Manage Exchange Server 2003 Using Windows PowerShell and WMI

Contributed by David Noel-Davies

Exchange Management Shell is arguably the most significant feature of Microsoft Exchange Server 2007. Built on top of Windows PowerShell and the Microsoft .NET Framework, Exchange Management Shell gives an administrator Exchange 2007 server-management capabilities beyond what the GUI-based console provides. For Exchange administrators, migrating to Exchange 2007 means they’ll need to get used to the fact that a number of administrative tasks can be performed only from the command line, not the GUI console. However, if you’re still on Exchange Server 2003, there’s a way to make the transition easier and get acquainted (or reacquainted) with using the command line. You can use PowerShell in conjunction with Windows Management Instrumentation (WMI) to perform some of the same tasks you can do via Exchange Management Shell in Exchange 2007. By taking this first step, you can get a head start on an impending migration to Exchange 2007 as well as use basic command-line scripting to improve your own productivity as an Exchange administrator.

WMI 101

WMI has been an integral part of the Windows OS since Windows 2000. WMI provides a consistent interface to access core Windows management functionality in a variety of ways—for example, though Exchange System Manager (ESM), programming and scripting languages, or various command-line tools.As long as an application comes installed with the relevant WMI providers (Exchange 2003 includes these providers), which are COM objects that encapsulate managed entities such as network adapters and disks, the WMI providers work behind the scenes with the WMI infrastructure, hiding the intricate details that let you carry out management tasks locally and remotely. You needn’t be a WMI expert to exploit this underlying power, although you’ll probably find it helpful to understand WMI fundamentals. (For more information about working with WMI, see the articles and resources in the Learning Path box at the right of this article. Also see Alain Lissoir’s Understanding WMI Scripting and Leveraging WMI Scripting, both Digital Press, 2003.)PowerShell is designed from the ground up to be IT-administrator friendly. Learning the basics of WMI is relatively easy, so that you can start doing useful work fairly quickly with this powerful and flexible scripting language. Deploying PowerShell in combination with WMI is a fast and effective way to automate common management operations for Exchange 2003. Once you get comfortable with using WMI scripting, you can readily adapt what you’ve learned to manage virtually any WMI-enabled applications (including Exchange 2007). Note that to use PowerShell, you’ll need to have .NET 2.0 or later installed on the system on which you’ll run PowerShell-WMI scripts.Getting Started

Get-WmiObject is the only cmdlet you’ll need when working with WMI and PowerShell, so let’s start our PowerShell-WMI exploration by looking at this cmdlet. At a minimum, you must supply the -class parameter to tell PowerShell which namespace it should focus on. The basic syntax for Get-WmiObject looks like this: Get-WmiObject -class win32_networkadapterconfiguration(Note that all commands in this article are typed interactively at the PowerShell command line, e.g., PS C:\>.) You can omit the -class parameter if you embed the class information in a WMI Query Language (WQL) statement passed via the -query parameter, like this:$nic = Get-WmiObject -query "select * from win32_networkadapterconfiguration"Since we’re primarily interested in working with the data that Get-WmiObject will return, which is actually an array or collection of objects, it’s a good idea to save the query result into a variable—$nic in the previous example. A valid PowerShell variable always starts with a dollar sign ($) followed by alphanumeric characters. An important point to note is that Windows and most tools that work with WMI default to using the root\cimv2 namespace if no namespace is specified. To work with any Exchange class, you must explicitly point to the Exchange WMI namespace, root\MicrosoftExchangeV2, as you’ll see in a moment. Note that if you don’t specify the Exchange namespace and try to connect to an Exchange WMI provider, you’ll get an error similar to that in Figure 1.Let’s start by retrieving basic Exchange server information from a specific administrative group. To do so, you can build a WQL statement pointing to the Exchange_Server WMI class, as Listing 1 shows. Notice the where clause in the query. As a best practice, always use a filter as part of the SELECT statement to limit the query’s scope. Execution without a filter will cause the cmdlet to take a long time to retrieve the requested information. This is especially likely to happen when you’re querying an Exchange organization with a sizable number of servers over busy networks; PowerShell will appear to hang until WMI finishes the query. The query results will be returned in the $exserver variable, which will hold an array (i.e., a collection of Exchange server objects) from the specified administrative group. Notice the use of the -computer parameter in Listing 1. This parameter can come in handy when you’re managing your Exchange server farms remotely from an administrative workstation or another server in the Active Directory (AD) domain. Be aware that the Get-WmiObject cmdlet supports remote-machine access, whereas native PowerShell 1.0 cmdlets don’t yet have this capability. Alternatively, you can look into third-party solutions such as NetCmdlets offered by /n software (http://www.nsoftware.com). Using the -computer parameter is also essential if PowerShell or Microsoft .NET Framework 2.0 isn’t a standard part of your Exchange server build; hence, you can’t run PowerShell locally on the Exchange 2003 server itself. If needed, you should also include the -credential parameter to specify an account in the form domain\user with sufficient rights and permissions on the target server. By default, Get-WmiObject works with the local machine if you omit the -computer parameter.You can ask PowerShell to tell you the number of servers in the administrative group without having to drill down in ESM to Administrative Groups\First Administrative Group\Servers in your Exchange organization and painstakingly count them one by one. Simply make a call to the collection’s count properly interactively by entering $exserver.count at the PowerShell console.Discover and Do More

