table 19-9 shows an example of an action table based on the Training application. In this action table, a request for approval of user's enrollment is e-mailed to the user's manager, and the workflow engine waits 15 minutes for a response. If a response isn't received from the manager within 15 minutes, the engine e-mails the user that the request was not processed. If the manager approves the course request, the engine registers the user and sends an e-mail that tells the user of the approval.

State	*New State*	*Event Type*	*Condition*	*Action*	*Expiration Interval*
	Created	OnCreate	CheckValidity	SendMail-toManager	
	Created	OnEnter	True		15
Created	Approved	OnChange	ApprovalState-ofItem	RegisterUser SendMailtoUser	
Created	NoResponse	OnExpiry	True	SendNoRe-sponsetoUser	

Table 19-9. *A simple action table based on the Training application.*

To determine what changes have been made to workflow items and evaluate whether they are valid according to the action table, the workflow engine implements both synchronous and OnTimer event handlers. When you drop a new document into a folder where a workflow process is enabled, the Workflow event handler in that folder fires before the item is committed to Exchange Server database. If workflow is enabled, the event handler creates the process instance for your item in the folder and determines the initial state of the item. If a user changes that item, the event handler checks the change against the valid state changes in the action table. If the state change is not valid, it is not committed to Exchange Server. If the time expires for a specific state and there is a state transition for that event in the action table, the OnTimer event will move the workflow item to the next state. Figure 19-10 shows the Workflow event handler that's installed by default as a COM+ application.

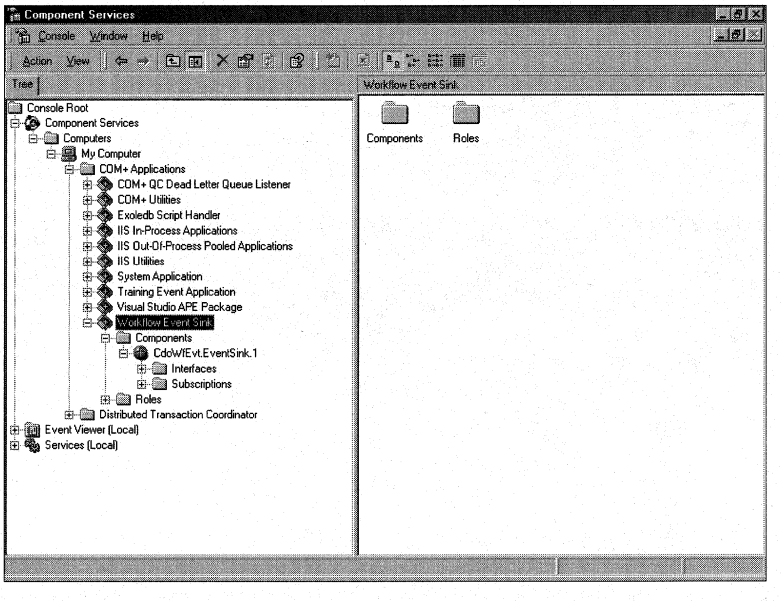


Figure 19-10. *The Workflow event handler installed as a COM+ application.*

The script file is used in conjunction with CDOWF, the server event handlers, and the action tables. Your script implements the conditions for the state transitions and the actions that occur at the time of those transitions. For example, to check the validity of an item before posting it in the workflow folder, you write script to check the necessary properties and return to the workflow engine a Boolean that indicates whether the item is valid. You must use condition functions in your script. If a condition is valid, for example, you might want to send an e-mail to the user who needs to approve the item. You would have to write script for the action portion of your workflow in order to send the e-mail.

Developing Workflow Applications

Enough talking about what workflow is—let's take a look at how to write a workflow application with Exchange Server 2000. In this section, I'll concentrate on the Workflow Designer for Exchange Server 2000 because it makes creating workflow applications easier. Everything we'll discuss in this section can be performed programmatically using CDOWF.

Setting Up the Workflow Environment

To start writing workflow applications, you first need to set up the workflow environment for Exchange Server 2000. I won't go into the gory details of how to set up all the accounts and infrastructure. You can find all this information in the Exchange Server 2000 SDK, in a section called "Adding the Workflow System Account." Essentially, you need to create an account and make it your default workflow system account.

Next, you need to add yourself to the Can Register Workflow role for the Workflow Event Sink COM+ application. This role lists all the users or groups that can actually register for the event handlers we discussed earlier. Figure 19-11 shows this COM+ application with the roles populated.

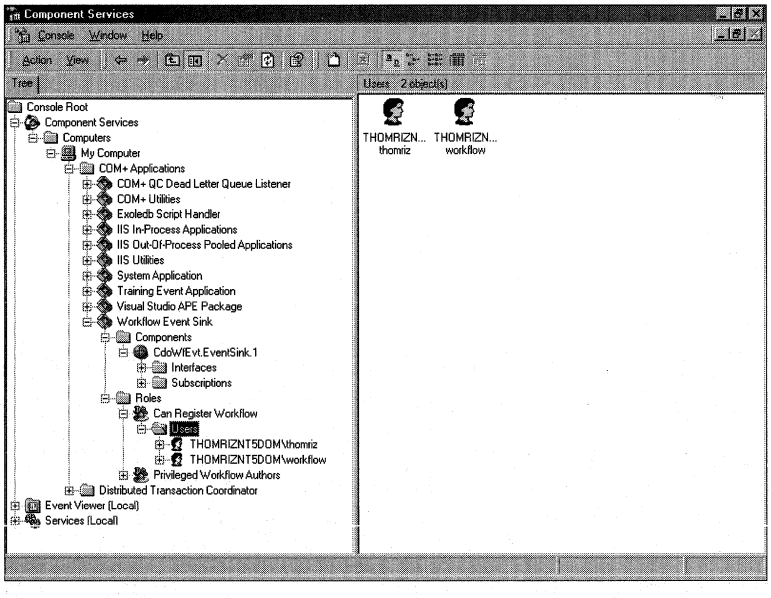


Figure 19-11. *The Can Register Workflow role populated with users and groups.*

If necessary, you also should add yourself to the Privileged Workflow Authors role. If you are not in this role, your workflow scripts will be sandboxed. In other words, they will only be able to modify the item undergoing workflow, send notification mail, or write to the workflow audit trail. They will not be able create objects at all. If you add yourself to the Privileged Workflow Authors role, you'll be able to create objects and access other items you have permissions for in Exchange Server 2000.

> **NOTE** If you are a privileged workflow author, you will be running under the workflow system account you initially specified. Also, Exchange 2000 supports ad-hoc workflows. This means that items coming into the folder where workflow is enabled must already have a workflow definition on them. Only restricted mode workflows can be run as ad-hoc. You cannot allow ad-hoc workflows and enable privileged mode.

Using the Workflow Designer

As mentioned, the Workflow Designer for Exchange Server 2000 makes building and deploying your workflow applications easier. The graphical user interface (GUI), shown earlier in Figure 19-14, simplifies the process of visualizing your workflow while automating the process of creating the event handler registrations and action tables in your workflow-enabled folders.

I won't cover all the GUI elements of the Workflow Designer; you can find that information easily in the Exchange Server 2000 documentation or just read the Workflow Designer screen. However, I will cover three important elements that make up your workflow in the designer: states, actions, and script. The states are the boxes in the GUI that indicate whether your workflow is pending, approved, rejected, or expired. The actions provide the transitions between the states. These actions, which I'll detail momentarily, can be found in your action table and include OnEnter, OnExit, OnCreate, among others. The script implements the condition checking and the actions.

The states in the Workflow Designer are not very interesting because they don't perform any task; they only serve as a destination for the workflow item to move to during the workflow process. Therefore, your only concern with states should be that your scripts allow you to check which state the workflow item is in, to see whether it's being approved or rejected or placed in any other state you specify.

Actions are one of the most important aspects of the Workflow Designer since they are associated with the transitions between states. These transitions define how the work in your workflow is performed. Figure 19-12 shows how to create actions in the Workflow Designer.

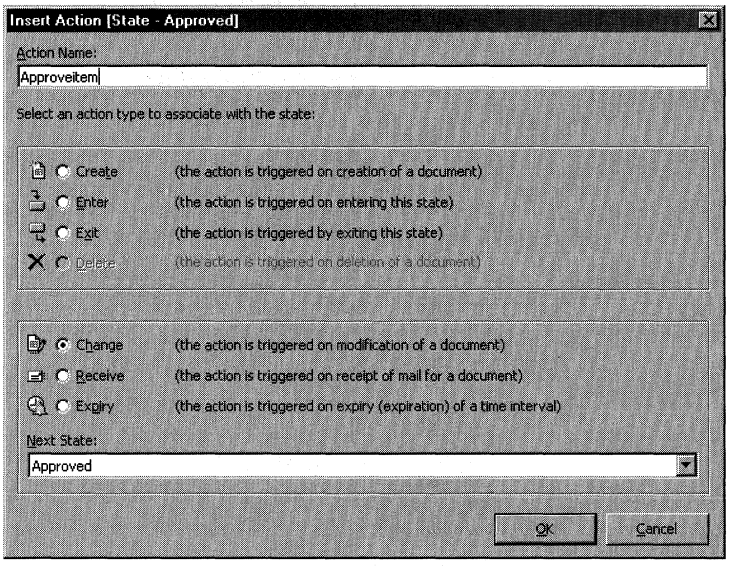


Figure 19-12. *Creating actions in the Workflow Designer.*

Table 19-10 presents the actions in the Workflow Designer and their uses. Be aware that some actions, such as the OnChange action, can appear multiple times on a state. But each time it appears for the same state, it must have different conditions that make the action valid for a workflow item—for example, a property changes value.

Workflow Designer Name	*Action Table Name*	*Description*
Create	OnCreate	An item was created. You must have at least one Create action in your workflow; otherwise, no items can be created in the folder.
Enter	OnEnter	This action manages the time used by the Expiry action. As soon as the state is entered using this action, the timer is started for the expiration interval you set. Entry into states is implicitly allowed, so you don't have to add this action to every state. You will normally use this action for timer-based workflow requirements only.
Exit	OnExit	The state is transitioning to a new state.

Table 19-10. *Actions of the Workflow Designer.*

Table 19-10 *continued*

Workflow Designer Name	*Action Table Name*	*Description*
Delete	OnDelete	The document is deleted. If you do not have a Delete action, workflow items cannot be deleted. If that's the case, Exchange Server will return an error if an application or user attempts to delete an item.
Change	OnChange	The document is modified. You can have multiple Change actions on a single state. If you have no Change actions, documents cannot be modified.
Receive	OnReceive	The workflow has received an e-mail message that correlates to a workflow item. This action allows your workflow to respond to e-mail.
Expiry	OnExpiry	The document has passed its time limit for the current state. This action is useful for time-based tasks, such as reminder notifications to managers to approve workflow items.

The following code is the XML representation of a real action table generated by the Workflow Designer. Notice the action and state names in the XML code. You'll see how to use XML action tables to simplify deploying workflow processes in the "Deploying Workflow Solutions" section.

```
<xml xmlns:s='uuid:BDC6E3F0-6DA3-11d1-A2A3-00AA00C14882'
   xmlns:dt='uuid:C2F41010-65B3-11d1-A29F-00AA00C14882'
   xmlns:rs='urn:schemas-microsoft-com:rowset'
   xmlns:z='#RowsetSchema'>
<s:Schema id='RowsetSchema'>
   <s:ElementType name='row' content='eltOnly' rs:updatable='true'>
      <s:AttributeType name='ID' rs:number='1' rs:write='true'>
         <s:datatype dt:type='string' dt:maxLength='4294967295'
            rs:precision='0' rs:long='true' rs:maybenull='false'/>
      </s:AttributeType>
      <s:AttributeType name='Caption' rs:number='2' rs:write='true'>
         <s:datatype dt:type='string' dt:maxLength='4294967295'
            rs:precision='0' rs:long='true' rs:maybenull='false'/>
      </s:AttributeType>
      <s:AttributeType name='State' rs:number='3' rs:write='true'>
         <s:datatype dt:type='string' dt:maxLength='4294967295'
            rs:precision='0' rs:long='true' rs:maybenull='false'/>
```

