realtimepublishers.com™

The Administrator Shortcut Guide™ To

VBScripting for Windows

Don Jones

Copyright Statement

Chapter 4: Advanced Scripting

Aside from the amazing things you can do with WMI and ADSI, scripting can provide a lot of additional functionality for making administration easier. For example, you can work with databases in a script, which gives you the ability to log WMI information into a SQL Server or Access database. In addition, the ability to run scripts on remote machines lets you extend your administrative reach and scope across your entire enterprise. In this chapter, I'll touch on these and other advanced topics, giving you a head start toward making your scripts more powerful and flexible.

Remote Scripting

We've already explored a form of remote scripting—running a script that affects remote computers from your computer. WMI and ADSI, in particular, are useful for this type of remote scripting. As Figure 4.1 illustrates, the script executes on one computer but performs operations against one or more remote computers. Typically, the script executes under the authority of the user account running the script. However, some technologies—including WMI—provide the means to specify alternative credentials, which the script can use when connecting to remote machines.

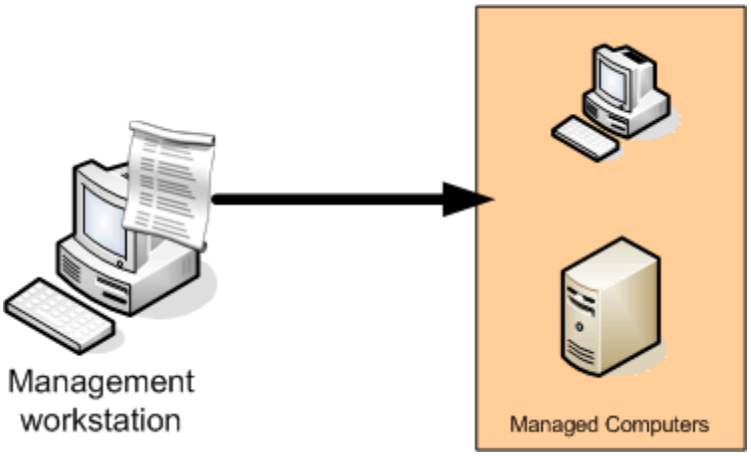


Figure 4.1: Basic remote scripting.

Another type of remote scripting is made possible by using the WshController object. As Figure 4.2 shows, this object actually copies a script from your machine to the remote machines, which execute the script independently. WshController allows you to monitor the remote script execution for errors and completion.

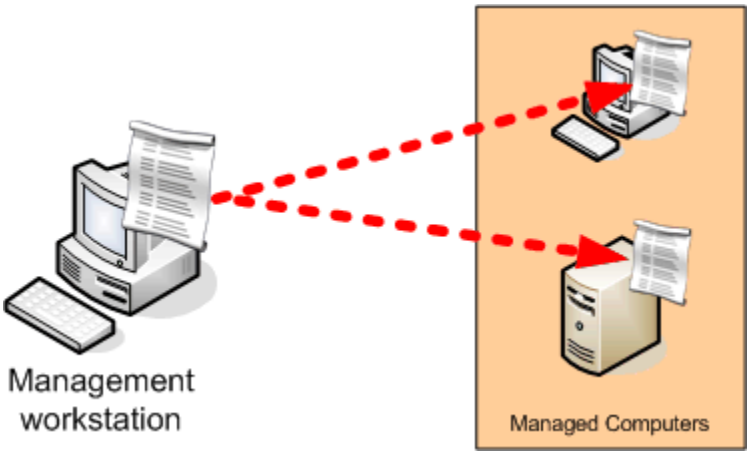


Figure 4.2: Using the WshController object for remote scripting.

The WshController Object

WshController is created just like any other object: by using the CreateObject() function. WshController has just one method, CreateScript(). This method returns a WshRemote object, which allows you to interact with a remote script. Suppose you have a script named C:\Script.vbs on your local machine that you want to run on a computer named ClientB. You would use a script similar to the following example:

```
Dim oController, oRemote
Set oController = WScript.CreateObject("WSHController")
Set oRemote = oController.CreateScript("c:\Script.vbs", _
 "ClientB")
oRemote.Execute
```

> Remote scripting is available only in the latest version of the Windows Script Host (WSH), version 5.6, and on NT 4.0 Service Pack 3 (SP3) and later versions of Windows. In general, you must be a local administrator on the remote machine, and remote WSH must be enabled in the registry of the remote machine. You can do so by navigating the registry to HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows Script Host\Settings, and adding a key named Remote as a REG_SZ (String) value. Set the value to 1 to enable remote WSH and 0 to disable it. It's disabled by default for security reasons. WSH 5.6 must be installed on both the machine sending the script and the machine that is to run it (the remote machine).

