# PowerTips MONTHLY

Part of the PowerShell.com reference library, brought to you by **Dr. Tobias Weltner**

Volume 5 | October 2013

This Month's Topic:

## WMI

Dr. Tobias Weltner

Sponsored by idera Application & Server Management

## Table of Contents

## 1. How WMI Works

WMI is a service that seems to work much like a database, yet WMI always returns live data. It is used to find out details about physical and logical computer configurations and works locally as well as remotely.

You basically ask for a "kind of animal" (class) and get back all live animals of that kind ("objects"). So the first thing you want to know is how you find the class names that represent the various parts of your computer.

## 2. Finding WMI Class Names

Get-WmiObject is a great and simple cmdlet to retrieve WMI information, and the -List parameter returns all available WMI class names:

```
# search for WMI class name
Get-WmiObject -Class *Share* -List

# search for WMI class name
Get-WmiObject -Class Win32_Share
```

You can also use Select-String to find interesting WMI classes. This will find all WMI classes related to "Print":

```
PS> Get-WmiObject -List | Select-String Print

Win32_PrinterDriver                    {StartService, St... {Caption, ConfigFile...
CIM_Printer                            {SetPowerState, R... {Availability, Avail...
Win32_Printer                          {SetPowerState, R... {Attributes, Availab...
Win32_PrintJob                         {Pause, Resume}      {Caption, Color, Dat...
Win32_TCPIPPrinterPort                 {}                   {ByteCount, Caption,...
Win32_PrinterConfiguration            {}                   {BitsPerPel, Caption...
Win32_PrinterSetting                   {}                   {Element, Setting}
Win32_PrinterShare                     {}                   {Antecedent, Dependent}
Win32_PrinterDriverDll                 {}                   {Antecedent, Dependent}
Win32_PrinterController                {}                   {AccessState, Antece...
Win32_PerfFormattedData_Spooler_... {}                   {AddNetworkPrinterCa...
Win32_PerfRawData_Spooler_PrintQ... {}                   {AddNetworkPrinterCa...
```

## 3. Find Useful WMI Classes

You can use Get-WmiObject to dump all WMI classes, and you can filter this by keyword. Still, you may get back too many classes.

You can improve search results though by excluding any class that has less than 5 properties (which is a rough guess that it is just a linker class without practical use for you). If you are not interested in performance counters, either, you can exclude these, too:

```
$Keyword = 'disk'

Get-WmiObject -Class "Win32_*$Keyword*" -List |
  Where-Object { $_.Properties.Count -gt 5 -and $_.Name -notlike '*_Perf*' }
```

The result looks similar to this:

```
    NameSpace: ROOT\cimv2

Name                                Methods              Properties
----                                -------              ----------
Win32_LogicalDisk                   {SetPowerState, R... {Access, Availabilit...
Win32_MappedLogicalDisk             {SetPowerState, R... {Access, Availabilit...
Win32_DiskPartition                 {SetPowerState, R... {Access, Availabilit...
Win32_DiskDrive                     {SetPowerState, R... {Availability, Bytes...
Win32_DiskQuota                     {}                   {DiskSpaceUsed, Limi...
```

Simply replace the search word in $Keyword with whatever you are looking for. And then, when you know the WMI class name, submit it to Get-WmiObject and omit the -List parameter to get the actual results:

```
Get-WmiObject -Class Win32_DiskPartition
```

## 4. Finding Commonly Used WMI Classes

To find the most useful WMI classes you can use Get-WmiObject, and let PowerShell provide you with a hand-picked list:

```
Select-Xml  $env:windir\System32\WindowsPowerShell\v1.0\types.ps1xml -XPath /Types/Type/Name |
ForEach-Object { $_.Node.InnerXML } | Where-Object { $_ -like '*#root*' } |
ForEach-Object { $_.Split('[\/]')[-1] } | Sort-Object -Unique
```

These are the WMI classes found especially helpful by the PowerShell developers:

```
CIM_DataFile
Msft_CliAlias
Win32_BaseBoard
Win32_BIOS
```

## Author Bio

Tobias Weltner is a long-term Microsoft PowerShell MVP, located in Germany. Weltner offers entry-level and advanced PowerShell classes throughout Europe, targeting mid- to large-sized enterprises. He just organized the first German PowerShell Community conference which was a great success and will be repeated next year (more on www.pscommunity.de).. His latest 950-page "PowerShell 3.0 Workshop" was recently released by Microsoft Press.