(continued)

```
</s:AttributeType>
<s:AttributeType name='NewState' rs:number='4' rs:write='true'>
   <s:datatype dt:type='string' dt:maxLength='4294967295'
      rs:precision='0' rs:long='true' rs:maybenull='false'/>
</s:AttributeType>
<s:AttributeType name='EventType' rs:number='5' rs:write='true'>
   <s:datatype dt:type='string' dt:maxLength='4294967295'
      rs:precision='0' rs:long='true' rs:maybenull='false'/>
</s:AttributeType>
<s:AttributeType name='Condition' rs:number='6' rs:write='true'>
   <s:datatype dt:type='string' dt:maxLength='4294967295'
      rs:precision='0' rs:long='true' rs:maybenull='false'/>
</s:AttributeType>
<s:AttributeType name='Action' rs:number='7' rs:write='true'>
   <s:datatype dt:type='string' dt:maxLength='4294967295'
      rs:precision='0' rs:long='true' rs:maybenull='false'/>
</s:AttributeType>
<s:AttributeType name='ExpiryInterval' rs:number='8' rs:write='true'>
   <s:datatype dt:type='string' dt:maxLength='4294967295'
      rs:precision='0' rs:long='true' rs:maybenull='false'/>
</s:AttributeType>
<s:AttributeType name='RowACL' rs:number='9' rs:write='true'>
   <s:datatype dt:type='string' dt:maxLength='4294967295'
      rs:precision='0' rs:long='true' rs:maybenull='false'/>
</s:AttributeType>
<s:AttributeType name='TransitionACL' rs:number='10' rs:write='true'>
   <s:datatype dt:type='string' dt:maxLength='4294967295'
      rs:precision='0' rs:long='true' rs:maybenull='false'/>
</s:AttributeType>
<s:AttributeType name='DesignToolFields' rs:number='11'
   rs:write='true'>
   <s:datatype dt:type='string' dt:maxLength='4294967295'
      rs:precision='0' rs:long='true' rs:maybenull='false'/>
</s:AttributeType>
<s:AttributeType name='CompensatingAction' rs:number='12'
   rs:write='true'>
   <s:datatype dt:type='string' dt:maxLength='4294967295'
      rs:precision='0' rs:long='true' rs:maybenull='false'/>
</s:AttributeType>
<s:AttributeType name='Flags' rs:number='13' rs:write='true'>
   <s:datatype dt:type='string' dt:maxLength='4294967295'
      rs:precision='0' rs:long='true' rs:maybenull='false'/>
</s:AttributeType>
<s:AttributeType name='EvaluationOrder' rs:number='14'
   rs:write='true'>
   <s:datatype dt:type='string' dt:maxLength='4294967295'
      rs:precision='0' rs:long='true' rs:maybenull='false'/>
</s:AttributeType>
```

```
      <s:extends type='rs:rowbase'/>
   </s:ElementType>
</s:Schema>
<rs:data>
   <rs:insert>
      <z:row ID='1' Caption='Create' State='' NewState='Pending'
         EventType='OnCreate' Condition='TRUE' Action=''
         ExpiryInterval='0' RowACL='' TransitionACL=''
             DesignToolFields='-1:1:' CompensatingAction='' Flags='0'
             EvaluationOrder='1000'/>
      <z:row ID='2' Caption='Delete' State='Pending' NewState=''
         EventType='OnDelete' Condition='true' Action=''
         ExpiryInterval='0' RowACL='' TransitionACL=''
         DesignToolFields='1:-2:' CompensatingAction='' Flags='0'
         EvaluationOrder='7000'/>
      <z:row ID='3' Caption='StartTimer' State='' NewState='Pending'
         EventType='OnEnter' Condition='TRUE' Action='sendMailToManager'
         ExpiryInterval='15' RowACL='' TransitionACL=''
         DesignToolFields='0:1:' CompensatingAction='' Flags='0'
         EvaluationOrder=''/>
      <z:row ID='5' Caption='ManagerApproved' State='Pending'
         NewState='Approved' EventType='OnChange'
         Condition='workflowsession.fields
            (&#x22;http://thomriz.com/schema/approvalstatus&#x22;).value =
            &#x22;Approved&#x22;'
          Action='strCourseName = GetCourseName
strStudentEmail = GetStudentEmail
strManagerEmail = GetManagerEmail
strBody = &#x22;Your manager approved you for the course: &#x22; &#x26;
   strCourseName
sendMail strBody,strStudentEmail &#x26; &#x22;,&#x22; &#x26;
   strManagerEmail,&#x22;Approved for course: &#x22; &#x26; strCourseName
addregistration
sendcalendarmessage'
   ExpiryInterval='0' RowACL='' TransitionACL='' DesignToolFields=
      '1:3:' CompensatingAction='' Flags='0' EvaluationOrder='3001'/>
      <z:row ID='6' Caption='ManagerRejected' State='Pending'
         NewState='Rejected' EventType='OnChange'
         Condition='workflowsession.fields
            (&#x22;http://thomriz.com/schema/approvalstatus&#x22;).value =
            &#x22;Rejected&#x22;'
          Action='strCourseName = GetCourseName
strStudentEmail = GetStudentEmail
strManagerEmail = GetManagerEmail
strBody = &#x22;Your manager rejected you for the course: &#x22;
   &#x26; strCourseName
sendMail strBody,strStudentEmail &#x26; &#x22;,&#x22; &#x26;
```

(continued)

```
   strManagerEmail,&#x22;Rejected for course: &#x22; &#x26; strCourseName '
   ExpiryInterval='0' RowACL='' TransitionACL='' DesignToolFields=
      '1:2:' CompensatingAction='' Flags='0' EvaluationOrder='3000'/>
      <z:row ID='7' Caption='NoResponse' State='Pending' NewState='Expired'
         EventType='OnExpiry' Condition='TRUE'
         Action='strCourseName = GetCourseName
strStudentEmail = GetStudentEmail
strManagerEmail = GetManagerEmail
strBody = &#x22;Your manager did not approve your attending of the course:
   &#x22; &#x26; strCourseName &#x26; &#x22; in enough time.  You will not
   be registered for this course.&#x22;
sendMail strBody, strStudentEmail &#x26; &#x22;,&#x22; &#x26;
   strManagerEmail, &#x22;Approval not received for course: &#x22; &#x26;
   strCourseName
'
   ExpiryInterval='0' RowACL='' TransitionACL='' DesignToolFields=
      '1:4:' CompensatingAction='' Flags='0' EvaluationOrder='5000'/>
      <z:row ID='8' Caption='' State='Rejected' NewState=''
         EventType='OnDelete' Condition='TRUE' Action='' ExpiryInterval=''
   RowACL='' TransitionACL='' DesignToolFields='2:-2:'
      CompensatingAction='' Flags='0' EvaluationOrder='7001'/>
      <z:row ID='10' Caption='' State='Approved' NewState=''
         EventType='OnDelete' Condition='TRUE' Action='' ExpiryInterval=''
   RowACL='' TransitionACL='' DesignToolFields='3:-2:'
       CompensatingAction='' Flags='0' EvaluationOrder='7002'/>
   <z:row ID='11' Caption='' State='Expired' NewState=''
      EventType='OnDelete' Condition='TRUE' Action='' ExpiryInterval=''
   RowACL='' TransitionACL='' DesignToolFields='4:-2:'
      CompensatingAction='' Flags='0' EvaluationOrder='7003'/>
   </rs:insert>
</rs:data>
</xml>
```

Instead of using the Workflow Designer to create your workflow process, you could programmatically create your action table by using just ADO and CDOWF. However, in most cases, you'll want to take advantage of the Workflow Designer and generate your action tables to XML, as shown in the preceding example. You can then import the XML action table into ADO and use that data to programmatically generate your workflow process.

Creating Event Scripts

You have a workflow engine, event handlers, and an action table, but you don't have a true workflow application yet. The real foundation of the workflow application is the script you write for the actions in your action table. Whether you want to send a message, change a property, or update an item, you need to implement this action in your script. Writing your workflow script is pretty straightforward; you'll probably

call ADO or CDO to perform functions in Exchange Server. The Workflow Designer includes a script editor, shown in Figure 19-13.

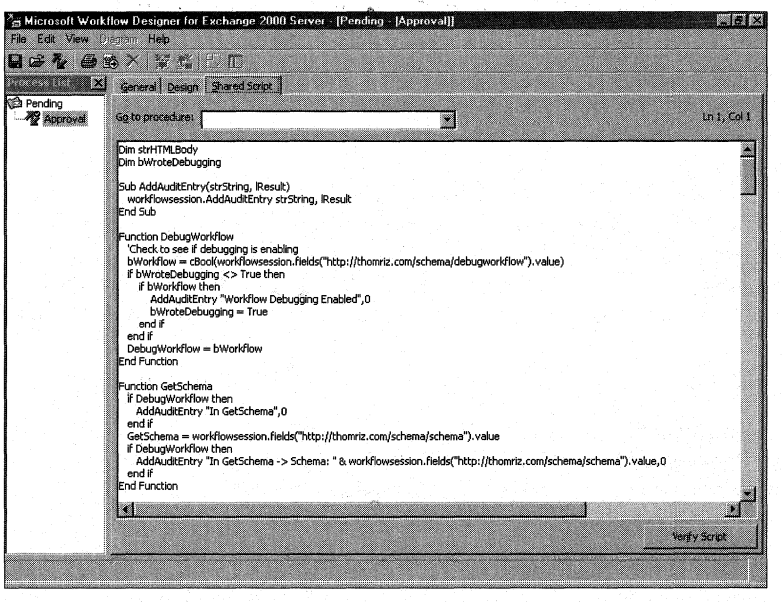


Figure 19-13. *The script editor built into the Workflow Designer.*

You don't have to use the Workflow Designer script editor to write your scripts. You instead can use another editor, such as Visual InterDev, and save the scripts to a common location or even write the handlers for your actions using COM components. You can then point the Workflow Designer, or a workflow process you programmatically create, to this common script file. The following script file is used in the Training application to implement the workflow process:

```
Dim strHTMLBody
Dim bWroteDebugging

Sub AddAuditEntry(strString, lResult)
   workflowsession.AddAuditEntry strString, lResult
End Sub

Function DebugWorkflow
   'Check to see whether debugging is enabled
   bWorkflow = cBool(workflowsession.fields _ &
      ("http://thomriz.com/schema/debugworkflow").value)
   if bWroteDebugging <> True then
     if bWorkflow then
         AddAuditEntry "Workflow Debugging Enabled",0
         bWroteDebugging = True
```

(continued)

```
        end if
    end if
    DebugWorkflow = bWorkflow
End Function

Function GetSchema
    if DebugWorkflow then
        AddAuditEntry "In GetSchema",0
    end if
    GetSchema = _
        workflowsession.fields("http://thomriz.com/schema/schema").value
    if DebugWorkflow then
        AddAuditEntry "In GetSchema -> Schema: " & _
            workflowsession.fields("http://thomriz.com/schema/schema").value,0
    end if
End Function

Function GetWorkflowSessionField(strField)
    if DebugWorkflow then
        AddAuditEntry "In GetWorkflowSessionField -> Value: " & strField,0
    end if
    GetWorkflowSessionField = _
        workflowsession.fields("http://thomriz.com/schema/" & strField).value
End Function

Function GetCourse(bReadOnly)
    if DebugWorkflow then
        AddAuditEntry "In GetCourse",0
    end if
    set oRec = CreateObject("ADODB.Record")
    if DebugWorkflow then
        AddAuditEntry "Course URL: " & workflowsession.fields _ &
            ("http://thomriz.com/schema/fullcourseurl").value,0
    end if
    if bReadOnly then
        iAccess = 1
    else
        iAccess = 3
    end if
    oRec.Open workflowsession.fields("http://thomriz.com/sche" & _
        " ma/fullcourseurl").value,workflowsession.ActiveConnection,iAccess
    if DebugWorkflow then
        AddAuditEntry "In GetCourse -> CourseName: " & _
            oRec.Fields("urn:schemas:httpmail:subject").value,0
    end if
    Set GetCourse = oRec
End Function
```

```
Function GetCourseName
   'Returns the name of the course
   set oRec = GetCourse(True)
   if DebugWorkflow then
      AddAuditEntry "In GetCourseName -> CourseName = " & _
         oRec.Fields("urn:schemas:httpmail:subject").value,0
   end if
   GetCourseName = oRec.Fields("urn:schemas:httpmail:subject").value
End Function

Sub ReplaceString(strToken, strReplacement)
   'Take the token and replace it in the global strHTMLBody
   strHTMLBody = Replace(strHTMLBody, strToken, strReplacement)
End Sub

Function GenerateHTMLBody()
   'Generates the HTML to send in the message.
   'Retrieve the message containing the HTML.
   'The HTML template must always be called WorkflowMessage.
   if DebugWorkflow then
      AddAuditEntry "In GenerateHTMLBody",0
   end if
   strHTMLBody = ""

   'Build the SQL statement.
   'Query for the e-mail message.
   strEmailsFolderPath = GetWorkflowSessionField("stremailsfolderpath")
   strSQL = "Select ""urn:schemas:httpmail:textdescription"" From " & _
      "scope('shallow traversal of """ & strEmailsFolderPath & _
      """') Where ""DAV:iscollection"" = false AND ""DAV:ishidden"" = " & _
      "false AND ""urn:schemas:httpmail:subject"" LIKE '%WorkflowMessage%'"

   'Create a new Recordset object
   Set rst = CreateObject("ADODB.Recordset")
   With rst
      'Open Recordset based on the SQL string
      .Open strSQL, workflowsession.ActiveConnection
   End With

   If rst.BOF And rst.EOF Then
      GenerateHTMLBody = ""
      Exit Function
   End If

   'On Error Resume Next
   rst.MoveFirst
   strHTMLBody = rst.Fields("urn:schemas:httpmail:textdescription").Value
```

