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 DS
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
' stuff 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
End If  'LoadConfiguration
```

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 FolderID of the exchnews public folder.
    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 11-33. Compare it to the same details page presented as an Outlook message, shown in Figure 11-34 on the following page.

Figure 11-33
The details of the intranet news item include rich text, hyperlinks, and attachments.

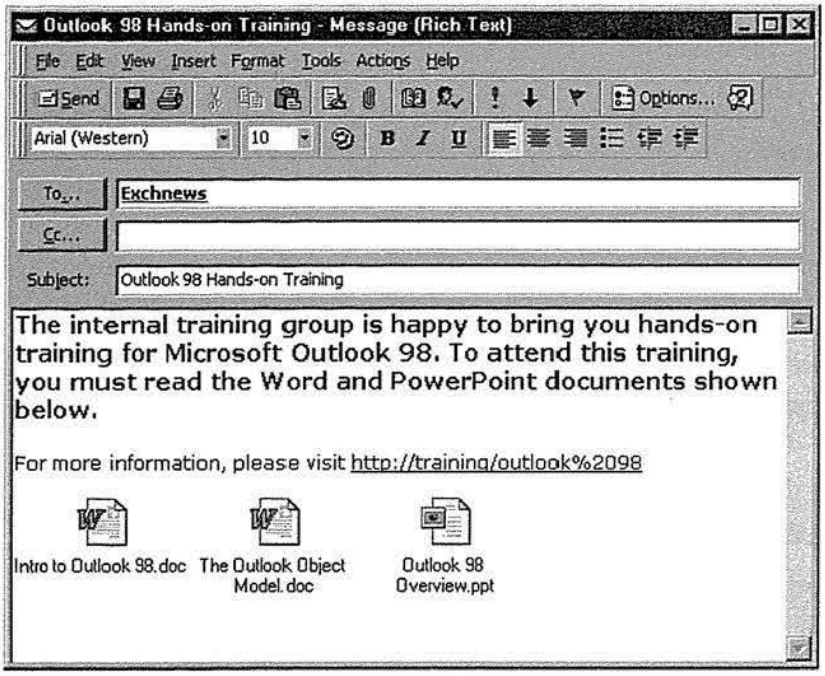


Figure 11-34
The item in Figure 11-33, shown as an Outlook message. Notice how the web and Outlook versions look almost identical. This is due to the CDO Rendering library's automatic conversion of rich text to HTML.

The Events Calendar 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 Events Calendar 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 docs
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)
```

(continued)

```
            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, it 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 Microsoft 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 11-35 shows the application in action.

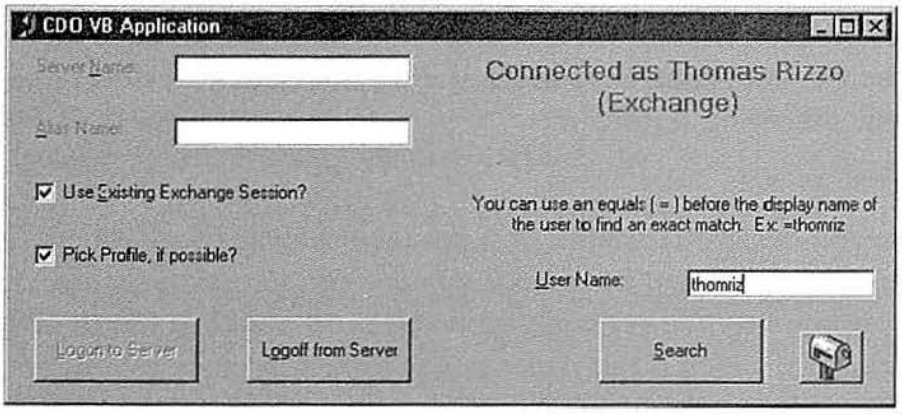


Figure 11-35
The Visual Basic CDO application.

Setting Up the Application

Before you can install the application, you must have a Windows NT 4.0 Server and a client with certain software installed. Table 11-5 describes the installation requirements.

Table 11-5
Installation Requirements for the CDO Visual Basic Application

Software Requirements	Installation Notes
Exchange Server 5.5 SP1	
CDO library (cdo.dll)	Exchange Server 5.5 SP1 installs CDO library 1.21. Outlook 98 installs CDO library 1.21.
For the client: Outlook 98	

To install the Visual Basic CDO 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 if 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
```

(continued)

```
            Else
                MsgBox "You need to enter a value in the " & _
                    "Server or Alias name.", _
                    vbOKOnly + vbExclamation, "CDO Logon"
                Exit Sub
            End If
        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 if 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 11-35.) 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. This value for this property is used in the status text, which indicates the connection state of the user, as shown in Figure 11-36.

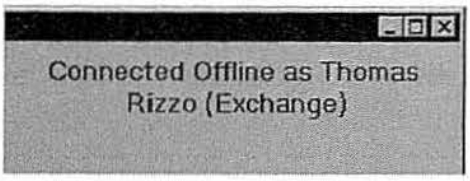


Figure 11-36
If the user is working offline, the connection status message displays this information.

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 very 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 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
```

(continued)

```
            MsgBox "Ambiguous entries found", vbOKOnly, "Search"
        Else
            Set oAddressEntry = oAddressEntries.GetFirst
            oAddressEntry.Details
        End If
    End If
End Sub
```

This code gets the Global Address List, 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 either be 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 equals (=) 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 11-37.