To find out more about public and in-house training, get in touch with him at tobias.weltner@email.de.

```
Win32_BootConfiguration
WIN32_CACHEMEMORY
Win32_CDROMDrive
Win32_ComputerSystem
Win32_ComputerSystemProduct
WIN32_DCOMApplication
WIN32_DESKTOP
WIN32_DESKTOPMONITOR
Win32_DeviceMemoryAddress
Win32_Directory
Win32_DiskDrive
Win32_DiskPartition
Win32_DiskQuota
Win32_DMAChannel
Win32_Environment
Win32_Group
Win32_IDEController
Win32_IRQResource
Win32_LoadOrderGroup
Win32_LogicalDisk
Win32_LogicalMemoryConfiguration
Win32_LogonSession
Win32_NetworkAdapter
Win32_NetworkAdapterConfiguration
WIN32_NetworkClient
Win32_NetworkConnection
Win32_NetworkLoginProfile
Win32_NetworkProtocol
Win32_NTDomain
Win32_NTEventlogFile
Win32_NTLogEvent
Win32_OnBoardDevice
Win32_OperatingSystem
Win32_OSRecoveryConfiguration
Win32_PageFileSetting
Win32_PageFileUsage
Win32_PerfRawData_PerfNet_Server
Win32_PhysicalMemoryArray
Win32_PingStatus
Win32_PortConnector
Win32_PortResource
Win32_Printer
Win32_PrinterConfiguration
Win32_PrintJob
Win32_Process
WIN32_PROCESSOR
Win32_ProcessXXX
Win32_Product
Win32_QuickFixEngineering
Win32_QuotaSetting
Win32_Registry
Win32_ScheduledJob
Win32_SCSIController
Win32_Service
Win32_Share
```

```
Win32_SoftwareElement
Win32_SoftwareFeature
WIN32_SoundDevice
Win32_StartupCommand
Win32_SystemAccount
Win32_SystemDriver
Win32_SystemEnclosure
Win32_SystemSlot
Win32_TapeDrive
Win32_TemperatureProbe
Win32_TimeZone
Win32_UninterruptiblePowerSupply
Win32_UserAccount
Win32_VoltageProbe
Win32_VolumeQuotaSetting
Win32_WMISetting
```

Just pick one and submit it to Get-WmiObject. Here is an example:

```
Get-WmiObject -Class Win32_BIOS
```

## Technical Editor Bio

Aleksandar Nikolic, Microsoft MVP for Windows PowerShell, a frequent speaker at the conferences (Microsoft Sinergija, PowerShell Deep Dive, NYC Techstravaganza, KulenDayz, PowerShell Summit) and the cofounder and editor of the PowerShell Magazine (http://powershellmagazine.com). He is also available for one-on-one online PowerShell trainings. You can find him on Twitter: https://twitter.com/alexandair

## 5. Finding All Object Properties

By default, PowerShell only displays a limited set of object properties. To view all available properties, add the following pipeline element and compare the results:

```
Get-WmiObject Win32_BIOS

Get-WmiObject Win32_BIOS | Select-Object * -First 1
```

The Select-Object statement selects all available columns and returns only the first element—just in case the initial command returned more than one object. Since you only need one sample object to investigate the available properties, this is a good idea to not get overwhelmed with redundant data.