Not too difficult. Of course, with just that code, you won't be able to track the remote script's status. Add the following to provide tracking:

```
Do While oRemote.Status = 0
 WScript.Sleep 100
Loop
MsgBox "Remote execution completed with status " & oRemote.Status
```

The Status property can be either 0 or 1: 0 means the remote script is still running, and 1 means it has completed. The WshRemote object has two methods: Execute and Terminate. We've explore how the Execute method is used. The Terminate method can be used to end a still-running script, and this method requires no parameters.

WshRemote also provides a child object, WshRemoteError. This child object provides access to errors occurring in remote scripts. Using it is a bit more complicated and requires the use of the default WScript object's ConnectObject() method.

WScript.ConnectObject

You've already seen how the intrinsic WScript object's CreateObject() and GetObject() methods are used. The ConnectObject method is similar to GetObject() in that it deals with a remote object. Rather than retrieving a reference to a remote object, however, ConnectObject allows you to synchronize object events. As we've previously explored, objects have three basic members:

- Properties, which describe the object and modify its behavior
- Methods, which make the object perform some action
- Collections, which provide access to child objects

There is actually one other type of member: an event. *Events* provide an object with the means to inform your script when something occurs. For example, buttons in the Windows user interface (UI) fire an event when someone clicks them. This event being fired tells the underlying code that the button was clicked, allowing the underlying code to take whatever action it's supposed to take when the button is clicked. The following example demonstrates this concept:

```
Dim oController, oRemoteScript
Set oController = WScript.CreateObject("WSHController")
Set oRemoteScript = oController.CreateScript("me.vbs","Server1")

WScript.ConnectObject oRemoteScript, "remote_"
oRemoteScript.Execute

Do While oRemoteScript.Status <> 2
 WScript.Sleep 100
Loop

WScript.DisconnectObject oRemoteScript
```

This example script closely follows the previous example to create a remote script, execute it, and wait until it finishes. But this example adds the ConnectObject method to synchronize the remote script's events with this script's events. Any remote events will be fired back to this script. This script needs to contain a subroutine, or *sub*, prefixed with "remote_", because that is what the script told ConnectObject to look for when events occur. You could add the following:

```
Sub remote_Error
 Dim oError
 Set oError = oRemoteScript.Error
 WScript.Echo "Error #" & oError.Number
 WScript.Echo "At line " & oError.Line
 WScript.Echo oError.Description
 WScript.Quit
End Sub
```

The DisconnectObject method is used when the script is over to cancel the connection between the remote script and the script shown here.

Remote Scripting Limitations

Remote scripting does have some limitations. Remote scripts shouldn't use InputBox(), MsgBox(), or WScript.Echo to produce output because remote scripts aren't given an interactive desktop to work with. Any output from a remote script will need to be written to a text file on a file server or some other central location where you can retrieve it and look it over.

Remote scripts also have some security limitations. Generally speaking, they'll run under the context of the LocalSystem account, although that does vary between Windows versions and may change in service packs for Windows XP and later versions to a less-powerful account. Also, because scripts run under that context, they may have difficultly accessing anything in the local profile of a user. For example, accessing registry keys in HKEY_CURRENT_USER won't necessarily connect to the currently logged on user (because the script isn't running under that user's context), which can create unexpected results for your scripts. If you absolutely need a script to run as the logged on user, assign the script as a logon script.

Database Scripting

Scripting is completely compatible with Microsoft's universal data access technology, called ActiveX Data Objects (ADO). As the name implies, ADO utilizes objects, so your scripts will use CreateObject() to instantiate these objects and assign them to variables.

Making Data Connections

ADO uses a Connection object to provide a connection to data sources, such as Access databases, SQL Server databases, Excel spreadsheets, and more. Creating the Connection object is simple:

```
Set oConn = CreateObject("ADODB.Connection")
```

You have a couple of options, however, for specifying the data to which you want to connect. The easiest—although, as I'll explain, not quite the most efficient—is to use an Open Database Connectivity (ODBC) Data Source Name (DSN). Windows XP and later computers provide an ODBC Control Panel in the Administrative Tools group; other versions of Windows provide this option from the Control Panel instead. Figure 4.3 shows the ODBC application and a list of configured DSNs for the current user (*System* DSNs are available for all users of the computer).