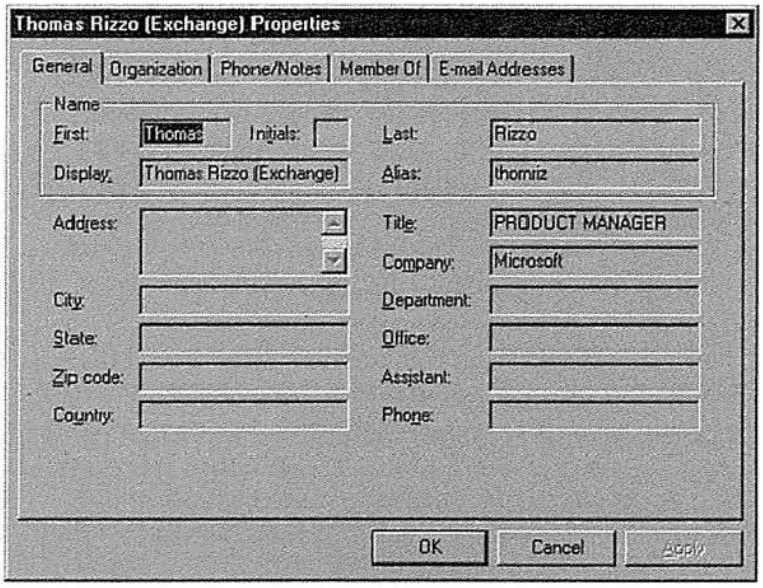


Figure 11-37

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 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 very 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 NT Challenge/Response or 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.

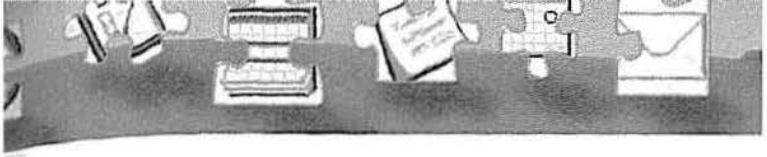
Learn Your Properties and Their IDs Well

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

sychronize 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 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 TWELVE

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 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 12-1 on the following page.

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.

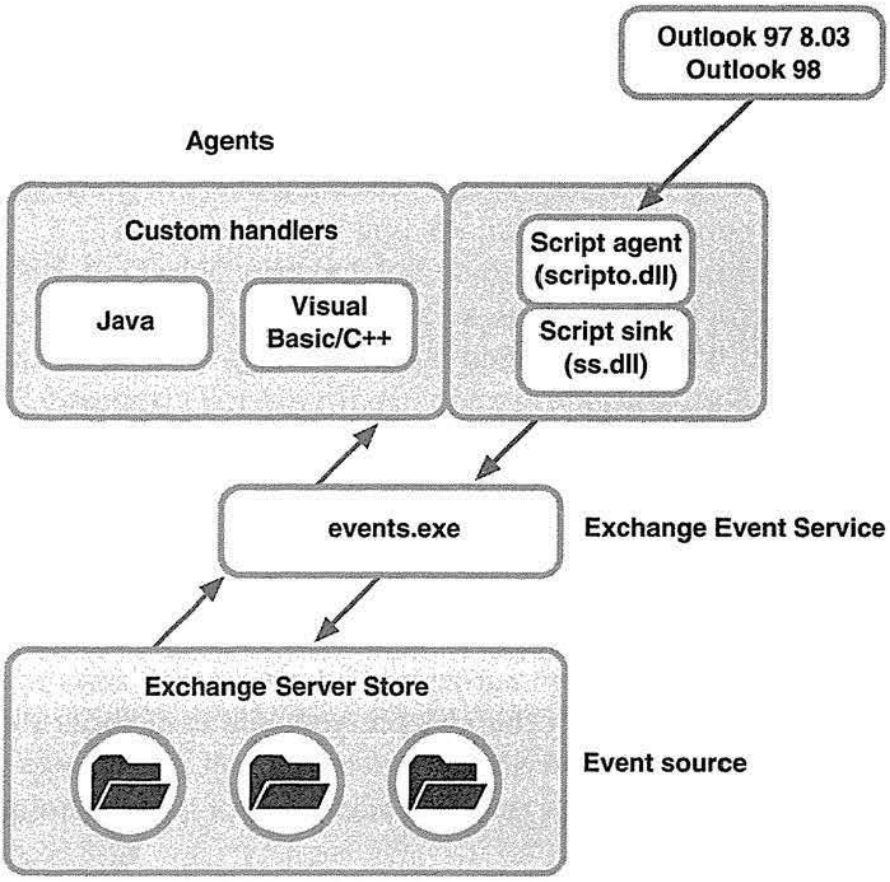


Figure 12-1
The Exchange Event Scripting Agent architecture.

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 timed 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 Collaborative Data Object (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. Currently, custom handlers can be authored using only C/C++ and 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 Microsoft 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. As well, 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 it. 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 12-2.

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 12-2
You change the Windows NT account that the Event Service runs under by using the Services 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 12-3.

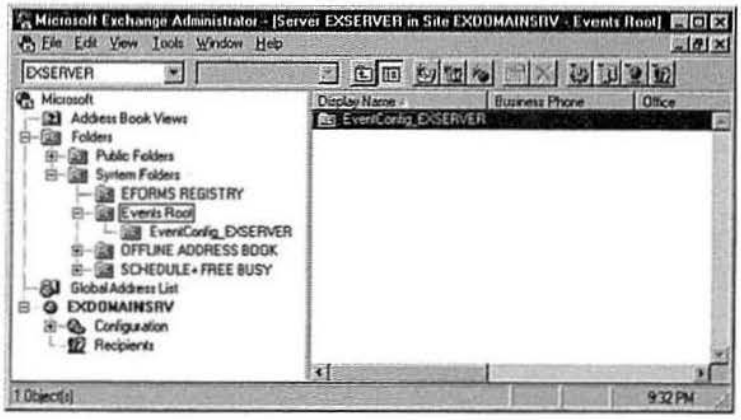


Figure 12-3
*The EventConfig_*servername *folder is found under the Events Root system folder. You must set user permissions for this folder if you want users to write agents.*

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 12-4, 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 12-5, 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 98 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.