The Get-Member cmdlet lets you discover methods and properties that apply to the Exchange object you’re currently working on. To do so, you pipe the object to the Get-Member cmdlet, like this:$exserver | Get-

MemberYou’ll see the output in Figure 2.As you can see, by combining a series of PowerShell statements, you can easily perform many useful Exchange administration tasks. For example, to conduct a quick inventory exercise to gather information such as the number of Exchange servers in a particular administrative group and their versions, you can accomplish this task by using code like the example in Listing 2. The first statement writes the header information into the CSV file. This is necessary to describe the data in each column separated by a comma. $exserver holds the list of server objects that’s sent down the pipeline and individually enumerated by foreach (which maps to the ForEach-Object cmdlet). If you want, you can then import the resulting comma-separated value (CSV) file into a spreadsheet for further processing.If you want to reuse the code that you’ve entered at the PowerShell command line as a script, copy the lines of code into a text document and save the file with the .ps1 extension. To run the script, you must spell out the script’s full pathname or qualify with the dot (.) and backslash (\) characters if the script is located in the current folder, like this:.\get-exServer.ps1It’s a good idea to have a common location for PowerShell scripts that are used for administrative purposes.Working with Mailboxes
The most frequent day-to-day task for an Exchange administrator has to be the management of user mailboxes. Before we look at how to do this using WMI, let’s review some important information first.No two storage groups (SGs) on an Exchange server can share the same name, but the Stores (databases) within each SG can be identically named. This means that you should always qualify the WQL query with the SG name when selecting an individual store to work with. Here’s an example:$exmb = Get-WmiObject -namespace root\microsoftexchangev2 -query "select * from Exchange_mailbox where (servername='exchangeserver01') and (storagegroupname='First Storage Group')"Notice that you specify the Exchange_mailbox class in the query using the same namespace introduced earlier. Here, the number of mailboxes will be the accumulated total obtained from all Stores within the specified SG. This statement will work only if at least one Store is already mounted. That is to say, dismounted Stores won’t be included in the computation since Exchange can’t access them. Pay attention to the fact that the mailbox count includes system mailboxes such as the system attendant, SMTP (servername-{GUID}), and SystemMailbox{GUID}.To limit the set of user mailboxes to a specific Store, modify the query so it looks like this:select * from Exchange_mailbox where (servername='exchangeserver01') and (storagegroupname='sg2') and (storename='mailbox store')Successful execution of the Get-WmiObject statement will return a collection of user-mailbox objects that’s stored in the $exmb variable. You can then perform a variety of operations. Let’s say that you want to obtain a quick summary of the total disk space consumed by all user mailboxes. Here’s the code you’d use at the PowerShell interactive console:$sum = 0.0 ; $i = 0 ; $exmsb | foreach { $i++ ; $sum += $_.size } ; $i ; $sum ; $sum/$i 15 424 28.2666666666667At the left side of the pipe character (|), the $sum and $i variables are initialized to zero. The semicolon tells PowerShell that these are actually separate statements. Next, $exmb is piped to the ForEach-Object