(continued)

```
    'Get the course
    set oCourse = GetCourse(True)
    'Load it into CDO Appointment
    set iAppt = CreateObject("CDO.Appointment")
    iAppt.DataSource.Open _
       oCourse.Fields("DAV:href").value,workflowsession.ActiveConnection,1

    strSchema = GetSchema
    'Replace the tokens with real values
    ReplaceString "%StudentName%", _
       GetWorkflowSessionField("strstudentfullname")
    ReplaceString "%Name%", _
       iAppt.Fields("urn:schemas:httpmail:subject").Value
    ReplaceString "%Category%", iAppt.Fields(strSchema & "category").Value
    strDate = Month(iAppt.StartTime) & "/" & Day(iAppt.StartTime) & "/" & _
       Year(iAppt.StartTime)
    ReplaceString "%Date%", strDate
    ReplaceString "%StartTime%", TimeValue(iAppt.StartTime)
    ReplaceString "%EndTime%", TimeValue(iAppt.EndTime)
    ReplaceString "%Location%", iAppt.Location
    ReplaceString "%Description%", iAppt.TextBody

    strHTTPURL = GetWorkflowSessionField("strrootdirectory") & _
       "workflow.asp?CourseID=" & iAppt.Fields("DAV:href") & _
       "&student=" & GetWorkflowSessionField("fullstudenturl")
    ReplaceString "%URLLink%", strHTTPURL

    If strHTMLBody <> "" Then
       GenerateHTMLBody = strHTMLBody
    Else
       GenerateHTMLBody = ""
    End If

    rst.Close

    Set rst = Nothing

End Function

Sub sendMail(strMsg , strAddress, strSubject)
    set msg = createobject("CDO.message")
    msg.To = strAddress
    msg.From = GetWorkflowSessionField("notificationaddress")
    msg.subject = strSubject
    msg.textbody = strMsg
    if DebugWorkflow then
       AddAuditEntry "In SendMail: Address -> " & strAddress & vblf & _
          "Subject -> " & strSubject & vblf & "Message: " & strMsg,0
    end if
```

```
   msg.send
End sub

Function GetStudentEmail
   if DebugWorkflow then
      AddAuditEntry "In GetStudentEmail",0
   end if
   GetStudentEmail = GetWorkflowSessionField("studentemail")
End Function

Function GetManagerEmail
   if DebugWorkflow then
      AddAuditEntry "In GetManagerEmail",0
   end if
   GetManagerEmail = GetWorkflowSessionField("manageremail")
End Function

Sub SendMailToManager
   'Get the manager's e-mail
   strManagerEmail = GetWorkflowSessionField("manageremail")
   if DebugWorkflow then
      AddAuditEntry "In SendMailToManager -> Manager: " & strManagerEmail,0
   end if
   set oMsg = createobject("CDO.message")
   set oRecord = GetCourse(True)
   oMsg.To = strManagerEmail
   oMsg.From = GetWorkflowSessionField("notificationaddress")
   oMsg.Subject = "Approval Required for course:  " & _
      oRecord.Fields("urn:schemas:httpmail:subject").value
   oMsg.AutoGenerateTextBody = True
   oMsg.MimeFormatted = True
   oMsg.HTMLBody = GenerateHTMLBody
   oMsg.Send
End Sub

Function GetStudent(bReadOnly)
   if DebugWorkflow then
      AddAuditEntry "In GetStudent",0
   end if
   set oRec = CreateObject("ADODB.Record")
   if DebugWorkflow then
      AddAuditEntry "Student URL: " & workflowsession.fields _
         ("http://thomriz.com/schema/fullstudenturl").value,0
   end if
   if bReadOnly then
       iAccess = 1
   else
       iAccess = 3
   end if
```

(continued)

```
    oRec.Open workflowsession.fields("http://thomriz.com/sche" & _
       "ma/fullstudenturl").value,workflowsession.ActiveConnection,iAccess
    if DebugWorkflow then
       AddAuditEntry "In GetStudent -> StudentName: " & _
          oRec.Fields("urn:schemas:httpmail:subject").value,0
    end if
    Set GetStudent = oRec
End Function



Sub addregistration
    set oRecord = GetStudent(False)
    strSchema = GetSchema
    strCourseURL = GetWorkflowSessionField("shortcourseurl")
    oRecord.Fields(strSchema & "registrations") = _
       oRecord.Fields(strSchema & "registrations") & strCourseURL & ","
    oRecord.Fields.Update
    oRecord.Close
    set oRecord = Nothing
End Sub

Sub sendcalendarmessage
    if DebugWorkflow then
        AddAuditEntry "In SendCalendarMessage",0
    end if
    'Get the original appointment
    set oOriginalAppt = CreateObject("CDO.Appointment")
    oOriginalAppt.Datasource.Open _
       GetWorkflowSessionField("fullcourseurl"),_
       workflowsession.ActiveConnection,1

    'Create a throwaway appointment
    set oAppt = CreateObject("CDO.Appointment")
    set oConfig = CreateObject("CDO.Configuration")
    strNotificationAddress = GetWorkflowSessionField("notificationaddress")
    oConfig.Fields("http://schemas.microsoft.com/cdo/config" & _
       "uration/sendemailaddress") = strNotificationAddress
    oConfig.Fields.Update

    oAppt.Configuration = oConfig

    oAppt.StartTime = oOriginalAppt.StartTime
    oAppt.EndTime = oOriginalAppt.EndTime
    oAppt.Subject = "Course: " & oOriginalAppt.Subject
    oAppt.Location = oOriginalAppt.Location
    strSchema = GetSchema
```

```
    oAppt.TextBody = "The Instructor is " & _
       oOriginalAppt.Fields(strSchema & "instructoremail").Value
    'Don't ask for a response since we don't care if they accept or decline
    oAppt.ResponseRequested = False

    Set oAttendee = oAppt.Attendees.Add
    strEmail = GetStudentEmail

    oAttendee.Address = strEmail
    oAttendee.Role = 0

    Set oMtg = oAppt.CreateRequest
    oMtg.Message.Send

end sub
```

This script uses ADO and CDO to perform its functions. However, another object is at work in the script: WorkflowSession. This intrinsic object (which you don't have to create) is passed to your script by the workflow engine. It allows you to access properties on the process instance as well as the audit trail specified for the workflow. Table 19-11 shows the most important properties and methods of this object. For more information on the properties and methods, refer to the Exchange 2000 Platform SDK.

I want to offer more explanation for some of the properties and methods in Table 19-11. One method that will be very useful for you in your development is the *GetUserProperty* method. This method takes three parameters. The first is the distinguished name of the object in the Active Directory, which can be either the Active Directory path to the object or the unique e-mail address of the object. The second parameter is the Active Directory attribute you want to get off the object. The third parameter works in conjunction with the first and tells CDO whether the first parameter is an Active Directory path (1) or an e-mail address (0). Probably the most common use for this method is to retrieve the manager of the owner of the item that is undergoing the workflow to get approval. You would retrieve the manager by getting the *manager* property off the current user's Active Directory object. You can get the e-mail address of the current user by using the *WorkflowSession.Sender* property. You can then retrieve the *mail* property from the manager's Active Directory object. The *manager* property will return to you the Active Directory path to the manager. The following example illustrates this scenario:

```
with WorkflowSession
   strUserAddress = "username@company.com"
   mgrDN = .GetUserProperty(strUserAddress,"manager",0)
   strUserMgrEmail = .GetUserProperty(mgrDN, "mail", 1)
end with
```

Property or Method Name	*Description*
ActiveConnection	This property returns an ADO Connection object. You should use this Connection object in your script's ADO and CDO functions, especially if you want them to take part in transactions.
AddAuditEntry	This method allows you to add an audit entry to the selected audit entry provider of the workflow process. You pass a string and a Long value to specify what the entry should say and the custom result you want for the value. By default, Exchange Server ships with one audit trail provider which writes to the Windows 2000 Event Log. You can create custom audit trail providers by creating COM components that implement the *IAuditTrail* interface.
DeleteReceivedMessage	This method deletes the received e-mail, if one exists, for the workflow item. You will usually call it in the Receive action.
DeleteWorkflowItem	This method deletes the workflow item.
Domain	This property returns the domain of the server. This property works in conjunction with the *Server* property to make it easier for you to generate file:// or http:// URLs.
ErrorDescription	This property is used in conjunction with the *ErrorNumber* property. *ErrorDescription* contains a description of the error to report to the audit trail provider.
ErrorNumber	This property holds the number of the errors to report to the client and the audit trail provider.
Fields	This property returns the ADO Fields collection for the workflow item. Using Fields, you can access built-in and custom schema on the workflow item.
GetNewWorkflowMessage	This method creates and returns a new WorkflowMessage object. The WorkflowMessage object allows you to send e-mail messages from restricted workflows, since you cannot create a CDO Message object in a restricted workflow.
GetUserProperty	This method gets an Active Directory attribute off an Active Directory object.

Table 19-11. *Properties and methods of the Workflowsession object.*

Table 19-11 *continued*

Property or Method Name	*Description*
IsUserInRole	This method checks to see whether a user is in a folder role. You pass to this method the user's e-mail address and the name of the role. The method will return a Boolean indicating whether that user is in that particular folder role. A folder role is a grouping of users who perform a particular function that you define for the folder. The roles are stored on the folder, so to implement roles-based workflow, you do not need permissions to modify or add properties to the Active Directory.
ItemAuthors	Since Exchange Server 2000 supports item-level permissions, you might want to set such permissions on workflow items. This property contains a collection that contains a list of all users with authoring ability on the workflow item.
ItemReaders	This property contains a collection of users who should have Reader permissions on the workflow item.
Properties	This property returns an *ISessionProps* interface to you so that you can add properties you need persisted for a single session that lasts for one *ProcessInstance* transition. Here's a good example of using this property: Suppose you have multiple actions that need to be evaluated to make a state transition. You do not want each action to check multiple times whether a certain property on the item already exists as part of the evaluation criteria. So use this property to cache the value and share the value between multiple condition scripts.
ReceivedMessage	This property returns the e-mail message that was received in correlation to a workflow item.
Sender	This property contains the SMTP address of the person who initiated the state transition.
Server	This property contains the name of the server and is used in conjunction with the *Domain* property.
StateFrom	This property contains the name of the state before the current process transition.
StateTo	This property holds the name of the state after the current transition.
TrackingTable	Used with e-mail workflows, this property contains a Recordset object that has a number of properties relating to the current workflow item. Refer to the Exchange 2000 Platform SDK for more information on this property.

Two properties in Table 19-11, *ItemAuthors* and *ItemReaders*, also demand more explanation. *ItemAuthors* is a collection used to specify per-item modify and delete permissions. If you add any users to this collection, only those users can modify or delete the item as well as read it. If you remove all users from this collection, the default permissions on the folder apply.

With *ItemReaders*, you can specify per-item read access. If you add users to this collection, only those users can read or view the item, but they cannot necessarily modify the item. This means that even when other users query, know the URL of the item, or try to retrieve a specific property on the item, they will not be able to modify the item unless they are in the ItemReaders collection. When you clear the collection, default folder permissions will apply.

Both *ItemAuthors* and *ItemReaders* return an *IMembers* interface. This interface supports one property and three methods: the *Count* property, and the *Add*, *Clear*, and *Delete* methods. *Count* returns the number of members in the collection. *Add* adds a new member by taking two parameters, *Name* and *Type*. *Name* must be a string that either specifies the e-mail address of the user or a role. Exchange supports the string literals "*Role 1*" through "*Role 16*" for adding roles. The *Type* parameter is an integer that specifies the type of user you are adding, whether it is an e-mail address (0) or a role (1). The *Clear* method clears all members from the collection. Finally, the *Delete* method deletes a member from the collection. You need to pass either a numbered index into the collection or a string that uniquely identifies a member of the collection. This string can be a role name such as "*Role 1*" or the e-mail address of the user you want to remove from the collection. The following example shows you how to add two different users to the *ItemAuthors* and *ItemReaders* collections on a workflow item.

```
strAddress = "user@domain.com"
WorkflowSession.ItemAuthors.Add strAddress, 0  'cdowfEmailAddress
strAddress = "user2@domain.com"
WorkflowSession.ItemReaders.Add strAddress, 0  'cdowfEmailAddress
```

DEBUGGING YOUR WORKFLOW SOLUTIONS

You can debug your workflow solutions by using either the audit trail provider included with Exchange 2000 or script debugging. To enable script debugging, you need to either select the script debugging option in the Workflow Designer or set to *True* the property on your workflow's event handler registration called *http://schemas.microsoft.com/cdo/workflow/enabledebug*. In order for the debugger to work, you must make sure that just-in-time (JIT) debugging is enabled in Windows 2000. You can do this by modifying a key in the registry under HKCU/Software/Microsoft/Windows Script/Settings/JITDebug and setting it to 1.

Deploying Workflow Solutions

Once you've drawn out your process, implemented your conditions and actions, and written your script, the next step is to deploy your workflow process to a folder. The Workflow Designer makes this step easy because you can save the workflow process into any folder in which you have permissions to create a workflow. Figure 19-14 shows how to Save Workflow Process To Folder in the Workflow Designer.

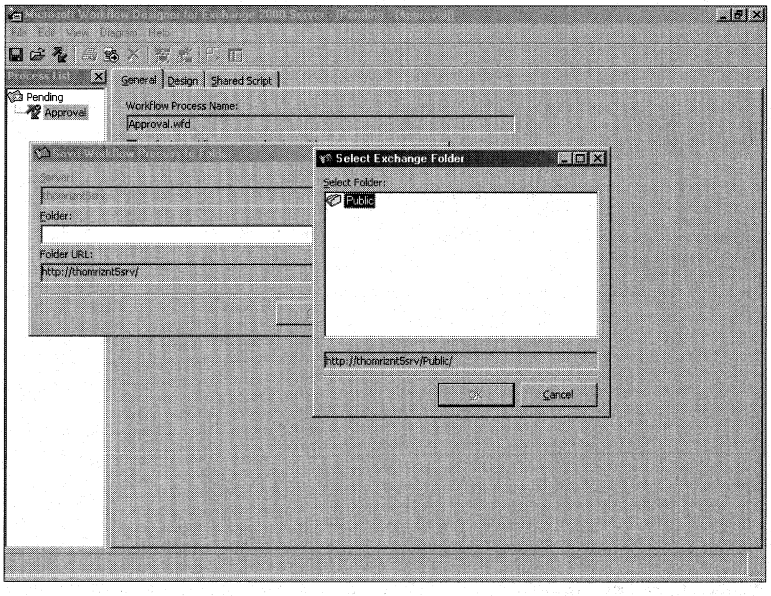


Figure 19-14. *The Save Workflow Process To Folder tool makes it easy to deploy workflow solutions.*

In some cases, you might need to programmatically deploy your solutions. Unfortunately, the Workflow Designer doesn't have an object model that you can automate to use the Save Workflow Process To Folder tool. Instead, you'll need to write some code to deploy your workflow process. If so, you should first use the Workflow Designer to export to XML the action table for your workflow process.

The following code, taken from the Training application setup program, shows how to deploy your workflow application programmatically. You'll notice the following steps in the code:

1. Create your common script file (if necessary).
2. Create a workflow ProcessDefinition object.
3. In the ProcessDefinition object, add your action table. To do so, create a new Recordset object, and use the XML features of ADO to load the XML version of the action table that the Workflow Designer saved for you.