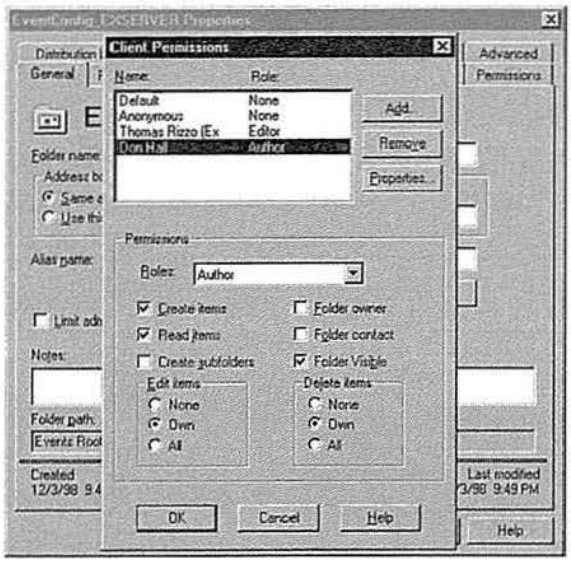


Figure 12-4

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.

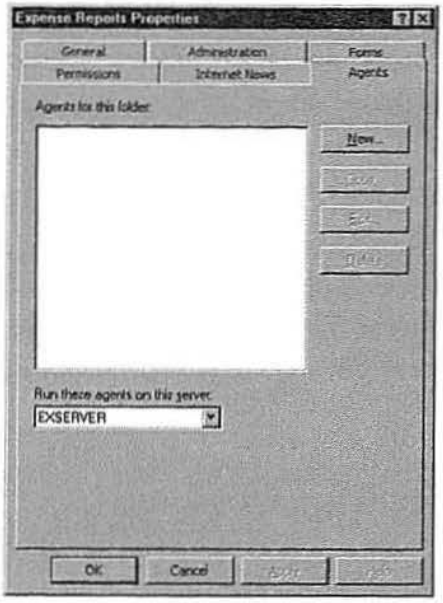


Figure 12-5
*The Agents tab for the Expense Reports public folder. You must have appropriate EventConfig_*servername *permissions, be an owner of the folder, and the Server Scripting add-in must be installed to see the Agents tab in Outlook.*

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 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 12-5. 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 do not 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 12-6.

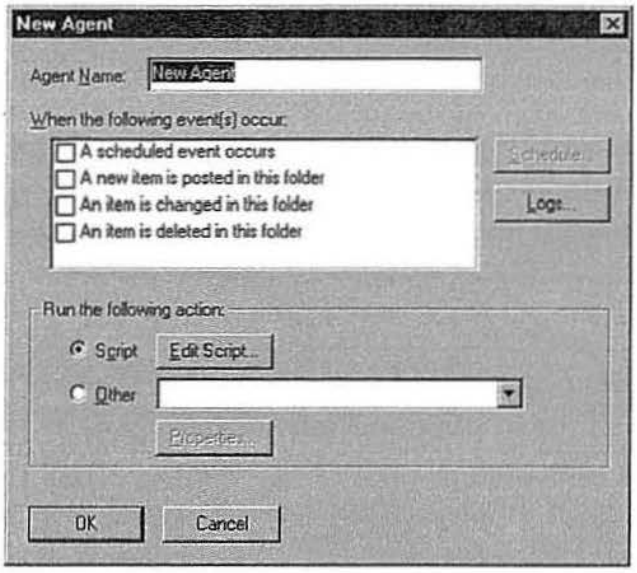


Figure 12-6
The New Agent dialog box. This dialog box allows you to pick the events that your agent will fire on.

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 12-7.

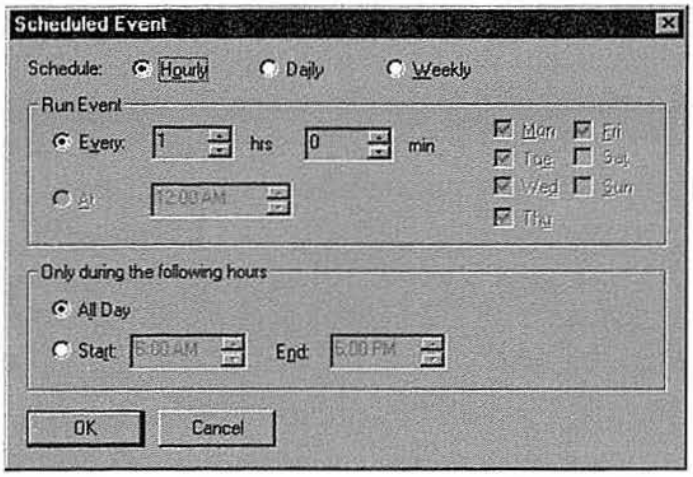


Figure 12-7
The Scheduled Event dialog box allows you to configure scheduled events for your agents.

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. 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 12-8.