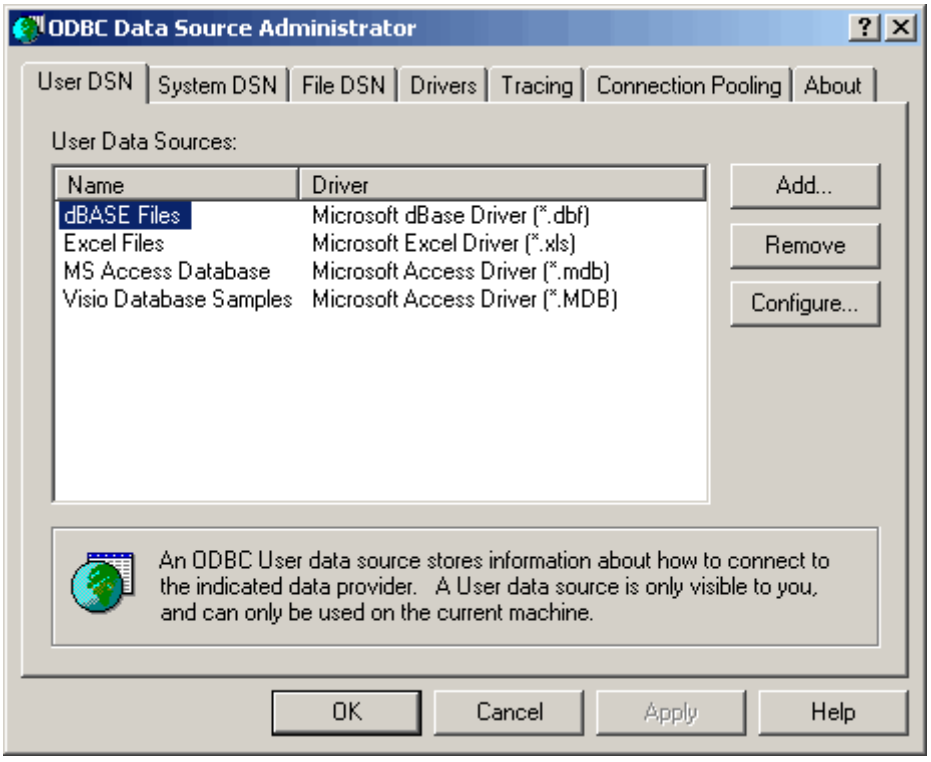


Figure 4.3: The ODBC panel.

For example, to add a DSN for an Access database, click Add, and select the Access driver from the list. Specify a name for your data source (such as "MyData"), and click Select to specify an Access MDB database. Once the DSN is created, you add code to open that DSN in your script:

```
oConn.Open "MyData"
```

There are a couple of downsides to using DSNs:

- They must be individually configured on each computer on which the script will run
- They access data through the older ODBC software in Windows—although this access method won't create a performance problem for most scripts, it isn't the most efficient way; you're effectively asking ADO to access data by using a technology that ADO itself (actually, the underlying OLE DB technology) was supposed to supersede

An alternative method is to use a native call and tell ADO which driver to use, what data to open, and so forth, all using a *connection string*. Sample drivers (or *providers,* in ADO terminology) include:

- Microsoft.Jet.OLEDB.4.0 (Access)
- SQLOLEDB (SQL Server)
- ADSDSOObject (Active Directory—AD)

Connection strings look a bit different depending on the driver used, because each driver requires slightly different information, for example:

- For an Access database: "Provider=Microsoft.Jet.OLEDB.4.0; Data Source=*folder\file.mdb*"
- For a SQL Server database: "Provider=SQLOLEDB; Data Source=*server*; Initial Catalog=*database*; User ID=*user*; Password=*password*"
- For AD: "Provider=ADSDSOObject; User ID=*user*; Password=*password*"

Thus, using this method, you would open the connection as follows for an Access database:

```
oConn.Open "Provider=Microsoft.Jet.OLEDB.4.0; " & _
 "Data Source=c:\databases\mydatabase.mdb"
```

This part of the process is the most difficult part of working with ADO. Once the connection is open, you can simply start querying and modifying data.

A Quick Database Lesson

Data in a database—whether it's an Excel spreadsheet or a SQL Server database—is organized into several logical components. A *table* is the main logical unit. Databases can consist of multiple tables. In an Excel file, each worksheet is treated as an individual table.