4. As part of creating your ProcessDefinition object, select your audit trail provider, set the location of your common script file, and set the mode that the workflow process should run under (restricted or privileged).
5. Save the ProcessDefinition object into the folder.
6. Create the event registration items for the OnSyncSave, OnSyncDelete, and OnTimer events. On the server events registration item for OnSyncSave and OnSyncDelete, set the properties in the *http://schemas.microsoft.com/cdo/workflow/* name space—for example, the pointer to the default process definition for the folder, whether ad hoc workflows are allowed in the folder, whether to enable script debugging, and whether to log to the audit trail provider successful state transitions.

```
Private Sub AddWorkflowProcess()
   Dim oRS As New ADODB.Recordset
   Dim oPD As New CDOWF.ProcessDefinition

   On Error GoTo errHandler

   'Add the common script file
   Dim oScriptRec As New ADODB.Record
   Dim oStream As New ADODB.Stream
   'Load the script file
   Dim fso As New Scripting.FileSystemObject
   Dim ofile As TextStream

   Set ofile = fso.OpenTextFile(App.Path & "\commonscript.txt")

   strCommonScript = ofile.ReadAll

   oScriptRec.Open strPath & "/Pending/commonscript", oConnection, _
      adModeReadWrite, adCreateNonCollection
   oStream.Open oScriptRec, adModeReadWrite, adOpenStreamFromRecord
   With oStream
      .Charset = "unicode"
      .Type = adTypeText
      .Position = 0
      .SetEOS
      .WriteText strCommonScript
      .Position = 0
      .Flush
      .Close
   End With
   strScriptURL = oScriptRec.Fields("DAV:href").Value

   'Load the action table
   oRS.Open App.Path & "\actiontable.xml"
```

```
With oPD
   .ActionTable = oRS
   .AuditTrailProvider = "CDOWF.AuditTrailEventLog"
   .CommonScriptURL = strScriptURL
   .Mode = cdowfPrivilegedMode
   .Fields("DAV:ishidden") = True
End With

oPD.DataSource.SaveTo strPath & "/Pending/WFDEF", oConnection, _
   adModeReadWrite, adCreateNonCollection

strPDHREF = oPD.Fields("DAV:href").Value

'Create the event registrations

'First create the timer event
arrRequired = GenerateRequiredEventArray("", "ontimer", _
   "CdoWfEvt.EventSink.1", "", "")
strNow = Now
arrOptional = GenerateOptionalEventArray("", "", "", "", 15, _
   strNow, "")

CreateEvtRegistration oConnection, strPath & "Pending/timer", _
   arrRequired, arrOptional, True

'Create the OnSyncSave and onSyncDelete registration
arrRequired = GenerateRequiredEventArray("WHERE ""DAV:ishidden"" = _
   false AND ""DAV:isfolder"" = false", "onsyncsave;onsyncdelete", _
   "CdoWfEvt.EventSink.1", "", "")
'Create a new array and add some further properties for workflow
Dim arrWorkflowRequired(6, 1)
For i = LBound(arrRequired) To UBound(arrRequired)
   arrWorkflowRequired(i, 0) = arrRequired(i, 0)
   arrWorkflowRequired(i, 1) = arrRequired(i, 1)
Next
'Add workflow properties
arrWorkflowRequired(3, 0) = _
   "http://schemas.microsoft.com/cdo/workflow/defaultprocdefinition"
arrWorkflowRequired(3, 1) = strPDHREF
arrWorkflowRequired(4, 0) = _
   "http://schemas.microsoft.com/cdo/workflow/adhocflows"
arrWorkflowRequired(4, 1) = 0
arrWorkflowRequired(5, 0) = _
   "http://schemas.microsoft.com/cdo/workflow/enabledebug"
arrWorkflowRequired(5, 1) = False
arrWorkflowRequired(6, 0) = _
   "http://schemas.microsoft.com/cdo/workflow/disablesuccessentries"
arrWorkflowRequired(6, 1) = False   'Enable success entries
```

(continued)

```
    CreateEvtRegistration oConnection, strPath & "Pending/workflowreg", _
        arrWorkflowRequired, arrWorkflowOptional, True
    Exit Sub
errHandler:
    MsgBox "Error in AddWorkflowProcess.  Error " & Err.Number & " " & _
        Err.Description
    End
End Sub
```

Workflow Security and Deployment Gotchas

I want to point out one gotcha you should be aware of when deploying workflow solutions. It's a feature that can trip you up if you don't understand it. The workflow event handlers have two COM+ roles that they implement, which we saw earlier: *CanRegisterWorkflow* and *PrivilegedWorkflowAuthors*. If you don't understand what these roles are used for and how the workflow engine leverages them, you might run into some issues. This section outlines how these two roles and the workflow engine work together.

The *CanRegisterWorkflow* role is used when someone attempts to register for the *Workflow* event handler. The *Workflow* event handler implements *ICreateRegistration*, so when someone attempts to register for the *Workflow* event handler, the event handler is called to verify whether it wants to allow the registration to go through. The *Workflow* event handler calls the COM+ method *IsUserInRole(CanRegisterWorkflow)* to determine whether the user attempting the registration is authorized to do so. If this call returns *True*, the *Workflow* event handler allows the registration to go through.

The *PrivilegedWorkflowAuthors* role is used to ensure that any executable workflow code to be run in Privileged mode was not tampered with by an unauthorized person. Here's the scenario: User A has privileged permissions and registers a new workflow that contains a script to run in privileged mode. User B is allowed to write only sandboxed workflows, but he does have write access to user A's script file. User B later inserts malicious script into these workflow files, knowing it will be run in Privileged mode since User A has privileged workflow permissions.

To prevent this, at run time, the *Workflow* event sink checks to see which user last modified the process definition, the script, and the event handler registration item. For each of these SIDs, the event sink calls COM+ *IsUserInRole(PrivilegedWorkflowAuthors)*. If any of these documents were last modified by a nonprivileged person, the *Workflow* engine knows the files have been tampered with. The *Workflow* engine immediately stops execution and logs a security error.

So if you want to run privileged mode workflows, you must make sure that the account used to save all the critical documents, such as the process definition, scripts, and event registrations, must be members of the *PrivilegedWorkflowAuthors* role.

EXCHANGE 2000 AND SECURITY

One feature that Exchange 2000 offers is the ability to set permissions at both the item and property levels for documents or other sorts of objects contained in the Exchange 2000 database. This functionality is different from Exchange 5.5, which allows you to set permissions only at the folder level. Imagine what kind of impact this ability will have on your applications. Now you can secure your applications even further when data is stored in even a single folder. This section outlines how to use the new security features in Exchange 2000 and shows you a Web application that allows you to try out these new security features.

> **NOTE** If you set item level security in Exchange 2000 and the item is replicated to an Exchange 5.5. server, your security will not be enforced since Exchange 5.5 does not support item level security.

Security Features

Exchange 2000 supports native Windows 2000 security descriptors. Using these descriptors, you can allow or deny access to an item or the item's properties, grant this access using Windows 2000 security identifiers, and access and set permissions by viewing and modifying the descriptor in an XML format from WebDAV or ADO/OLEDB. By providing you with an XML representation of the security descriptor, Exchange 2000 makes it very easy for you to work with security settings; you do not have to learn Windows API programming to change permissions. Furthermore, the technology in Exchange 2000 takes your XML descriptor and turns it into the correct binary representation of the descriptor in the Exchange database. You can access the XML-formatted descriptor by querying for the property *http://schemas.microsoft.com/exchange/security/descriptor*. The following code is an example of what is returned on an item when you query for this property from ADO:

```
'ADO code
Dim oRecord As New ADODB.Record
oRecord.Open "file://./backofficestorage/domain/apps/items/exchange.eml"
```

Following are the XML results:

```
<S:security_descriptor
xmlns:S="http://schemas.microsoft.com/security/"
xmlns:D="urn:uuid:c2f41010-65b3-11d1-a29f-00aa00c14882/"
D:dt="microsoft.security_descriptor">
 <S:revision>1</S:revision>
 <S:owner S:defaulted="0">
  <S:sid>
   <S:string_sid>S-1-5-21-1659004503-152049171-1202660629-1110
   </S:string_sid>
```

(continued)

```
    <S:nt4_compatible_name>THOMRIZNT5DOM\thomriz</S:nt4_compatible_name>
    <S:user_principal_name>thomriz@thomriznt5dom.extest.microsoft.com</
S:user_
principal_name>
    <S:display_name>Thomas Rizzo</S:display_name>
   </S:sid>
  </S:owner>
  <S:primary_group S:defaulted="0">
   <S:sid>
    <S:string_sid>S-1-5-21-1659004503-152049171-1202660629-513
    </S:string_sid>
    <S:nt4_compatible_name>THOMRIZNT5DOM\Domain Users
    </S:nt4_compatible_name>
   </S:sid>
  </S:primary_group>
  <S:dacl S:defaulted="1" S:protected="0" S:autoinherited="1">
   <S:revision>2</S:revision>
   <S:effective_aces>
    <S:access_allowed_ace S:inherited="1">
     <S:access_mask>1fcfff</S:access_mask>
     <S:sid>
      <S:string_sid>S-1-5-21-1659004503-152049171-1202660629-1110
      </S:string_sid>
      <S:nt4_compatible_name>THOMRIZNT5DOM\thomriz</S:nt4_compatible_name>
      <S:user_principal_name>thomriz@thomriznt5dom.extest.microsoft.com
      </S:user_principal_name>
      <S:display_name>Thomas Rizzo</S:display_name>
     </S:sid>
    </S:access_allowed_ace>
    <S:access_allowed_ace S:inherited="1">
     <S:access_mask>1fcfff</S:access_mask>
     <S:sid>
      <S:string_sid>S-1-5-21-1659004503-152049171-1202660629-1105
      </S:string_sid>
      <S:nt4_compatible_name>THOMRIZNT5DOM\Domain EXServers
      </S:nt4_compatible_name>
     </S:sid>
    </S:access_allowed_ace>
    <S:access_allowed_ace S:inherited="1">
     <S:access_mask>1fcfff</S:access_mask>
     <S:sid>
      <S:string_sid>S-1-5-21-1659004503-152049171-1202660629-500
      </S:string_sid>
      <S:nt4_compatible_name>THOMRIZNT5DOM\Administrator</
S:nt4_compatible_name>
      <S:display_name>Administrator</S:display_name>
     </S:sid>
```

```
    </S:access_allowed_ace>
    <S:access_allowed_ace S:inherited="1">
     <S:access_mask>1fcfff</S:access_mask>
     <S:sid>
      <S:string_sid>S-1-5-21-1659004503-152049171-1202660629-519
      </S:string_sid>
      <S:nt4_compatible_name>THOMRIZNT5DOM\Enterprise Admins
      </S:nt4_compatible_name>
     </S:sid>
    </S:access_allowed_ace>
    <S:access_allowed_ace S:inherited="1">
     <S:access_mask>1fcfff</S:access_mask>
     <S:sid>
      <S:string_sid>S-1-5-21-1659004503-152049171-1202660629-512
      </S:string_sid>
      <S:nt4_compatible_name>THOMRIZNT5DOM\Domain Admins
      </S:nt4_compatible_name>
     </S:sid>
    </S:access_allowed_ace>
    <S:access_allowed_ace S:inherited="1">
     <S:access_mask>12088f</S:access_mask>
     <S:sid>
      <S:string_sid>S-1-1-0</S:string_sid>
      <S:nt4_compatible_name>\Everyone</S:nt4_compatible_name>
     </S:sid>
    </S:access_allowed_ace>
   </S:effective_aces>
  </S:dacl>
</S:security_descriptor>
```

This XML code is for a nonfolder security descriptor—specifically, for an item. A folder security descriptor is a little bit different, but as you can see in the XML structure shown in the code above, a security descriptor is made up of a Discretionary Access Control List (DACL), which in turn is made up of Access Control Entries (ACE). Each ACE in the DACL either grants or denies a trustee a certain set of rights to the object. The access mask, which you see defined in the XML, describes the set of rights that are granted or denied to a user. Access masks are 32-bit numbers where the upper 16 bits describe generic rights and the lower 16 bits describe object-specific rights.

Tables 19-12, 19-13, and 19-14 outline the different sets of access rights you can have: standard access rights, access rights on folders, and on nonfolders (items), respectively. Values from each table can be combined and put into the access mask to create the correct security descriptor for the user.

Name	*Value (Hex)*	*Description*
fsdrightDelete	*0x00010000*	Delete
fsdrightReadControl	*0x00020000*	Read access to security descriptor and owner
fsdrightWriteSD	*0x00040000*	Write DACL permissions
fsdrightWriteOwner	*0x00080000*	Used to assign write owner
fsdrightSynchronize	*0x00100000*	Used to synchronize access to the object

Table 19-12. *Standard rights (non-Exchange, Windows 2000).*

If you were to create an access mask with all the properties in Table 19-12, the value would be *0x1F0000*.

Name	*Value (Hex)*	*Description*
fsdrightListContents	*0x00000001*	Right to list contents of the directory.
fsdrightCreateItem	*0x00000002*	Right to add a file to the folder.
fsdrightCreateContainer	*0x00000004*	Right to add a subfolder
fsdrightReadProperty	*0x00000008*	Right to read extended attributes
fsdrightWriteProperty	*0x00000010*	Right to write extended attributes
fsdrightReadAttributes	*0x00000080*	Right to read file attributes. Currently unused.
fsdrightWriteAttributes	*0x00000100*	Right to change file attributes. Currently unused.
fsdrightWriteOwnProperty	*0x00000200*	Right to modify own items (Exchange-specific property)
fsdrightDeleteOwnItem	*0x00000400*	Right to delete own items
fsdrightViewItem	*0x00000800*	Right to view items (Exchange-specific property)
fsdrightOwner	*0x00004000*	Owner of the folder. Provided for backwards compatability.
fsdrightContact	*0x00008000*	Contact for the folder. Provided for backwards compatability.