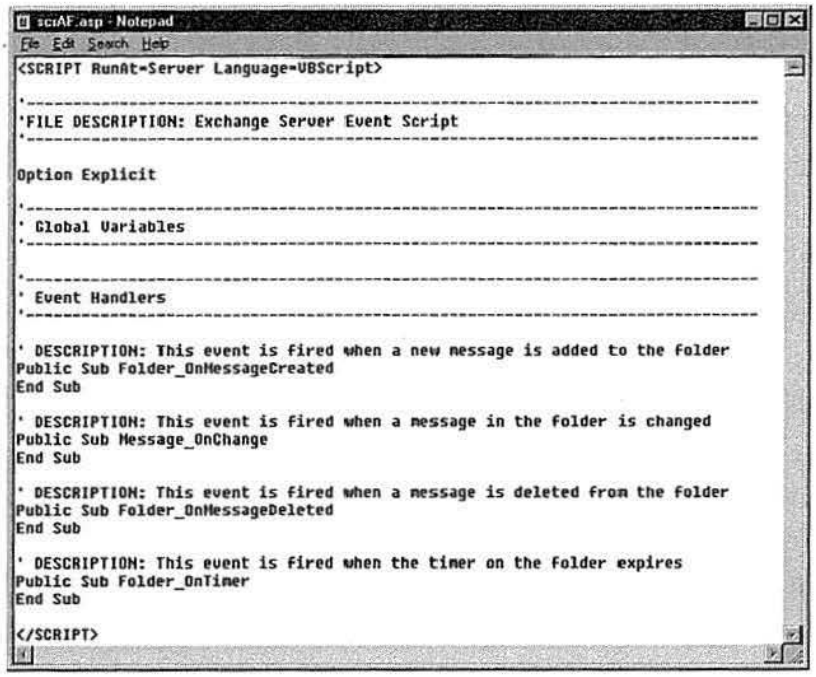


Figure 12-8
A new script shown in Notepad. Notice how the new agent automatically contains four event procedures to handle the four events supported by the Event Service.

Supported Event Types

As mentioned earlier, the Event Service supports four different event types: message create, change, delete, and timer-based. 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

Collaboration Data Objects (CDO), which you learned about in Chapter 11, represent 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.

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 6. 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 computer where the script is currently executing. Figure 12-9 shows the Microsoft Script Debugger debugging a script.

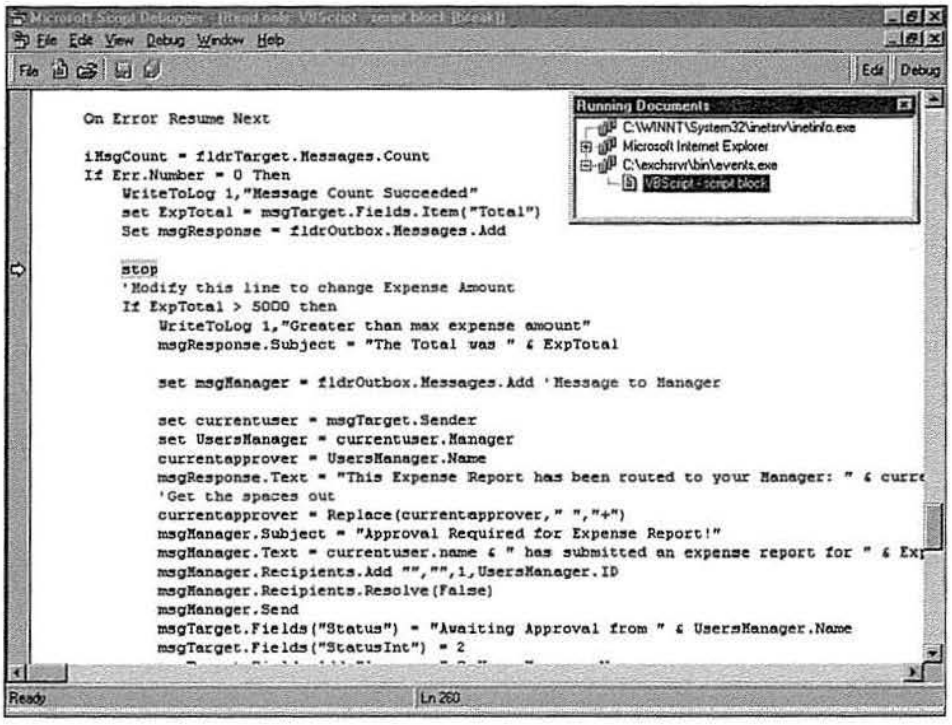


Figure 12-9
The Microsoft Script Debugger allows you to step through your scripts running on the Exchange Server.

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 12-10 on the next page shows an example of an agent log file.

You can access the log file for your agent via remote 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.

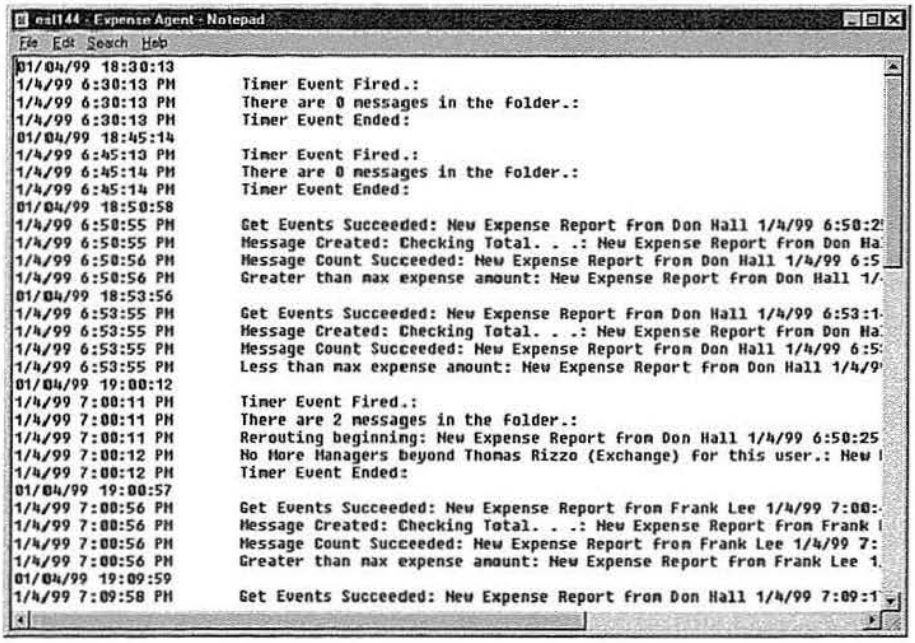


Figure 12-10
An agent log file in Notepad. Each agent has an associated log file in which you can write your own status or error-logging information.