cmdlet, which adds together the size of each unique user mailbox. The current object is identified by the use of the special $_ character literal. Finally, the remaining statements (i.e., the command’s output) display the result for the total number of mailboxes processed, the sum of mailbox sizes, and the average mailbox size in KB.Another option is to use the little known Measure-Object cmdlet to achieve the same result and more. The following sample PowerShell code shows how you can identify more than one object property for the cmdlet to process: in this case, both the size and totalitems attributes of each user mailbox.$exmb | Measure-Object -sum size, totalitems -average -maximum -minimumYou’ll see this output for the command: Count : 15 Average : 28.2666666666667 Sum : 424 Maximum : 361 Minimum : 0 Property : sizeCount : 15 Average : 27.6 Sum : 414 Maximum : 401 Minimum : 0 Property : totalitemsIncreasing users’ mailbox quota is a common request for Exchange administrators, as is the need to rank the top users in an Exchange organization based on criteria such as message count and mailbox size, to enable capacity planning, for instance. You can easily obtain users’ mailbox-usage statistics by using the following PowerShell statement: $exmb | Sort-Object size -descending | Select-Object -first 5 MailboxDisplayName,Size,TotalItems | Format-Table -autosizeYou’ll see this output for the command: MailboxDisplayName   Size TotalItems ------------------   ---- ---------- Antoni Keydic   12065 501 Administrator   5080 2100 SystemMailbox{ - - - - } 361   401 Davido Jonki   125   21 Jeni Yotkieyz   112   57 (Note that the Exchange 2007 Get-MailboxStatistics cmdlet does essentially the same thing as the previous command.) Another item that will provide you with useful mailbox statistics is the read-only DeletedMessageSizeExtended property. This property gives the cumulative size of all deleted messages that have been emptied from a user’s Deleted Items folder in Outlook. In reality, these messages are still retained in the Exchange database on the server until purged, according to retention-policy settings. This is an important variable to monitor, as deleted items don’t count in a user’s total mailbox-size limit. You can obtain this information simply by including the DeletedMessageSizeExtended property in the last few code examples that I’ve shown. You can use the Get-Member command to discover other properties of $exmb that they might like to include in the report. For example, to retrieve methods and properties that apply to the object in question, enter the command $exmb | Get-MemberScript Security
If you’re a PowerShell newbie, you’ve probably seen this infamous error message when you tried to run a .ps1 script using the following command: .\get-exServer.ps1File C:\get-exServer.ps1 cannot be loaded because the execution of scripts is disabled on this system. Please see "get-help about_signing" for more details. By default, the .ps1 file extension is associated with Windows Notepad, and Notepad will automatically launch in place of PowerShell. To go one step further, Microsoft has tightened the default security settings in PowerShell to avoid another outbreak reminiscent of the “I-love-you” VBScript menace some years back. Therefore, you’ll need to change the default security execution policy to enable the script to run under PowerShell.Out of the box, PowerShell enforces the Restricted secure execution policy that prevents a script from running. To permit scripts that aren’t digitally signed to execute on the local computer, you can change the execution policy to either Unrestricted or RemoteSigned. By far,