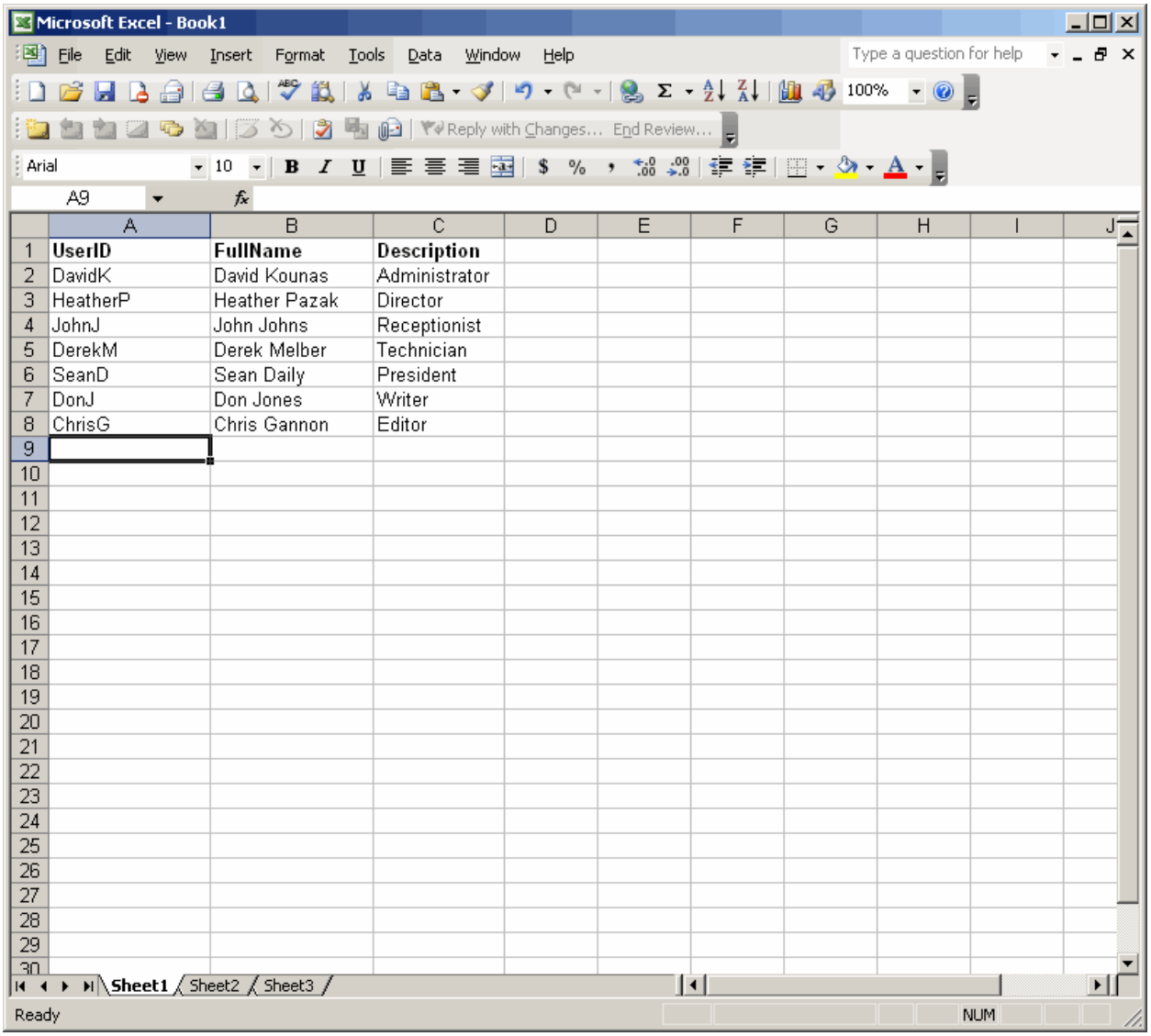
A *row* or *entity* or *record* contains a single entry in a table (for example, rows in an Excel spreadsheet or rows in an Access database table). A *column* or *domain* or *field* represents a single piece of information. For example, a column in an Excel spreadsheet might contain user names, and another column contains domain names for those users. ADO works with rows and columns to provide you with access to the data in a database.