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 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 Event Log, as shown in Figure 12-11. (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.

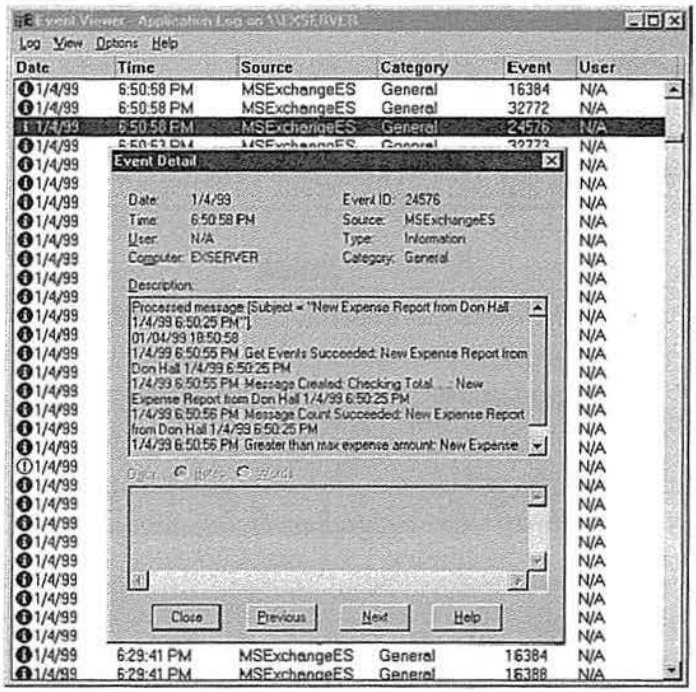


Figure 12-11
The event details for the Expense Report application. Since the Script.Response *method was used, the Description field has detailed information.*

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 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 12-1 describes the installation requirements for the application.

Table 12-1
Installation Requirements for the Expense Report Application

Required Software	Installation Notes
Exchange Server 5.5 SP1 with Outlook Web Access	
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 SP1 installs CDO library 1.21 and CDO Rendering library 1.21. Outlook 98 installs CDO library 1.21.
For the client: A web browser Outlook 98	For the web browser, Internet Explorer 4.0 is recommended. You can run the client software on the same machine or on a separate machine.

To install the Expense Report application, copy the Expense Report 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 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 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 12-12. Click OK twice to return to Outlook.

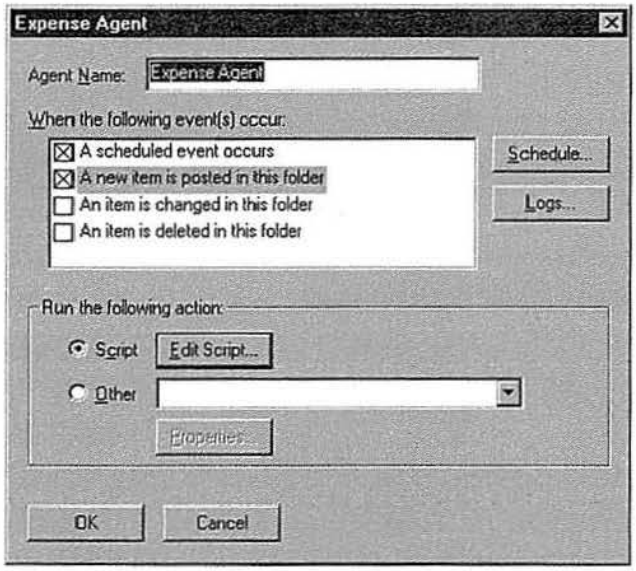


Figure 12-12
The configured Expense Agent dialog box.

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 12-13. 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 12-14. 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 an Excel document to implement the same functionality as the web page.

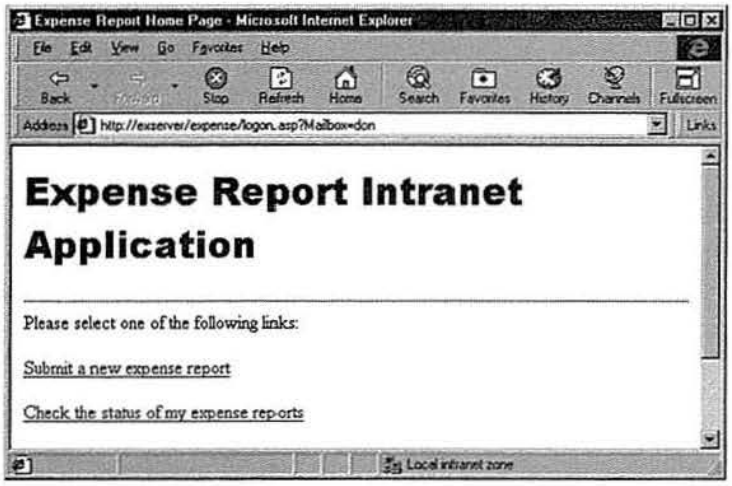


Figure 12-13
The main page of the Expense Report application.

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).

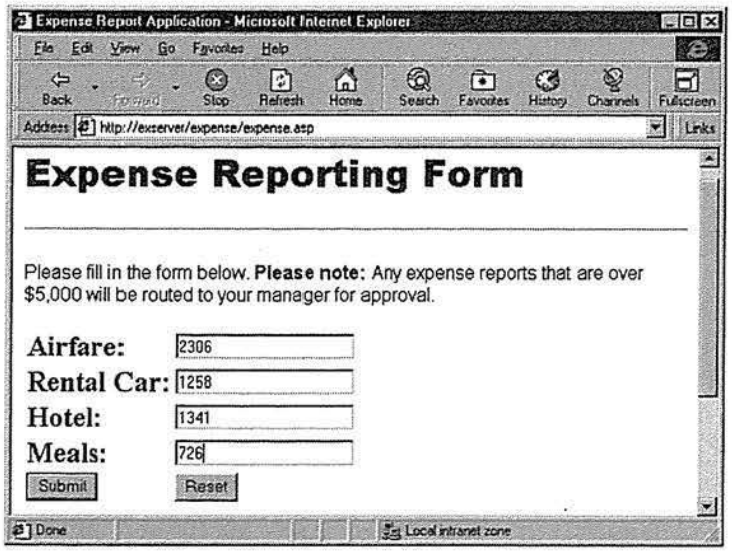


Figure 12-14
The page used to enter and submit expense reports.