Table 19-13. *Folder rights.*

If a user has all rights in Table 19-13, the value of the mask would be *0xCFFF*.

Name	*Value (Hex)*	*Description*
fsdrightReadBody	*0x00000001*	Right to read data from a file.
fsdrightWriteBody	*0x00000002*	Right to write data to a file.
fsdrightAppendMessage	*0x00000004*	Same as *fsdrightWriteBody*. Not currently used.
fsdrightReadProperty	*0x00000008*	Right to read extended attributes.
fsdrightWriteProperty	*0x00000010*	Right to write extended attributes.
fsdrightExecute	*0x00000020*	Right to execute a file. Currently not used.
fsdrightReadAttributes	*0x00000080*	Right to read file attributes.
fsdrightWriteAttributes	*0x00000100*	Right to change file attributes.
fsdrightWriteOwnProperty	*0x00000200*	Right to modify own item (Exchange-specific property).
fsdrightDeleteOwnItem	*0x00000400*	Right to delete own item (Exchange-specific property).
fsdrightViewItem	*0x00000800*	Right to view item (Exchange-specific property).

Table 19-14. *Item rights.*

If a user has all the rights in Table 19-14, the value would be *0x0FBF*.

As you can see in the preceding XML code example, you can have an access_allowed_ACE, which contains the access mask for the rights you are going to allow for the user on the item or folder. You can also have an access_denied_ACE, which specifies an access mask that contains the rights you are going to deny for the user on the folder or item. If the user has all rights, as in the preceding XML example, the access mask will be *1FCFFF*, which means all rights from all the tables above are granted to the user. Creating the access mask is just a matter of adding the hexidecimal values from the tables together and creating the ACE for the user. We'll see an example on how to create an ACE later in this chapter using the XMLDOM.

One fact you need to know even before we drill down into a sample application is that there are certain rights that the user needs to be able to access the security descriptor. These rights are *fsdrightReadControl*, *fsdrightWriteSD*, and *fsdrightWriteOwner*.

Sample Security Application

One of the sample applications included on the companion CD will, I know, make using the XML security descriptor much easier for you. The security sample application is a Web application that highly leverages the XMLDOM, XSL as well as WebDAV to show you how to work with and set the XML security descriptors in Exchange 2000. Figure 19-15 shows the main interface for the security application.

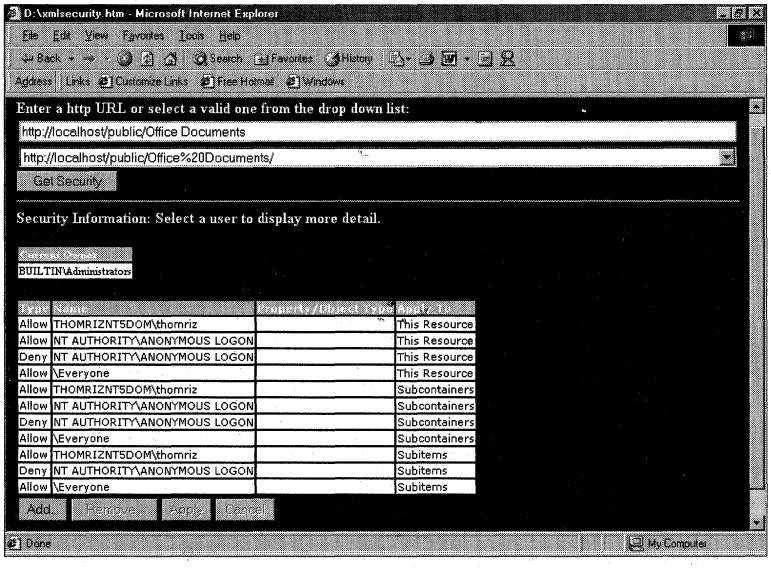


Figure 19-15. *The main interface for the security application.*

One task to try when working with the security application is restricting who can view items in an Exchange folder. The way to do this in the application is included in the next bit of code. As you can see in Figure 19-16, the user can view the document named ADO and Exchange 2000. However, after applying the setting that denies this user access to the document, as shown in Figure 19-17, the user can no longer see that specific document but can see all the other documents contained in the folder. Figure 19-17 shows what the user sees after applying the security change.

The application works by combining the XMLDOM, XSL, and WebDAV to get, display, and set both item-level and property-level security in Exchange. As you can see from the following code snippet, when you click the OK button to retrieve the security for an object in Exchange, the application does a PROPFIND on the *http://schemas.microsoft.com/exchange/security/descriptor* property. The application then uses XSL to display the HTML that you see in the browser.

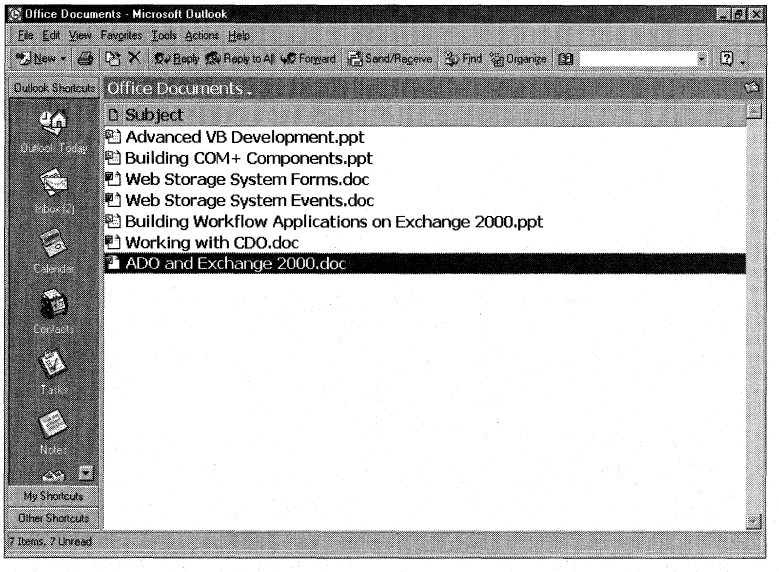


Figure 19-16. *Before applying the security settings, the user can view the full list of documents.*

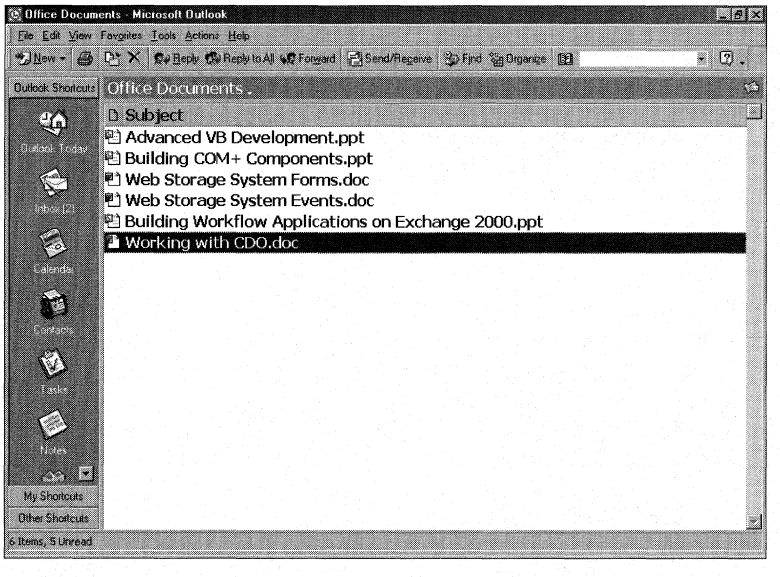


Figure 19-17. *After applying the security settings, the user can no longer view the restricted files.*

```
function cmdGetSec.onclick()
{
   var strReqBody = "";

   if(txtUrl.value == "")
   {
      alert("Please Enter or select a valid URL.");
      return;
   }

   strReqBody = "<?xml version=\"1.0\" encoding=\"utf-8\"?>" +
      "<propfind xmlns=\"DAV:\">" +
      "<prop xmlns:r=\"http://schemas.microsoft.com/exchange/security/\">"
      + "   <r:descriptor/>" +
      "</prop>" +
      "</propfind>"

   DAVRequest.open("PROPFIND", txtUrl.value, false);
   DAVRequest.setRequestHeader("Content-Type", "text/xml");
   DAVRequest.setRequestHeader("Depth", "0");
   DAVRequest.setRequestHeader("Translate", "f");
   DAVRequest.send(strReqBody);

   if(chkMultiStatusForErr(DAVRequest) >= 0)
   {

      xmlResponse = DAVRequest.responseXml;
      perm_entries_dest.innerHTML =
         xmlResponse.transformNode(xslPerm_Entries);
      ace_entries_dest.innerHTML = "<div/>";
      ace_edit_dest.innerHTML = "<div/>";
      add_dest.innerHTML = "<DIV/>";


      xmlResponse.transformNodeToObject(xslAceEntries.documentElement,
         xmlAceEntries);

      HiLitePermsTable();
      propfindForResources();

      cmdAdd.disabled = false;
      cmdRemove.disabled = true;
      cmdAceApply.disabled = true;
      cmdAceCancel.disabled = true;

      strOwner = owner.innerText;
   }
}
```

To add a new Access Control Entry, all the application does is take your entered data and turn it into a new XML node that it adds to the XML security descriptor. To do this, the application fills in the required properties to correctly generate a new ACE such as an NT4-compatible name, the access mask, as well as the ACE type. The code that does the XML work is shown here:

```
function addAce(strUserName, strMask, strPropName, strApplyTo, _
   strRoleProp, strRoleScope, fbAllow)
{
   //   Basic variable declarations.
   var baseNode      = null;
   var nodeFirstAce  = null;
   var nodeSubNode     = null;
   var nodeNewNode     = null;
   var nOrder        = 0;

   nOrder = getAceOrder(strUserName, null, strPropName, strApplyTo,
      strRoleProp, strRoleScope, fbAllow);

   while(nOrder > -1)
   {
      removeAce(nOrder);
      nOrder = getAceOrder(strUserName, strMask, strPropName, strApplyTo,
         strRoleProp, strRoleScope, fbAllow);
   }

   //   Establish the base node.
   baseNode = xmlAceEntries.documentElement;

   //   Create the ace node.
   nodeNewNode = xmlAceEntries.createNode(1, "ace", "");

   //   Add the NT4_Compatible_Name sub node.
   nodeSubNode = xmlAceEntries.createNode(1, "NT4_Compatible_Name", "");
   nodeSubNode.text = strUserName;
   nodeNewNode.appendChild(nodeSubNode);

   //   Add the Access_Mask sub node.
   nodeSubNode = xmlAceEntries.createNode(1, "Access_Mask", "");
   nodeSubNode.text = strMask;
   nodeNewNode.appendChild(nodeSubNode);

   //   Add the Ace_Type sub node.
   nodeSubNode = xmlAceEntries.createNode(1, "Ace_Type", "");

   nodeSubNode.text = getAceType(strPropName, fbAllow);
```

(continued)

```
    nodeNewNode.appendChild(nodeSubNode);

    //   Add the Apply_To sub node.
    nodeSubNode = xmlAceEntries.createNode(1, "Apply_To", "");
    nodeSubNode.text = strApplyTo;
    nodeNewNode.appendChild(nodeSubNode);

    //   Add the Property_Name sub node.
    nodeSubNode = xmlAceEntries.createNode(1, "Property_Name", "");
    nodeSubNode.text = strPropName;
    nodeNewNode.appendChild(nodeSubNode);

    if(strRoleScope != null && strRoleScope != "")
    {
       nodeSubNode = xmlAceEntries.createNode(1, "Role_Scope", "");
       nodeSubNode.text = strRoleScope;
       nodeNewNode.appendChild(nodeSubNode);
    }

    if(strRoleProp != null && strRoleProp != "")
    {
       nodeSubNode = xmlAceEntries.createNode(1, "Role_Property", "");
       nodeSubNode.text = strRoleProp;
       nodeNewNode.appendChild(nodeSubNode);
    }

    baseNode = xmlAceEntries.documentElement;
    baseNode.appendChild(nodeNewNode);

    refreshAceList();
}
```

To set the security, the application does a PROPPATCH to the same property but with the new security descriptor formatted as XML and the new security added as XML nodes.

```
function cmdAceApply.onclick()
{
    var xmldomAce = new ActiveXObject("Microsoft.XMLDOM");
    var xmlAccessMaskNode = null;
    var xmlAceTypeNodes = null;
    var xmlAceTypeNode = null;
    var xmlUserNameNode = null;
    var xmlApplyToNode = null;
    var xmlPropNode = null;

    cmdAceApply.disabled = true;
    cmdAceCancel.disabled = true;

    addAcesOutstandingAcesToXML();
```

```
    xmlAceEntries.transformNodeToObject(xslProppatch.documentElement,
       xmldomProppatch);

    DAVRequest.open("PROPPATCH", txtUrl.value, false);
    DAVRequest.setRequestHeader("Content-Type", "text/xml");
    DAVRequest.setRequestHeader("Translate", "f");
    DAVRequest.send(xmldomProppatch);

    if(chkMultiStatusForErr(DAVRequest) >= 0)
    {
       HiLitePermsTable();

       if(nCurrentAce > -1)
       {
          xmldomAce.loadXML(
             xmlAceEntries.childNodes(0).childNodes(nCurrentAce).xml);
          ace_entries_dest.innerHTML =
             xmldomAce.transformNode(xslDips_ACE_Info);
          ace_edit_dest.innerHTML =
             xmldomAce.transformNode(xslACE_Edit.documentElement);

          perm_entries_dest.innerHTML =
             xmldomProppatch.transformNode(xslPerm_Entries);
          HiLiteRow(thetable, nCurrentAce+1);
       }
    }

    cmdGetSec.onclick();

    return "";
}
```

INSTANT MESSAGING

The final topic that we'll cover in the book is instant messaging. Instant messaging provides two sets of services. The first service is the ability to send and receive instant messages. Instant messages are different from e-mail in that they pop up on the receiving user's screen immediately after the message is sent—there is no delay. They also differ from e-mail because a user can actually block another user from sending the message. The only way you can "block" people from sending you e-mail is by automatically deleting the message.