Querying and Displaying Data

You'll probably find it easier to query data by using SQL-style queries, even when you're not accessing a SQL Server database; ADO understands the SQL query language and makes it pretty easy to use SQL with any type of data source. The basic syntax for a query looks like this:

```
SELECT column, column, column
FROM table
WHERE comparison
```

For example, suppose you have an Excel spreadsheet like the one that Figure 4.4 shows.



	A	B	C
1	**UserID**	**FullName**	**Description**
2	DavidK	David Kounas	Administrator
3	HeatherP	Heather Pazak	Director
4	JohnJ	John Johns	Receptionist
5	DerekM	Derek Melber	Technician
6	SeanD	Sean Daily	President
7	DonJ	Don Jones	Writer
8	ChrisG	Chris Gannon	Editor

Figure 4.4: Example Excel spreadsheet.

The table name in this case is Sheet1$ (Excel adds a dollar sign to the end of the worksheet name). There are three columns: UserID, FullName, and Description.

> ☞ In general, it's easiest to have table and column names that don't contain spaces or punctuation. However, if they do, you need to surround them in square brackets: [User ID], for example.

Suppose you wanted to query the UserID for every row in the table:

```
SELECT UserID FROM [Sheet1$]
```

If you only wanted the UserID of users whose description was "Writer," you would use this:

```
SELECT UserID FROM [Sheet1$] WHERE Description = 'Writer'
```

The results of your query are a set of rows—or as database people like to say, a set of records. In ADO parlance, that is a *recordset*, and it's represented by a Recordset object. To implement your query in VBScript, assuming a Connection object named oConn had been created and opened:

```
Set oRS = oConn.Execute("SELECT UserID FROM [Sheet1$]")
```

That leaves you with a Recordset object containing the specified rows and columns; in this case, it would contain seven rows and one column (assuming we're querying the Excel spreadsheet I showed you).

You should first determine whether your recordset contains anything. Recordset objects contain an internal pointer, which points to the current record. The objects also provide methods for moving the pointer to the next record, and properties for telling you when you've moved the pointer to the beginning of the file (recordset) or the end of the file (recordset). Those two properties, BOF (for the beginning of the file) and EOF (for the end of the file), will both be True for an empty recordset. So you can use the following comparison:

```
If oRS.EOF and oRS.BOF Then
 'no records
Else
 'records
End If
```

You can access the data in the recordset by simply referring to the column names. For example, the following will display the User ID for the current record:

```
WScript.Echo oRS("UserID")
```

Finally, you can move to the next record with this:

```
oRS.MoveNext
```

> ✎ In case you're wondering, there is a MovePrevious method. However, the default type of recordset returned by the Connection object's Execute() method is an efficient *forward-only* recordset, meaning that once you've used MoveNext to advance to the next record, you can't move back.
>
> Covering the vast complexity and flexibility of the entire set of ADO objects is a bit beyond the scope of this guide. However, you can check out the ADO documentation in the MSDN Library at http://www.microsoft.com/msdn; just look for the Data Access category.

Thus, writing a script that outputs all of the user IDs by using a DSN named "Excel," might look like this:

```
Dim oConn, oRS
Set oConn = CreateObject("ADODB.Connection")
oConn.Open "Excel"
Set oRS = oConn.Execute("SELECT UserID FROM [Sheet1$]")
If oRS.EOF and oRS.BOF Then
 WScript.Echo "No records returned"
Else
 Do Until oRS.EOF
  WScript.Echo "UserID: " & oRS("UserID")
  oRS.MoveNext
 Loop
End If

oRS.Close
oConn.Close
```

Notice that I threw in two lines of code to close the recordset and connection when the script ends. You don't strictly need to do so because VBScript will more or less do it automatically when the script ends, but it's a good practice.