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 12-15 on the next page. If the amount is under the limit, the agent automatically approves the expense report and routes it for payment.

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.

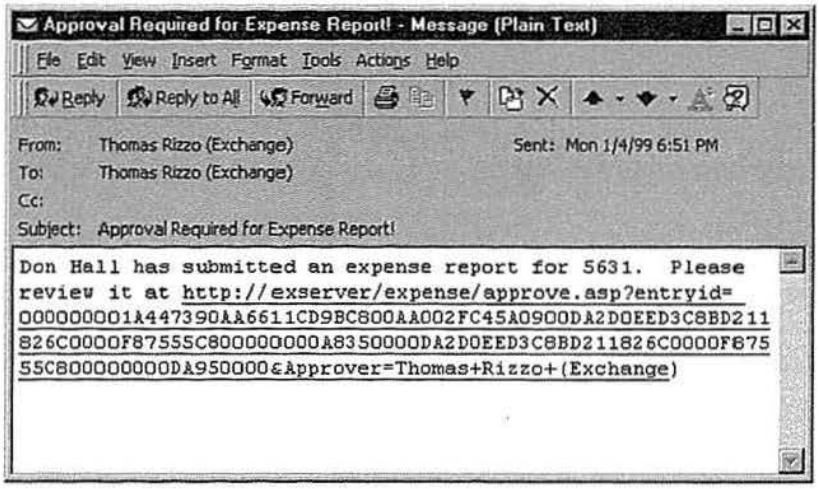


Figure 12-15
E-mail, with a link to an expense report sent to a manager by the Expense Agent, requesting approval of the expense.

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, a user can go to a web page to track the status of their expense reports. As you can see in Figure 12-16, 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.

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 12-17.

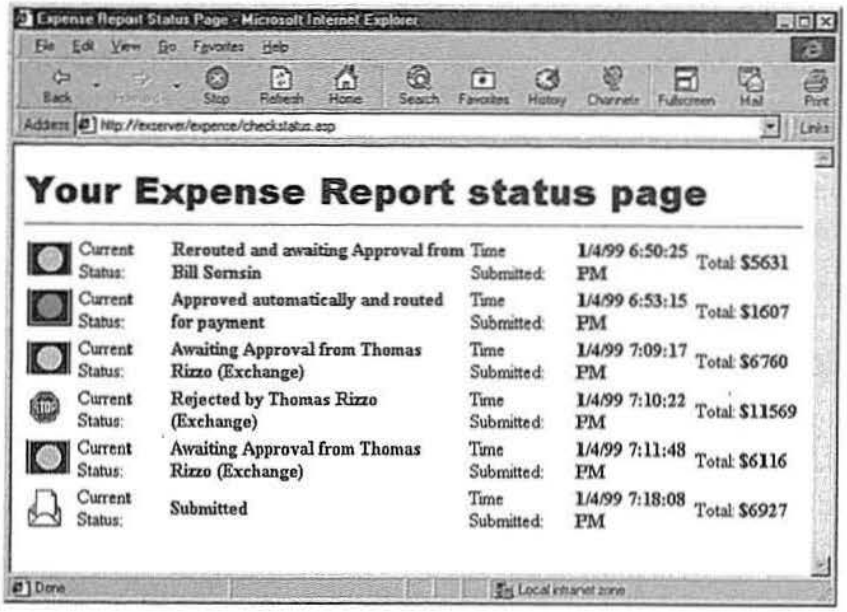


Figure 12-16

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.

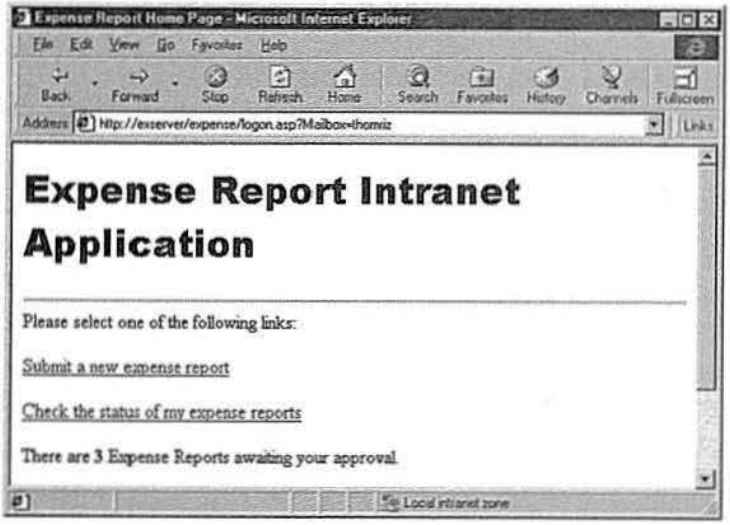


Figure 12-17

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 the CDO chapter.