The second and perhaps the most interesting service is the ability to track and display "presence" information about specified contacts. For example, a user could track its team, that is, determine whether team members are online, away from their computers, and so on. With presence technology, however, a user can track the availability and status of virtual teams across the Internet. This is where the power of presence technology shines.

Another great feature of instant messaging is that it is included with Microsoft Exchange 2000. When a user installs Exchange, the user has the option of installing instant messaging and setting up an instant messaging infrastructure. The instant messaging infrastructure leverages Active Directory and allows the user to extend the services he can offer and develop in the Exchange environment. Furthermore, the MSN Messenger Service client can be installed on workstations so that users in an organization can leverage instant messaging technology.

At this point, the main question you probably have is, "How does instant messaging relate to developers?" You can take advantage of these features in your collaborative Exchange applications by integrating instant messaging, presence technology, or both. A number of controls produced by Microsoft make the process of integrating instant messaging in your applications very easy. In this chapter, we will take a look at one of these controls, the IM Contact View control. This control provides a user interface to display presence information and send instant messages as well as a programmatic interface and event model that you can use in your applications.

Before we look at the details of the control, I need to show you the main page of the Training application, which has been updated with instant messaging functionality. As you can see in Figure 19-18, students can now see which instructors are online, and they can filter the list to view only those instructors for their courses. Students can send instant messages to their instructors. The instructors can block students from seeing their availability.

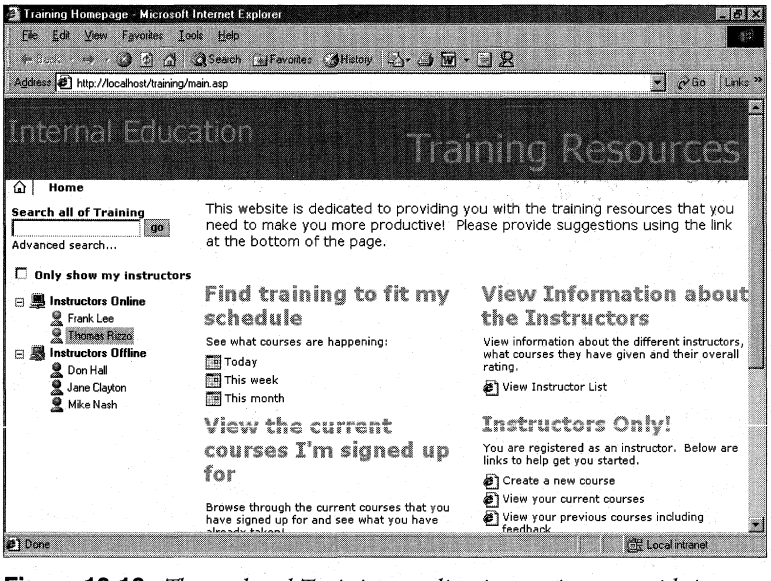


Figure 19-18. *The updated Training application main page with instant messaging technology added.*

Programming the IM Contact View Control

The IM Contact View control is an ActiveX control that you can place on your Web pages to implement IM functionality. This control has a number of properties and methods that you can use to customize the control's interface. You can set properties for the control by using an HTML Object tag, or you can set properties programmatically. The IM View Control is put onto the main.asp page in the application by using an OBJECT tag, as shown here:

```
<OBJECT
   codebase="msimcntl.cab#Version=1,0,0,001"
   classid=CLSID:B06EDBC7-287D-405C-A899-9C7F8358EF26
   codeType=application/x-oleobject
   id=MSIMContactList
   name=MSIMContactList VIEWASTEXT height=500>
   <PARAM NAME="Service" VALUE=0>
   <!--<PARAM NAME="List" VALUE="$$Messenger\Contact">-->
   <PARAM NAME="HotTracking" VALUE=FALSE>
   <PARAM NAME="AllowCollapse" VALUE=TRUE>
   <PARAM NAME="ShowSelectAlways" VALUE=TRUE>
   <PARAM NAME="ShowLogonButton" VALUE="1">

   <PARAM NAME="OfflineCollapsed" VALUE=TRUE>
   <PARAM NAME="OnlineCollapsed" VALUE=FALSE>
   <PARAM NAME="OnlineRootLabel" VALUE="Instructors Online">
   <PARAM NAME="OfflineRootLabel" VALUE="Instructors Offline">

   <PARAM NAME="FilterOffline" VALUE=FALSE>
   <PARAM NAME="Group" VALUE=True>

</OBJECT>
```

As you can see, the PARAM tags pass the name of a property and the value for that property. Table 19-15 describes the available properties on the IM Contact View control.

The control supports seven methods and one function, all of which are very straightforward. Table 19-16 describes them.

The final aspect of the IM Contact View control that you should be aware of is its event support. You can place the control on a Web page, so the control supports events that tell you when users of your application have instantiated the control, are adding new contacts, or are requesting menus, or even when the control is shutting down. Using these events, you can customize many aspects of the control. In the Training application, the menu events are used to add a custom menu item to the control so that right from the IM list, users can get details about instructors such as their office locations, phone numbers, and areas of expertise. This enhancement is shown in Figure 19-19 on page 962.

Property Name	*Description*
AllowCollapse	This Boolean property specifies whether the user is allowed to collapse the root nodes in the contact list. If set to *False*, the user cannot collapse the root nodes.
ExtentHeight	This read-only long value returns the optimal height the control needs to be in order to be displayed without a scrollbar. You can then use DHTML to readjust your Web page to properly fit the control.
ExtentWidth	This read-only long value returns the optimal width the control needs to be in order to be displayed without a scrollbar. You can use DHTML to readjust your Web page to properly fit the control.
FilterOffline	This Boolean value specifies whether offline contacts should be displayed in the list. *True* means that you want to filter offline contacts.
Group	This Boolean value specifies whether online and offline contacts should be grouped within their respective nodes in the tree. *True* means that you do want the contacts grouped. Note that you can change the text, as I did, for the tree nodes that the contacts are grouped under by using the *OfflineRootLabel* and *OnlineRootLabel* properties, which I'll discuss later.
HotTracking	This Boolean specifies whether the IM list will track the cursor as it moves over the list. *True* means that as the mouse cursor moves over contacts in the list, a hyperlink hand icon and an underline appear.
LoggedOn	A Boolean that indicates whether the user is logged on.
List	This string is the identifier for the collection of contacts to be used in the control. In the preceding sample code, the usage of this property is shown as commented out, but you could have the control automatically load the MSN Messenger contact list as its default. You can also create and identify your own lists in the code and load those as well. If you don't specify the value for this property, the control generates a random value for the property.

Table 19-15. *The IM Contact View control properties.*

Table 19-15 *continued*

Property Name	*Description*
OfflineCollapsed	This Boolean value specifies whether the offline contact list will be expanded or collapsed upon loading. *True* means that the offline contacts should be collapsed.
OfflineRootLabel	This string property allows you to customize the text that appears for the offline contact list. In the Training application, this property is set to *Instructors Offline*.
OnlineCollapsed	This Boolean is the same as the *OfflineCollapsed* property, except that it affects the online contact list.
OnlineRootLabel	This string property allows you to customize the text that appears for the online contact list. In the Training application, this property is set to *Instructors Online*.
SelectedMenuOptions	This read-only Long value specifies which menu options can be performed for a user. This property is not useful unless a contact is selected in the user interface when you query the property on the control.
Service	This property returns the *Service* interface. Providing more details about this interface is beyond the scope of this book.
ShowIcons	This Boolean property specifies whether the state icons should be shown. *True* means to show the state icons, such as the green person icon indicating available and the red person icon indicating offline.
ShowLogonButton	This Boolean specifies whether to show a logon prompt if the local user is not logged onto the IM service. *True* means that you want the logon prompt to be displayed.
ShowSelectAlways	This Boolean specifies whether to show the selection in the list even if the control does not have focus. *True* means that even if the user is typing into a form in a separate part of your application window, the current contact selection in the control will remain highlighted.

Method Name	*Description*
Add	Adds a list of contacts by using each contact's list ID. For MSN Messenger contacts, this would be $$Messenger\Contact. You can specify your own custom list here as well. You can also add contacts using a contact's IM address such as *username@imaddress.com*. Finally, you can add a number of contacts by passing a Variant array of strings that represent the IM addresses of all the contacts you want to add.
AddMenuItem	This function adds a menu item to the context menu that is displayed when contacts are right-clicked. (This method is covered in more detail in the programming samples at the end of this chapter.)
BlockSelected	Adds the selected contacts in the UI to the blocked list. These contacts will no longer be able to send instant messages to the user.
EmailSelected	Initiates an e-mail message to the selected contacts in the UI.
IMSelected	Sends an IM message to the selected contacts in the UI.
InviteSelected	Initiates an invitation to the selected contacts in the UI to join an application.
Remove	Removes a list or contact from the IM list. You need to pass the address of the IM user or name of the list as a parameter.
UnblockSelected	Removes the selected contacts in the UI from the blocked list.

Table 19-16. *IM Contact View Control methods and functions.*

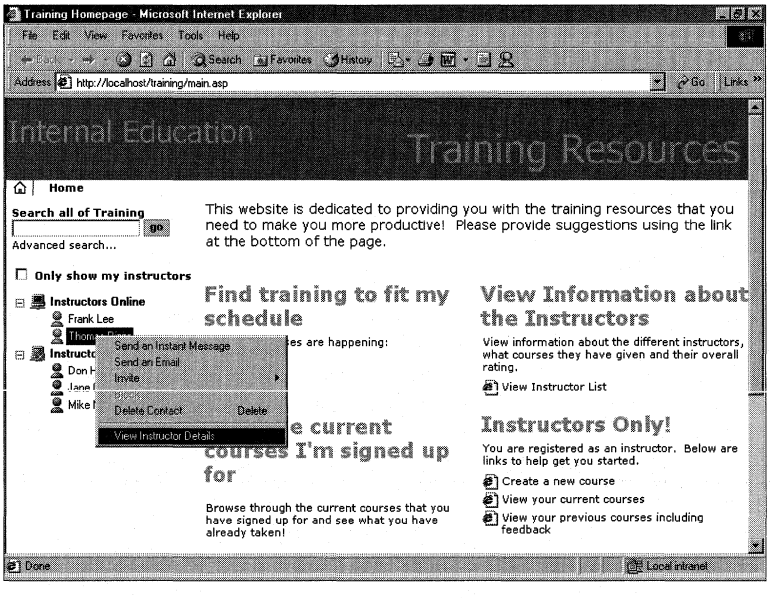


Figure 19-19. *Adding a custom menu option for your application to the IM Contact View control.*

Table 19-17 highlights the events you can receive from the IM Contact View control.

Event Name	*Description*
OnAddContactUI	This event is called when a user attempts to add a new contact using the Messenger Add Contact User Interface. The user can add a new contact by right-clicking on a node and selecting Add Contact. You are passed a Boolean variable *pfEnableDefault*, which, if you set it to *False*, cancels the user's action and prevents the user interface for adding a contact from appearing.
OnAddResult	This event is called when a contact or list has been added or an attempt has been made to add it. You are passed a Long value named *lResult*, which is the result of the attempt, as well as a Variant named *vUser*. This Variant holds either the list or contact that was added.
OnEmailContact	This event indicates that e-mail is being sent to the selected contacts. You are passed a Variant named *vUser*, which is the contact or contacts. You are also passed *pfEnableDefault*, which you can set to *False* to cancel the action.
OnExtentsChange	This event is called when the size of the list has changed. You are passed two Long values, *nWidth* and *nHeight*, which indicate the new width and height.
OnLocalStateChange	This event is called when the user changes the IM local state. This state can be online, offline, out to lunch, and so on. You are passed a variable *vState*, which is an Integer that identifies the new state.
OnLogOff	This event is called when the user logs off an IM service. You are passed a variable *lResult*, which is the result of the logoff attempt, and also a string *bstrService*, which is the service the user is logging off of, whether it be MSN or Exchange Instant Messaging.
OnLogon	This event is called when a user logs on to an IM service. You are passed a Long value, *lResult*, which is the result of the logon attempt, usually *0* for successful. You are also passed a string called *bstrService*, which is the service the user logged on to.
OnMenuRequest	This event indicates that a context menu is about to be displayed. We'll cover this event in more detail when we drill down into the Training application.
OnMenuSelect	This event indicates that a context menu item has been selected. We'll look at this event in more detail when we look at the Training application.
OnNewIMSession	This event is called when the user receives a request for an IM session from another user. You are passed a Variant, *vUser*, which is the user attempting to initiate the new session, as well as a Boolean, *pfEnableDefault*, which you can set to *False* to prevent the session from being initiated.

Table 19-17. *Events for the IM Contacts View control.* *(continued)*

Table 19-17 *continued*

Event Name	*Description*
OnReady	This event is called when the IM control is initialized and ready.
OnRemoveResult	This event is called when a contact or list is removed. You are passed *lResult*, which is the result of the removal, and *vUser*, which is the contact or list that was removed.
OnSelect	This event fires when a contact on the IM list is selected. The variable *vUser* is passed to indicate who was selected.
OnShutdown	This event fires when the control is shutting down.

One issue to note about the events is that a bunch of them are passed a *vUser* variable. This variable actually corresponds to an IMContact object. The IMContact object has some properties that are very useful, such as *FriendlyName*, *EmailAddress*, *LogonName*, and *State*. You can guess by their names what they mean: *FriendlyName* is the friendly name of the user in the control, *EmailAddress* is the e-mail address, *LogonName* is the logon name, and *State* returns a number that indicates the user's state. For State, the most common states and values are: *online(2)*, *busy(10)*, *idle(18)*, and *away(34)*.