As an extended example, the script that Listing 4.1 shows queries a DSN named "Excel" (which is assumed to be an Excel spreadsheet) and creates new users in an NT or AD domain. This script assumes that the Excel spreadsheet contains columns named UserID, FullName, and Description, much like the example I showed you earlier. The script creates passwords for the new users, and writes those passwords to a text file for distribution.

```
' PART 1: Open up the Excel spreadsheet
' using ActiveX Data Objects
Dim oCN
Set oCN = CreateObject("ADODB.Connection")
oCN.Open "Excel"

Dim oRS
Set oRS = oCN.Execute("SELECT * FROM [Sheet1$]")


' PART 2: Get a reference to the
' Windows NT domain using ADSI
Dim oDomain
Set oDomain = GetObject("WinNT://DOMAIN")


' PART 3: Open an output text file
' to store users' initial passwords
Dim oFSO, oTS
Set oFSO = CreateObject("Scripting.FileSystemObject")
```

```
Set oTS = oFSO.CreateTextFile("c:\passwords.txt",True)


' PART 4: For each record in the recordset,
' add the user, set the correct user
' properties, and add the user to the
' appropriate groups

' create the necessary variables
Dim sUserID, sFullName, sDescription
Dim sPassword, oUserAcct

' now go through the recordset one
' row at a time
Do Until oRS.EOF

  ' get the user information from this row
  sUserID = oRS("UserID")
  sFullName = oRS("FullName")
  sDescription = oRS("Description")

  ' make up a new password
  sPassword = Left(sUserID,2) & DatePart("n",Time) & _
   DatePart("y",Date) & DatePart("s",Time)

  ' create the user account
  Set oUserAcct = oDomain.Create("user",sUserID)

  ' set account properties
  oUserAcct.SetPassword sPassword
  oUserAcct.FullName = sFullName
  oUserAcct.Description = sDescription

  ' save the account
  oUserAcct.SetInfo

  ' write password to file
  oTS.Write sUserID & "," & sPassword & vbCrLf


  ' PART 5: All done!
  ' release the user account
  Set oUserAcct = Nothing

  ' move to the next row in the recordset
  oRS.MoveNext

Loop


' PART 6: Final clean up, close down
oRS.Close
oTS.Close
WScript.Echo "Passwords have been written to c:\passwords.txt."
```

Listing 4.1: An example script that queries a DSN named "Excel."

Note that you'll need to insert the correct domain name, which I've boldfaced, in order for the script to work. In an AD domain, users will be created in the default Users container.

Modifying Data

ADO isn't limited to pulling data from a database; it can modify and add information, too. There are a couple of ways to do so. The most straightforward, perhaps, is to issue a data modification query, using the Connection object's Execute method. This method is the same method that returns a Recordset object when used with a SELECT query; with a data modification query, it doesn't return anything. Here's how it works:

```
'Delete rows
oConn.Execute "DELETE FROM table WHERE criteria"

'Change rows
oConn.Execute "UPDATE table SET column=value WHERE criteria"

'Add rows
oConn.Execute "INSERT INTO table (column, column) " & _
 "VALUES ('value', 'value')"
```

For example, let's take the previous example, which creates user accounts and writes their passwords to a file. If the Excel spreadsheet had an additional column named Password, we could modify the script to save the passwords right into the spreadsheet instead of into a separate file. Listing 4.2 shows the lines of code that get removed in ~~strikethrough~~, and the changed lines in boldface.

```
' PART 1: Open up the Excel spreadsheet
' using ActiveX Data Objects
Dim oCN
Set oCN = CreateObject("ADODB.Connection")
oCN.Open "Excel"

Dim oRS
Set oRS = oCN.Execute("SELECT * FROM [Sheet1$]")


' PART 2: Get a reference to the
' Windows NT domain using ADSI
Dim oDomain
Set oDomain = GetObject("WinNT://DOMAIN")


~~' PART 3: Open an output text file~~
~~' to store users' initial passwords~~
~~Dim oFSO, oTS~~
~~Set oFSO = CreateObject("Scripting.FileSystemObject")~~
~~Set oTS = oFSO.CreateTextFile("c:\passwords.txt",True)~~


' PART 4: For each record in the recordset,
```

```
‘ add the user, set the correct user
‘ properties, and add the user to the
‘ appropriate groups

‘ create the necessary variables
Dim sUserID, sFullName, sDescription
Dim sPassword, oUserAcct

‘ now go through the recordset one
‘ row at a time
Do Until oRS.EOF

  ‘ get the user information from this row
  sUserID = oRS(“UserID”)
  sFullName = oRS(“FullName”)
  sDescription = oRS(“Description”)

  ‘ make up a new password
  sPassword = Left(sUserID,2) & DatePart(“n”,Time) & _
   DatePart(“y”,Date) & DatePart(“s”,Time)

  ‘ create the user account
  Set oUserAcct = oDomain.Create(“user”,sUserID)

  ‘ set account properties
  oUserAcct.SetPassword sPassword
  oUserAcct.FullName = sFullName
  oUserAcct.Description = sDescription

  ‘ save the account
  oUserAcct.SetInfo

  ‘ write password to database
  oCN.Execute “UPDATE [Sheet1$] SET Password = ‘“ & _
   sPassword & “‘ WHERE UserID = ‘“ & sUserID & “‘“


  ‘ PART 5: All done!
  ‘ release the user account
  Set oUserAcct = Nothing

  ‘ move to the next row in the recordset
  oRS.MoveNext

Loop


‘ PART 6: Final clean up, close down
oRS.Close
oTS.Close
WScript.Echo “Passwords have been written to the database.”
```

Listing 4.2: Example script that writes passwords to an existing Excel spreadsheet rather than a separate file.