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

    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 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. The code for the *WriteToLog* function is shown at the top of the next page.

```
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_OnMessageCreated* code is shown here:

```
' DESCRIPTION: This event is fired when a new message is added to
' the folder
Public Sub Folder_OnMessageCreated
    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. " & _
```

(continued)

```
                "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 15 minutes, 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-based 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
```

(continued)

```
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 over 15 minutes
            'Set another variable to the current message
            set msgTarget = oMessage
            WriteToLog 1,"Rerouting beginning"
            set ExpTotal = oMessage.Fields("Total")
            'Reroute the message
            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 info
            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, _
```

(continued)

```
                        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 11, which covers CDO development. The most interesting section of 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 only contain those items that meet your specified criteria:

```
<% 'Check to see if 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 Agents 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 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 12-18 shows the object hierarchy for the Exchange Event Service Configuration library.

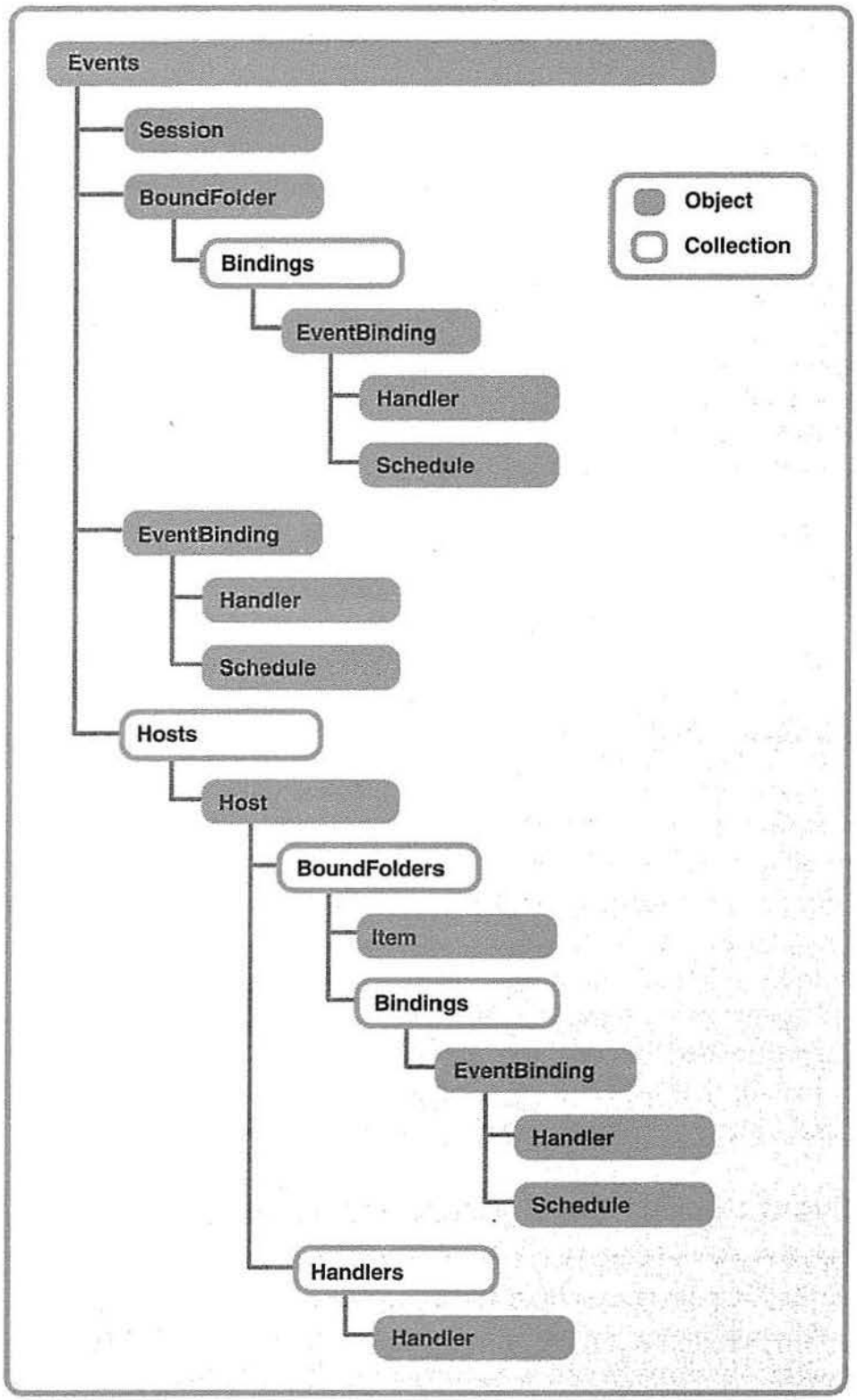


Figure 12-18
The object hierarchy for the Exchange Event Service Configuration library.

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 12-19. 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 the next chapter.) 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.

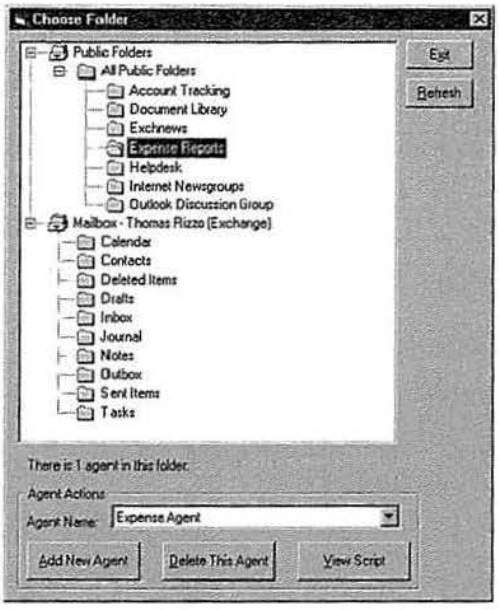


Figure 12-19
The main interface for the Agent Install application. The tree view allows you to pick a folder that you want to perform actions on.

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 12-20. 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-based events.

Figure 12-20
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 12-21, 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.

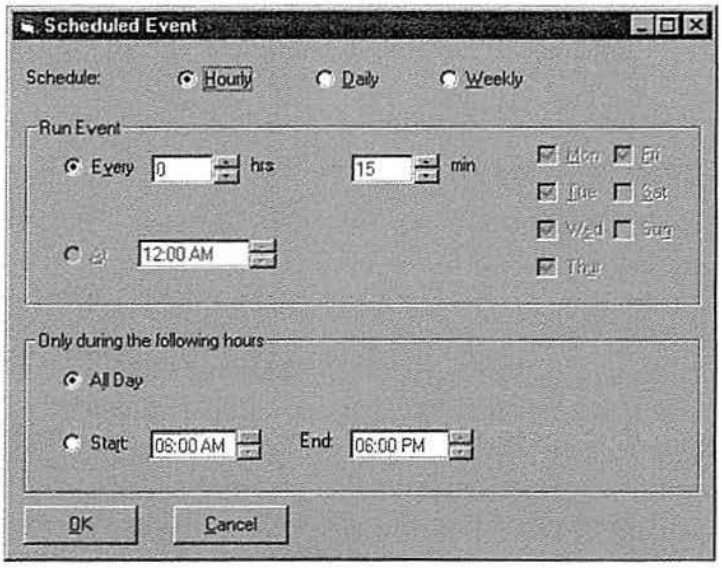


Figure 12-21
The Scheduled Event dialog box for the Agent Install program. This dialog box mimics the Scheduled Event dialog box in Outlook.

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 Agents Install application.

Accessing Existing Agents

As shown earlier in Figure 12-19, 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 temp file with a unique name
    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 temp 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
```