Putting It All Together

Now that I've provided some information on the IM Contacts View control, we can look at the Training application implementation of the control. I'm going to highlight certain code sections of the implementation to kick-start your development efforts with the IM Contacts View control.

Getting Contacts into the List Programmatically

To figure out who the instructors are and get them into the list, the application uses ASP code on the server side to query the instructors folder and retrieve the entire list of instructors. The code then figures out which instructors are teaching classes that the current student is taking or has completed. Both sets of lists are stored in ASP session variables so that this query does not have to occur on every load of the page. The list is streamed out to the Web browser as two local VBScript arrays, one called *arrInstructorsIM* and the other called *arrMyInstructorsIM*. I stream them into an array so that I can pass the array to the *Add* method of the control and add all the instructors at once.

The addition is performed only if the user is logged onto an IM service and after the IM control has initialized. We can check both requirements programmatically by first checking the *LoggedOn* property for a *True* value. To make sure that I am attempting to add contacts only after the control has successfully initialized, the code listens for the OnReady event. Once both of these conditions have been satisfied, the code

programmatically adds either all instructors or only my instructors to the control, depending on what I check in the check box for the application. The code that performs these steps is shown here:

```
<SCRIPT LANGUAGE="vbscript">

<%

   'Check the session variables to see if the array of instructor contacts
   'is available
   'By using session variables, the IM contacts will refresh
   'only after a new session starts. If you want to refresh
   'every time the user hits the page, remove the session
   'variables.
   'There are two arrays, one for all instructors
   'and one for just this user's instructors.
   if IsEmpty(Session("arrInstructorsIM"))  then
      dim arrInstructorsIM()
      i=0
      'Query the store for the list of available instructors

      strQuery = "Select ""urn:schemas:contacts:email1"" FROM " & _
         "scope('shallow traversal of " & chr(34) _
         & strInstructorsFolderPath & chr(34) & "') WHERE " & _
         """DAV:ishidden"" = false" _
         & " AND ""DAV:iscollection"" = false"

      set oRS = Server.CreateObject("ADODB.Recordset")
      oRS.Open strQuery,Session("oConnection")
      Do Until oRS.EOF
         'Fill in the array with this info
         Redim Preserve arrInstructorsIM(i)
         arrInstructorsIM(i) = _
            oRS.Fields("urn:schemas:contacts:email1").Value
         i=i+1
         oRS.MoveNext
      Loop
      oRS.Close
      set oRS = Nothing

      on error resume next
      err.clear
      'Check UBound
      iThrowAway = UBound(arrInstructorsIM)
      if err.number = 0 then

%>
```

(continued)

```
            on error resume next
            dim arrInstructorsIM(<%=UBound(arrInstructorsIM)%>)

            <%
            for y=LBound(arrInstructorsIM) to UBound(arrInstructorsIM)
               response.write vblf & "arrInstructorsIM(" & y & ") = """ & _
                  arrInstructorsIM(y) & """"
            next
         else
            arrInstructorsIM=""
            %>
            arrInstructorsIM=""

         <%
         end if
         if err.number = 0 then
            'Save the local array into the Session variable
            Session("arrInstructorsIM") = arrInstructorsIM
         else
            Session("arrInstructorsIM") = ""
         end if

      else
         'Traverse the list of available instructors, and store them
         'in local VBScript vars

         arrInstructorsIM = Session("arrInstructorsIM")

         on error resume next
         err.clear
         'Check UBound
         iThrowAway = UBound(arrInstructorsIM)
         if err.number = 0 then

            %>

            on error resume next
            dim arrInstructorsIM(<%=UBound(arrInstructorsIM)%>)

            <%

            for y=LBound(arrInstructorsIM) to UBound(arrInstructorsIM)
               response.write vblf & "arrInstructorsIM(" & y & ") = """ & _
                  arrInstructorsIM(y) & """"
            next
         else
            %>
            arrInstructorsIM = ""
```

```
   <%
   end if
end if

if IsEmpty(Session("arrMyInstructorsIM")) then
   dim arrMyInstructorsIM()
   'Query the store for the list of my instructors.
   'Do this by querying the store for the courses
   'the user is registered for, and then pull off the instructor
   'e-mail. This shows all courses, both past and present.
   'You could also limit this to current courses only.
   strUserEmail = Session("UserEmail")


   strQuery = "Select """ & strSchema & "registrations""," & _
      """DAV:href"" FROM scope('shallow traversal of " & chr(34) _
      & strStudentsFolderPath & chr(34) & "') " & _
      "WHERE ""DAV:ishidden"" = false" _
      & " AND ""DAV:iscollection"" = false AND " _ &
      ""urn:schemas:contacts:email1""" _
      & "='" & strUserEmail & "'"

   set oRS = Server.CreateObject("ADODB.Recordset")
   oRS.Open strQuery,Session("oConnection")

   if oRS.RecordCount <> 0 then
      'Actually has a student record
      strRegistrations = oRS.Fields(strSchema & "registrations")


      'on error resume next
      if Not(IsNull(strRegistrations)) then
         arrRegistrations = Split(strRegistrations,",")
         strSQL = "Select """ & strSchema & "instructoremail"", " & _
            """urn:schemas:mailheader:subject""," _ &
            """DAV:href"",""urn:schemas:calendar:dtstart""," & _
            """urn:schemas:calendar:dtend"" " & _
            "FROM scope('shallow traversal of """ & _
            strScheduleFolderPath & _
            """') WHERE (""DAV:iscollection"" = false) AND " & _
            (""DAV:ishidden"" = false)"

         set rst = Server.CreateObject("ADODB.Recordset")
         With rst
            'Open recordset based on the SQL string
            .Open strSQL, Session("oConnection")
         End With
```

(continued)

```
      y=-1
      If Not(rst.BOF And rst.EOF) Then
         rst.MoveFirst
         Do Until rst.EOF
            for i = LBound(arrRegistrations) to UBound(arrRegistrations)
               'Print out the course
               if UCASE(rst.fields("DAV:href").Value) = _
                  UCASE((strScheduleFolderPath & _
                  arrRegistrations(i))) then
                  'Found a course the user is registered for
                  'Add it to an array
                  y=y+1
                  Redim Preserve arrMyInstructorsIM(y)
                  arrMyInstructorsIM(y) = _
                     rst.Fields(strSchema & "instructoremail").Value
               end if
            next
            rst.MoveNext
         loop
         rst.Close
         set rst=Nothing
      end if
   end if

   on error resume next
   err.clear
   'Check UBound
   iThrowAway = UBound(arrMyInstructorsIM)
   if err.number = 0 then

      %>
      on error resume next
      dim arrMyInstructorsIM(<%=UBound(arrMyInstructorsIM)%>)

      <%
      for y=LBound(arrMyInstructorsIM) to UBound(arrMyInstructorsIM)
         response.write vblf & "arrMyInstructorsIM(" & y & ") = " & _
         """" & arrMyInstructorsIM(y) & """"
      next
   else
   %>

   <%
   end if
   if err.number = 0 then
      'Save the local array into the Session variable
      Session("arrMyInstructorsIM") = arrMyInstructorsIM
```

```
         else
            Session("arrMyInstructorsIM") = ""
         end if

      end if
   else
      'Traverse the list of my instructors, and store them
      'in local VBScript vars
      arrMyInstructorsIM = Session("arrMyInstructorsIM")

      on error resume next
      err.clear
      'Check UBound
      iThrowAway = UBound(arrMyInstructorsIM)
      if err.number = 0 then

      %>
         dim arrMyInstructorsIM(<%=UBound(arrMyInstructorsIM)%>)


      <%
         for y=LBound(arrMyInstructorsIM) to UBound(arrMyInstructorsIM)
            response.write vblf & "arrMyInstructorsIM(" & y & ") = """ & _
               arrMyInstructorsIM(y) & """"
         next
      else
      %>
         arrMyInstructorsIM = ""
      <%
      end if
   end if
%>

SUB MSIMContactList_OnReady()
   'Only display if available
   if MSIMContactList.LoggedOn = True then
      document.all.IMSection.style.display = ""
   end if
   settimeout "AddIMContacts",1000,"vbscript"

end sub

SUB MSIMContactList_OnLogOn(BYVAL lResult, BYVAL bstrService)
   document.all.IMSection.style.display = ""
   settimeout "AddIMContacts",1000,"vbscript"
End SUB
```

(continued)

```
SUB MSIMContactList_OnLogOff()
   document.all.IMSection.style.display = "none"
End SUB

SUB EvaluateCheckbox()
   if document.all.myinstructors.checked = True then
      AddMyIMContacts
   else
      AddIMContacts
   end if
End Sub

SUB AddIMContacts()
   'Adds all instructors
   'Check to see if the user is logged on
   if MSIMContactList.LoggedOn = True then
      err.clear
      'Clear the list
      Randomize
      lRandomNumber = Int((30000 * Rnd)+1)
      MSIMContactList.List = CStr(lRandomNumber)
      if IsArray(arrInstructorsIM) then
         AddSelectedContact(arrInstructorsIM)
      end if
   end if
end sub


SUB AddMyIMContacts()
   'Adds only my instructors
   on error resume next
   err.clear
   if MSIMContactList.LoggedOn = True then
      'Clear the list
      Randomize
      lRandomNumber = Int((30000 * Rnd)+1)
      MSIMContactList.List = CStr(lRandomNumber)
      if IsArray(arrMyInstructorsIM) then
         AddSelectedContact(arrMyInstructorsIM)
      end if
   end if
end sub


sub AddSelectedContact(vUserAddress)
   MSIMContactList.Add vUserAddress
   if err.number <> 0 then
      msgbox "Add Contact: An error occured: " + err.number + ": " +
```

```
            err.description
        err.clear
    end if
end sub
```

Adding Dynamic Menu Items

The other task the Training application performs is dynamically adding a menu item to the context menu of the control. This menu item, as we saw, allows users to display instructor details as a menu option. The way to implement this functionality is to leverage the OnMenuRequest and OnMenuSelect events.

The OnMenuRequest event is fired when a user requests a context menu from the control. This event passes you a variable *vSelected*, which is the currently selected user in the control for which the menu was requested. Using this variable, you can decide whether you want the menu to appear. The instantiation of the menu is controlled by the next variable *pfDefaults*, which you can set to *False* if you do not want the menu to appear.

Inside of your event handler for the OnMenuRequest event, you can use the *AddMenuItem* function to add custom menu items. The *AddMenuItem* function takes a string that specifies the name of the menu for the new item. If you send an empty string, the control creates a new divider. This function also takes a Long value, which is the index of the location in the menu where you want the new item to appear. If you do not specify this location, the new menu item appears at the bottom of the menu list. Finally, this function will return to you a Long value that is a unique identifier for your menu item. You should retain this value so that you can use it with the next event we're going to cover, OnMenuSelect.

The OnMenuSelect event fires when a user selects a menu item from a context menu. This event passes you a *vSelected* variable, which is the user object of the currently selected user. This method also passes you a Long value, *lCmd*, which identifies which menu item was selected. If you saved the value returned from *AddMenuItem*, you can compare the identifier passed by OnMenuSelect with the custom menu identifier you saved. If they match, you should perform the custom action required by your application. In my case, I get the friendly name of the user. I could have retrieved the e-mail address of the user if I had used the *EmailAddress* property on the *vSelected* variable that is passed by the OnMenuSelect event. I then launch a new instance of IE to an ASP page I created and pass the friendly name of the selected contact to that ASP page. The ASP page finds and displays the details for the selected contact. All the code for this scenario is shown here:

```
SUB MSIMContactList_OnMenuRequest(BYVAL vSelected, BYREF pfDefaults)
   throwaway = MSIMContactList.AddMenuItem("")
   DetailsCmd = MSIMContactList.AddMenuItem("View Instructor Details")
End Sub
```

(continued)

```
Sub MSIMContactList_OnMenuSelect(BYVAL vSelected, BYVAL lCmd)
   If lcmd = DetailsCmd then
      on error resume next
      err.clear
      'Try to grab the friendly name; if an error, then
      'not a valid selection
      strFriendlyName = vSelected.FriendlyName
      'Could also be strEmail = vSelected.EmailAddress
      if err.number = 0 then
         'Open a new window to show instructor details
         window.open "iminstructordetail.asp?InstructorName=" & _
            vSelected.FriendlyName,null, _
            "location=false,menubar=false,toolbar=false"
      end if

   end if
End Sub
```

Index

Note: Page numbers in italics refer to figures or tables.

Special Characters and Numbers

A

B

C

D

E

F

G

H

I

J

K–L

M

N

O

P

Q

R

S

T

U

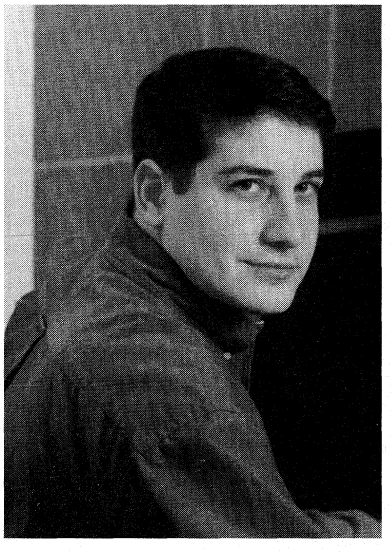
V

W

X

Y–Z

THOMAS RIZZO



Thomas Rizzo is a product manager in the Microsoft Exchange Server product group, where he focuses on helping developers learn about the Exchange and Microsoft Outlook development platform. Before working in the Exchange Server group, Tom worked as a systems engineer in Microsoft's Washington, D.C., office. There he helped the United States government develop and deploy Microsoft technologies. Tom holds a Bachelor's degree from Georgetown University in Washington, D.C.