There is another way to change data when you’re using a Recordset object: Simply change the columns similar to the way you read data from them. When you’re finished, use the Recordset’s Update method. Listing 4.3 shows the entire script again, modified to use this new method.

```
' PART 1: Open up the Excel spreadsheet
' using ActiveX Data Objects
Dim oCN
Set oCN = CreateObject("ADODB.Connection")
oCN.Open "Excel"

Dim oRS
Set oRS = oCN.Execute("SELECT * FROM [Sheet1$]")


' PART 2: Get a reference to the
' Windows NT domain using ADSI
Dim oDomain
Set oDomain = GetObject("WinNT://DOMAIN")


' PART 3: Open an output text file
' to store users' initial passwords
Dim oFSO, oTS
Set oFSO = CreateObject("Scripting.FileSystemObject")
Set oTS = oFSO.CreateTextFile("c:\passwords.txt",True)


' PART 4: For each record in the recordset,
' add the user, set the correct user
' properties, and add the user to the
' appropriate groups

' create the necessary variables
Dim sUserID, sFullName, sDescription
Dim sPassword, oUserAcct

' now go through the recordset one
' row at a time
Do Until oRS.EOF

  ' get the user information from this row
  sUserID = oRS("UserID")
  sFullName = oRS("FullName")
  sDescription = oRS("Description")

  ' make up a new password
  sPassword = Left(sUserID,2) & DatePart("n",Time) & _
   DatePart("y",Date) & DatePart("s",Time)

  ' create the user account
  Set oUserAcct = oDomain.Create("user",sUserID)

  ' set account properties
  oUserAcct.SetPassword sPassword
  oUserAcct.FullName = sFullName
  oUserAcct.Description = sDescription

  ' save the account
  oUserAcct.SetInfo

  ' write password to database
  oRS("Password") = sPassword
```

```
  oRS.Update

  ‘ PART 5: All done!
  ‘ release the user account
  Set oUserAcct = Nothing

  ‘ move to the next row in the recordset
  oRS.MoveNext

Loop


‘ PART 6: Final clean up, close down
oRS.Close
oTS.Close
WScript.Echo “Passwords have been written to the database.”
```

Listing 4.3: Example script that uses the Recordset object.

The Recordset object also supports an AddNew method for adding new rows, and a Delete method for deleting rows; check out the ADO documentation for details on using these methods in your scripts.

Windows Script Files

So far, all the scripts I’ve used in this guide are designed to be put into VBS files and run more or less as-is. WSH provides another file type, WSF (for Windows Script File), which is a more powerful and flexible format. WSF files are XML-formatted and provide better capabilities for creating scripts that accept command-line parameters. Here’s a brief example:

```
<package>
    <job id=“VBS”>
    <?job debug=“true”?>
        <script language=“VBScript”>
           WScript.Echo “This is VBScript”
        </script>
    </job>
    <job id=“JS”>
    <?job debug=“true”?>
        <script language=“JScript”>
           WScript.Echo(“This is JScript”);
        </script>
    </job>
</package>
```

Notice that the script contains several distinct elements:

- The entire script is contained in a <package>.
- Multiple <job> elements can exist, each with its own ID.
- Each <job> can contain a <script>, which can be in VBScript or JScript (or any other installed scripting language).
- The script itself is contained between the <script> tag and the </script> tag.

This example doesn't provide much beyond what a plain text file could do. The real fun of the WSF format comes with additional sections. Consider the example that Listing 4.4 shows.

```
<job>
    <runtime>
        <named
            name="server"
            helpstring="The server to run the script on"
            type="string"
            required="true"
        />

    <description>
    This script connects to a remote server and restarts it
    </description>

    <example>
    Example: Restart.wsf /server:servername
    </example>

    </runtime>
<script language="VBScript">
  'insert script here
</script>
</job>
```

Listing 4.4: Example WSF script.