(continued)

```
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 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 timed 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
```

(continued)

```
    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
    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"
End If
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 glimps of how to disable and delete agents. To disable an agent, all you need to do i set the Active property on the Binding object to *0* and then call the *SaveChang* method. To delete an agent, find the Binding object that corresponds to th agent you want to delete, and then call the *Delete* method on the Binding collection and pass the Binding object to *Delete*. Call the *SaveChanges* metho to save the changes.

Agent Hosts

Although not used in the Agent Install application, you can enumerate th Exchange Server hosts capable of running agents. The Event Service Configu ration library offers a Hosts collection, which provides you with a Count prop erty for the number of available hosts and an Item property that will return specific Host object. The following code fragment shows how you can print ou 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 th HostName property on the BoundFolder object. Remember that all agents i a particular folder must run on the same host. You cannot have different agent 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 metho 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 fo 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 tha are separate from the home server where the folders generating the events ar located. This support allows you to isolate the agent server from your othe servers that host mailboxes or public folder applications. It is good practice t set up these agent servers so that errant scripts do not bog down your standar Exchange Servers. While occasionally logic errors might make your scripts en ter infinite loops or generate errors, the Event Service and agent technologie have built-in time-out capabilities that will terminate bad scripts after a speci fied amount of time.

Running the Script Engine in MTS

You can place the Event Scripting Agent (Scripto.dll) into MTS, which allow you to run the Scripting Agent using a specific Windows NT account for secu rity purposes and also to run the Scripting Agent in a dedicated and isolate process. MTS will manage instantiating the Scripting Agent as well as shutting it down if any anomalies occur during processing.

To make it easier for you to install the Event Scripting Agent as an MTS component, Microsoft Exchange Server 5.5 includes a prebuilt MTS package for you to use. To install the package, follow these steps:

1. Make sure you have MTS installed on the server where you are running the Event Service. As of the writing of this book, the latest version of MTS is version 2.0, and it ships with the Windows NT 4.0 Option Pack.
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 12-22.

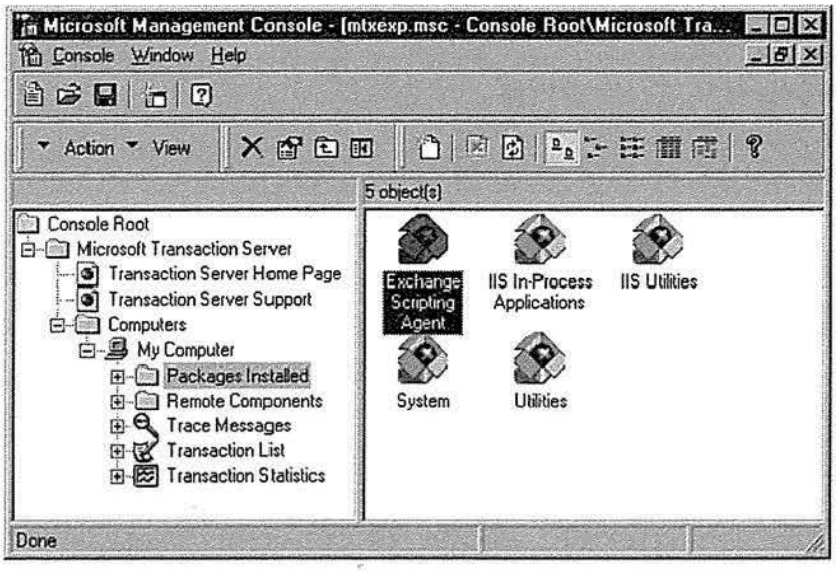
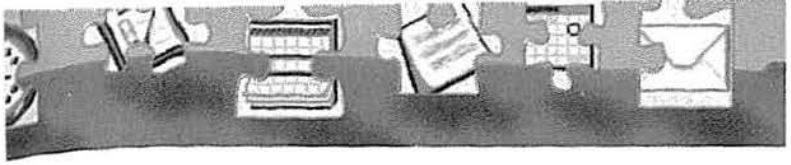


Figure 12-22
The Exchange Event Scripting Agent installed as an MTS component.

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.



CHAPTER THIRTEEN

Exchange Server Routing Objects

In the last 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 12 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 to 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.