AllSigned—which demands that all scripts, local or remote, be digitally signed before they’re allowed to run—is the most secure execution policy. Nevertheless, malicious scripts that are digitally signed can still damage your system if you aren’t careful. You can use the Get-ExecutionPolicy cmdlet to determine the execution policy that’s currently in force out of the four available as listed above. For scripts that run in testing and lab environments, setting the execution policy to RemoteSigned would fit most requirements. To set the policy to RemoteSigned, at the PowerShell console, enter Set-ExecutionPolicy RemoteSignedThe change will take effect immediately; however, to change the execution policy, you’ll need to have administrative rights and permissions on the machine on which you’ll run PowerShell scripts. As PowerShell becomes more and more prevalent, you’ll want to ensure that IT has control over users’ ability to execute PowerShell scripts. In an AD environment, consider deploying the Administrative Templates for Windows PowerShell, which let you centrally administer and control the use of PowerShell in your organization via Group Policy. You can download the templates at http://www.microsoft.com/downloads/details.aspx?FamilyID=2917a564-dbbc-4da7-82c8-fe08b3ef4e6d&DisplayLang=en.
Parting Words
I hope that by now you’re convinced that it’s relatively straightforward to put PowerShell and WMI to work for you. Using this duo gives you some of the functionality that Exchange Management Shell in Exchange 2007 provides and can be a preferable alternative to automating Exchange 2003 tasks using VBScripts, which are typically longer and more complex than PowerShell scripts. There are many other Exchange WMI classes worth investigating, and we’ll explore them in an upcoming article.

Figure 1Figure 1: Error resulting when Exchange WMI namespace isn’t specified with Get-WmiObject Get-WmiObject : Invalid class At line:1 char:14 + Get-WmiObject <<<< -class exchange_server Figure 2Figure 2: Output from running $exserver | Get-Member TypeName: System.Management.ManagementObject#root\microsoftexchangev2\Exchange_ServerName MemberType Definition EnableMessageTracking Method System.Management.ManagementBaseObject EnableMessageTracking(System.Bool... MoveMTAData Method System.Management.ManagementBaseObject MoveMTAData(System.String MTAData... AdministrativeGroup Property System.String AdministrativeGroup {get;set;} AdministrativeNote Property System.String AdministrativeNote {get;set;} Caption Property System.String Caption {get;set;} CreationTime Property System.String CreationTime {get;set;} Description Property System.String Description {get;set;} DN Property System.String DN {get;set;} ExchangeVersion Property System.String ExchangeVersion {get;set;} FQDN Property System.String FQDN {get;set;} GUID Property System.String GUID {get;set;} ... Listing 1Listing 1: WQL Statement Querying the Exchange_Server WMI Class PS C:\> $exserver = Get-WmiObject -namespace root\microsoftexchangev2 -query "select * from exchange_server where (AdministrativeGroup='First Administrative Group')" -computer ExchangeServer01 Listing 2Listing 2: Determining Basic Exchange Server Information in Your Organization PS C:\> "Name,Version,AdministrativeGroup,RoutingGroup, MonitoringEnabled" | Out-File "exinv.csv" -encoding ASCII PS C:\> $exserver = Get-WmiObject –namespace root\microsoftexchangev2 -query "select * from exchange_server where (AdministrativeGroup='First Administrative Group')" PS C:\> $exserver | foreach { >> $exserverItem = $_.name + "," + $_.exchangeversion ` >> + "," + $_.AdministrativeGroup + "," + $_.RoutingGroup ` >> + "," + $_.MonitoringEnabled >> $exserveritem | Out-File "exinv.csv" –append -encoding ASCII >> } >> PS C:\> type exinv.csv Name,Version,AdministrativeGroup,RoutingGroup,MonitoringEnabled ExchangeServer01,Version 6.5 (Build 7638.2: Service Pack 2),First Administrative Group,First Routing Group,True ExchangeServer02,Version 6.5 (Build 7638.2: Service Pack 2),First Administrative Group,First Routing Group,True PS C:\> Contents of exinv.csv Name,Version,AdministrativeGroup,RoutingGroup,MonitoringEnabled ExchangeServer01,Version 6.5 (Build 7638.2: Service Pack 2),First Administrative Group,First Routing Group,True ExchangeServer02,Version 6.5 (Build 7638.2: Service Pack 2),First Administrative Group,First Routing Group,True