This script defines a new <runtime> section, which contains several helpful sub-elements. The first is <named>. This element defines a named command-line argument. In this case, the name of the argument is "server," and it is intended to be a string value. It is required; if the script is executed without this argument, WSH won't allow the script to run. If this script were named "restart.wsf," you would execute it by running

```
restart.wsf /server:servername
```

from a command-line. The type can be "string," "Boolean," or "simple." In the case of "simple," the argument doesn't take a value.

Within your script, you can use the following code to display an automatically-generated "help file" for your script based on its arguments' "helptext" parameters and the <example> and <description> elements:

```
WScript.Arguments.ShowUsage
```

The <example> and <description> elements are just text and should be self-explanatory.

Users can also display the help text by running the script with the standard /? argument.

Within your script, you would access these arguments by using the WshArguments object. Using the above WSF file as an example, you might do something like the following in the main body of the script:

```
oArgs = WScript.Arguments.Named
sServerName = oArgs.Item("server")
```

The variable sServerName would now contain the value specified for the "server" argument. Because WSF files provide this great functionality for defining arguments, and because they can automatically produce a "help file" screen, it's a great format for using VBScript to create your own command-line utilities.

Signing Scripts

I recommend enabling WSH 5.6's script signature verification policies on all computers in your environment (read more about this feature on the Windows Scripting home page at http://www.microsoft.com/scripting and at http://www.ScriptingAnswers.com). This feature, when used properly, will prevent all unsigned scripts from executing, helping to prevent script-based viruses.

In general, you enable the feature by editing the registry. Navigate to HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows Script Host\Settings, and add a REG_DWORD value named "TrustPolicy." Set the value to 2 to fully enable signature verification.

There are also HKEY_LOCAL_MACHINE-related registry keys that affect verification policy, and you may need to modify these in Windows XP and WS2K3 machines in order to fully enable the verification policy. I provide an ADM template in the Downloads section of http://www.ScriptingAnswers.com; use this template to centrally configure and manage the signature verification policy settings via Group Policy objects (GPOs). Simply import the ADM template into a GPO and configure it as desired in the User and Computer configuration sections of the GPO.

However, in order to make sure *your* scripts run, you will need to sign them using a digital certificate issued for the purposes of code signing. You can purchase such a certificate from commercial certification authorities (CAs) such as VeriSign (http://www.verisign.com) and Equifax (http://www.equifax.com), or issue one from an internal CA, if your organization has one.

Your client and server computers must be configured to trust the publisher of the certificate you use. Consult the Windows documentation for information about importing a new trusted certificate publisher, if necessary; most commercial CAs are trusted by default.

Microsoft provides an object called Scripting.Signer that can take a certificate and sign a script. Note that signing a script marks the script with the signature, meaning you can't change the script without invalidating the signature (and preventing the script from running). If you need to modify the script, you will need to re-sign it.

The following code sample shows how to write a script that signs other scripts:

```
Set oSigner = CreateObject("Scripting.Signer")
sFile = InputBox("Path and filename of script to sign?")
sCert = InputBOx("Name of certificate to use?")
sStore = InputBox("Name of certificate store?")

oSigner.SignFile(sFile, sCert, sStore)
```

You'll simply need to know the name of your certificate and the certificate store in which the certificate is installed.

Summary

In this chapter, I've introduced you to some advanced VBScript topics, including script signing and security, flexible WSF files, remote scripting and script events, and ADO. Combined with what you've learned about VBScript's basics, WMI, and ADSI, you should be able to start producing some useful scripts on your own.

I'll leave you with some links to additional online resources that are designed specifically for Windows administrative scripting:

- My Web site at http://www.ScriptingAnswers.com
- The Microsoft TechNet Script Center at http://www.microsoft.com/technet/community/scriptcenter/default.mspx
- Clarence Washington's excellent Win32 Script Repository at http://cwashington.netreach.net/
- The Desktop Engineer's Junk Drawer at http://desktopengineer.com/
- Windows Scripting on MSN Groups at http://groups.msn.com/windowsscript/_homepage.msnw?pgmarket=en-us
- *Windows & .NET Magazine's* Windows Scripting Solutions at http://www.winnetmag.com/WindowsScripting/
- Chris Brooke's *Scripting for MCSEs* column at http://www.mcpmag.com/columns/columnist.asp?ColumnistsID=7

I think you'll find that the interest in administrative scripting is growing and that there is a constantly expanding set of resources for you to take advantage of. Good luck, and enjoy!