The manuscript for this book was prepared using Microsoft Word 2000. Pages were composed by Microsoft Press using Adobe PageMaker 6.52 for Windows, with text in Garamond and display type in Helvetica Black. Composed pages were delivered to the printer as electronic prepress files.

Cover Graphic Designer

Girvin | Branding & Design

Cover Illustrator

Glenn Mitsui

Interior Graphic Artist

Rob Nance

Principal Compositor

Daniel Latimer

Principal Proofreader/Copy Editor

Roger LeBlanc

Indexer

Liz Cunningham

MICROSOFT LICENSE AGREEMENT

Book Companion CD

IMPORTANT—READ CAREFULLY: This Microsoft End-User License Agreement ("EULA") is a legal agreement between you (either an individual or an entity) and Microsoft Corporation for the Microsoft product identified above, which includes computer software and may include associated media, printed materials, and "on-line" or electronic documentation ("SOFTWARE PRODUCT"). Any component included within the SOFTWARE PRODUCT that is accompanied by a separate End-User License Agreement shall be governed by such agreement and not the terms set forth below. By installing, copying, or otherwise using the SOFTWARE PRODUCT, you agree to be bound by the terms of this EULA. If you do not agree to the terms of this EULA, you are not authorized to install, copy, or otherwise use the SOFTWARE PRODUCT; you may, however, return the SOFTWARE PRODUCT, along with all printed materials and other items that form a part of the Microsoft product that includes the SOFTWARE PRODUCT, to the place you obtained them for a full refund.

SOFTWARE PRODUCT LICENSE

The SOFTWARE PRODUCT is protected by United States copyright laws and international copyright treaties, as well as other intellectual property laws and treaties. The SOFTWARE PRODUCT is licensed, not sold.

1. **GRANT OF LICENSE.** This EULA grants you the following rights:

   a. **Software Product.** You may install and use one copy of the SOFTWARE PRODUCT on a single computer. The primary user of the computer on which the SOFTWARE PRODUCT is installed may make a second copy for his or her exclusive use on a portable computer.

   b. **Storage/Network Use.** You may also store or install a copy of the SOFTWARE PRODUCT on a storage device, such as a network server, used only to install or run the SOFTWARE PRODUCT on your other computers over an internal network; however, you must acquire and dedicate a license for each separate computer on which the SOFTWARE PRODUCT is installed or run from the storage device. A license for the SOFTWARE PRODUCT may not be shared or used concurrently on different computers.

   c. **License Pak.** If you have acquired this EULA in a Microsoft License Pak, you may make the number of additional copies of the computer software portion of the SOFTWARE PRODUCT authorized on the printed copy of this EULA, and you may use each copy in the manner specified above. You are also entitled to make a corresponding number of secondary copies for portable computer use as specified above.

   d. **Sample Code.** Solely with respect to portions, if any, of the SOFTWARE PRODUCT that are identified within the SOFTWARE PRODUCT as sample code (the "SAMPLE CODE"):

      i. **Use and Modification.** Microsoft grants you the right to use and modify the source code version of the SAMPLE CODE, *provided* you comply with subsection (d)(iii) below. You may not distribute the SAMPLE CODE, or any modified version of the SAMPLE CODE, in source code form.

      ii. **Redistributable Files.** Provided you comply with subsection (d)(iii) below, Microsoft grants you a nonexclusive, royalty-free right to reproduce and distribute the object code version of the SAMPLE CODE and of any modified SAMPLE CODE, other than SAMPLE CODE (or any modified version thereof) designated as not redistributable in the Readme file that forms a part of the SOFTWARE PRODUCT (the "Non-Redistributable Sample Code"). All SAMPLE CODE other than the Non-Redistributable Sample Code is collectively referred to as the "REDISTRIBUTABLES."

      iii. **Redistribution Requirements.** If you redistribute the REDISTRIBUTABLES, you agree to: (i) distribute the REDISTRIBUTABLES in object code form only in conjunction with and as a part of your software application product; (ii) not use Microsoft's name, logo, or trademarks to market your software application product; (iii) include a valid copyright notice on your software application product; (iv) indemnify, hold harmless, and defend Microsoft from and against any claims or lawsuits, including attorney's fees, that arise or result from the use or distribution of your software application product; and (v) not permit further distribution of the REDISTRIBUTABLES by your end user. Contact Microsoft for the applicable royalties due and other licensing terms for all other uses and/or distribution of the REDISTRIBUTABLES.

2. **DESCRIPTION OF OTHER RIGHTS AND LIMITATIONS.**

- **Limitations on Reverse Engineering, Decompilation, and Disassembly.** You may not reverse engineer, decompile, or disassemble the SOFTWARE PRODUCT, except and only to the extent that such activity is expressly permitted by applicable law notwithstanding this limitation.
- **Separation of Components.** The SOFTWARE PRODUCT is licensed as a single product. Its component parts may not be separated for use on more than one computer.
- **Rental.** You may not rent, lease, or lend the SOFTWARE PRODUCT.
- **Support Services.** Microsoft may, but is not obligated to, provide you with support services related to the SOFTWARE PRODUCT ("Support Services"). Use of Support Services is governed by the Microsoft policies and programs described in the user manual, in "on-line" documentation, and/or in other Microsoft-provided materials. Any supplemental software code provided to you as part of the Support Services shall be considered part of the SOFTWARE PRODUCT and subject to the terms and conditions of this EULA. With respect to technical information you provide to Microsoft as part of the Support Services, Microsoft may use such information for its business purposes, including for product support and development. Microsoft will not utilize such technical information in a form that personally identifies you.

- **Software Transfer.** You may permanently transfer all of your rights under this EULA, provided you retain no copies, you transfer all of the SOFTWARE PRODUCT (including all component parts, the media and printed materials, any upgrades, this EULA, and, if applicable, the Certificate of Authenticity), **and** the recipient agrees to the terms of this EULA.
- **Termination.** Without prejudice to any other rights, Microsoft may terminate this EULA if you fail to comply with the terms and conditions of this EULA. In such event, you must destroy all copies of the SOFTWARE PRODUCT and all of its component parts.

3. **COPYRIGHT.** All title and copyrights in and to the SOFTWARE PRODUCT (including but not limited to any images, photographs, animations, video, audio, music, text, SAMPLE CODE, REDISTRIBUTABLES, and "applets" incorporated into the SOFTWARE PRODUCT) and any copies of the SOFTWARE PRODUCT are owned by Microsoft or its suppliers. The SOFTWARE PRODUCT is protected by copyright laws and international treaty provisions. Therefore, you must treat the SOFTWARE PRODUCT like any other copyrighted material **except** that you may install the SOFTWARE PRODUCT on a single computer provided you keep the original solely for backup or archival purposes. You may not copy the printed materials accompanying the SOFTWARE PRODUCT.
4. **U.S. GOVERNMENT RESTRICTED RIGHTS.** The SOFTWARE PRODUCT and documentation are provided with RESTRICTED RIGHTS. Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of the Commercial Computer Software—Restricted Rights at 48 CFR 52.227-19, as applicable. Manufacturer is Microsoft Corporation/One Microsoft Way/Redmond, WA 98052-6399.
5. **EXPORT RESTRICTIONS.** You agree that you will not export or re-export the SOFTWARE PRODUCT, any part thereof, or any process or service that is the direct product of the SOFTWARE PRODUCT (the foregoing collectively referred to as the "Restricted Components"), to any country, person, entity, or end user subject to U.S. export restrictions. You specifically agree not to export or re-export any of the Restricted Components (i) to any country to which the U.S. has embargoed or restricted the export of goods or services, which currently include, but are not necessarily limited to, Cuba, Iran, Iraq, Libya, North Korea, Sudan, and Syria, or to any national of any such country, wherever located, who intends to transmit or transport the Restricted Components back to such country; (ii) to any end user who you know or have reason to know will utilize the Restricted Components in the design, development, or production of nuclear, chemical, or biological weapons; or (iii) to any end user who has been prohibited from participating in U.S. export transactions by any federal agency of the U.S. government. You warrant and represent that neither the BXA nor any other U.S. federal agency has suspended, revoked, or denied your export privileges.
6. **NOTE ON JAVA SUPPORT.** THE SOFTWARE PRODUCT MAY CONTAIN SUPPORT FOR PROGRAMS WRITTEN IN JAVA. JAVA TECHNOLOGY IS NOT FAULT TOLERANT AND IS NOT DESIGNED, MANUFACTURED, OR INTENDED FOR USE OR RESALE AS ON-LINE CONTROL EQUIPMENT IN HAZARDOUS ENVIRONMENTS REQUIRING FAIL-SAFE PERFORMANCE, SUCH AS IN THE OPERATION OF NUCLEAR FACILITIES, AIRCRAFT NAVIGATION OR COMMUNICATION SYSTEMS, AIR TRAFFIC CONTROL, DIRECT LIFE SUPPORT MACHINES, OR WEAPONS SYSTEMS, IN WHICH THE FAILURE OF JAVA TECHNOLOGY COULD LEAD DIRECTLY TO DEATH, PERSONAL INJURY, OR SEVERE PHYSICAL OR ENVIRONMENTAL DAMAGE. SUN MICROSYSTEMS, INC. HAS CONTRACTUALLY OBLIGATED MICROSOFT TO MAKE THIS DISCLAIMER.

DISCLAIMER OF WARRANTY

NO WARRANTIES OR CONDITIONS. MICROSOFT EXPRESSLY DISCLAIMS ANY WARRANTY OR CONDITION FOR THE SOFTWARE PRODUCT. THE SOFTWARE PRODUCT AND ANY RELATED DOCUMENTATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OR CONDITION OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT. THE ENTIRE RISK ARISING OUT OF USE OR PERFORMANCE OF THE SOFTWARE PRODUCT REMAINS WITH YOU.

LIMITATION OF LIABILITY. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL MICROSOFT OR ITS SUPPLIERS BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR ANY OTHER PECUNIARY LOSS) ARISING OUT OF THE USE OF OR INABILITY TO USE THE SOFTWARE PRODUCT OR THE PROVISION OF OR FAILURE TO PROVIDE SUPPORT SERVICES, EVEN IF MICROSOFT HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN ANY CASE, MICROSOFT'S ENTIRE LIABILITY UNDER ANY PROVISION OF THIS EULA SHALL BE LIMITED TO THE GREATER OF THE AMOUNT ACTUALLY PAID BY YOU FOR THE SOFTWARE PRODUCT OR US$5.00; PROVIDED, HOWEVER, IF YOU HAVE ENTERED INTO A MICROSOFT SUPPORT SERVICES AGREEMENT, MICROSOFT'S ENTIRE LIABILITY REGARDING SUPPORT SERVICES SHALL BE GOVERNED BY THE TERMS OF THAT AGREEMENT. BECAUSE SOME STATES AND JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY, THE ABOVE LIMITATION MAY NOT APPLY TO YOU.

MISCELLANEOUS

This EULA is governed by the laws of the State of Washington USA, except and only to the extent that applicable law mandates governing law of a different jurisdiction.

Should you have any questions concerning this EULA, or if you desire to contact Microsoft for any reason, please contact the Microsoft subsidiary serving your country, or write: Microsoft Sales Information Center/One Microsoft Way/Redmond, WA 98052-6399.

PN 097-0002297

Programming Microsoft Outlook and Microsoft Exchange

Second Edition

Build collaborative business solutions that bridge—and extend—enterprise resources.

Develop and run core business services across the enterprise using Microsoft's powerful messaging and collaboration tools—Outlook 2000, Exchange Server 5.5, and Exchange 2000 Server. This book delivers detailed guidance—plus the Digital Dashboard Starter Kit and a full cache of code on CD—to help you build rich, extensible solutions for tracking, messaging, workflow, knowledge management, and real-time collaboration.

- Understand key Outlook building blocks—including folders and forms—and begin constructing solutions using the Outlook Object Model and Microsoft Visual Basic®, Scripting Edition (VBScript)
- Create a digital dashboard that gives users single-click access to the company's knowledge sources
- Write custom scripts and agents and expand your development potential using Microsoft Exchange Event Service and scripting agents
- Learn how the Active Directory™ Services Interface (ADSI) can increase your control of Exchange Server
- Extend your application's reach to the Web by using Microsoft Internet Information Services, Site Server, Outlook HTML Form Converter, Outlook Web Access, and Active Server Pages
- Use Collaboration Data Objects (CDO) to write applications that interact with Exchange Server and also render content in HTML for the Web
- Expand your Outlook 2000–based applications by creating COM add-ins
- Start building applications for Exchange 2000—including working with new Web Store technology
- Begin preparing for MCSD Exam 70-105

ISBN 0-7356-1019-3

7 90145 10193 8

9 780735 610194 90000

Included on CD-ROM—
E-Book Plus Code Samples

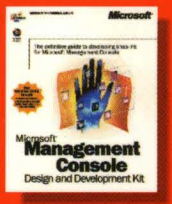
- Sample applications from the book—including the External Out-of-Office Messenger
- Exchange 2000 SDK
- Digital Dashboard Starter Kit
- Help files and utilities
- Supplementary chapters
- Fully searchable electronic version of the book

*For **system requirements**, please see the section "About the Companion CD."*

About the Author

Thomas Rizzo is an accomplished solutions developer who currently serves as product manager in the Microsoft Exchange group. In his five years at Microsoft, Tom has garnered two major awards—Outstanding Technical Contributor from Microsoft Technical Education and 1996 Federal Systems Engineer of the Year.

Also available:



The definitive guide to developing snap-ins for Microsoft Management Console.

Microsoft Management Console Design and Development Kit

ISBN: 0-7356-1038-X

To see the full line of developer resources from Microsoft Press, go to: **mspress.microsoft.com**

U.S.A.	**$49.99**	
U.K.	£32.99	[V.A.T. included]
Canada	$72.99	

[Recommended]

Programming/Microsoft Exchange/Microsoft Office

Microsoft®