To check out the formal data types and also methods present in objects, pipe the result to Get-Member instead:

```
Get-WmiObject Win32_BIOS | Get-Member
```

## 6. Getting WMI Help

To really dive into WMI, you will want to get full documentation for all the WMI classes. While documentation is available on the Internet, the URLs for those websites are cryptic.

Here is a clever way how you can translate WMI class names into the cryptic URLs: use PowerShell 3.0 Invoke-WebRequest to search for the WMI class using a major search engine, and then retrieve the URL:

```
function Get-WmiHelpLocation
{
  param ($WmiClassName='Win32_BIOS')

  $Connected = [Activator]::CreateInstance([Type]::GetTypeFromCLSID([Guid]'{DCB00C01-570F-4A9B-8D69-
199FDBA5723B}')).IsConnectedToInternet
  if ($Connected)
  {
      $uri = 'http://www.bing.com/search?q={0}+site:msdn.microsoft.com' -f $WmiClassName

      $url = (Invoke-WebRequest -Uri $uri -UseBasicParsing).Links |
      Where-Object href -like 'http://msdn.microsoft.com*' |
      Select-Object -ExpandProperty href -First 1
      Start-Process $url
      $url
  }
  else
  {
      Write-Warning 'No Internet Connection Available.'
  }
}
```

When you run Get-WmiHelpLocation, it opens the web page in your default browser that documents the WMI class you specified, and also returns the URL:

```
PS> Get-WmiHelpLocation Win32_Share
http://msdn.microsoft.com/en-us/library/aa394435(VS.85).aspx
```

## 7. Check Windows License Status

In PowerShell, you can directly access the raw licensing data like this:

```
PS> Get-WmiObject SoftwareLicensingService


__GENUS                                    : 2
__CLASS                                    : SoftwareLicensingService
__SUPERCLASS                               :
__DYNASTY                                  : SoftwareLicensingService
__RELPATH                                  : SoftwareLicensingService.Version="
                                             6.1.7601.17514"
__PROPERTY_COUNT                           : 32
__DERIVATION                               : {}
__SERVER                                   : TOBIASAIR1
__NAMESPACE                                : root\cimv2
__PATH                                     : \\TOBIASAIR1\root\cimv2:SoftwareLi
                                             censingService.Version="6.1.7601.1
                                             7514"
ClientMachineID                            :
DiscoveredKeyManagementServiceMachineName  :
DiscoveredKeyManagementServiceMachinePort  : 0
IsKeyManagementServiceMachine              : 0
KeyManagementServiceActivationDisabled     : False
KeyManagementServiceCurrentCount           : 4294967295
KeyManagementServiceDnsPublishing          : True
KeyManagementServiceFailedRequests         : 4294967295
KeyManagementServiceHostCaching            : True
KeyManagementServiceLicensedRequests       : 4294967295
KeyManagementServiceListeningPort          : 0
KeyManagementServiceLowPriority            : False
KeyManagementServiceMachine                :
KeyManagementServiceNonGenuineGraceRequests : 4294967295
KeyManagementServiceNotificationRequests    : 4294967295
KeyManagementServiceOOBGraceRequests       : 4294967295
KeyManagementServiceOOTGraceRequests       : 4294967295
KeyManagementServicePort                   : 1688
KeyManagementServiceProductKeyID           :
KeyManagementServiceTotalRequests          : 4294967295
KeyManagementServiceUnlicensedRequests     : 4294967295
PolicyCacheRefreshRequired                 : 0
RemainingWindowsReArmCount                 : 3
RequiredClientCount                        : 4294967295
TokenActivationAdditionalInfo              :
TokenActivationCertificateThumbprint        :
TokenActivationGrantNumber                 : 4294967295
TokenActivationILID                        :
TokenActivationILVID                       : 4294967295
Version                                    : 6.1.7601.17514
VLActivationInterval                       : 4294967295
VLRenewalInterval                          : 4294967295
PSComputerName                             : TOBIASAIR1
```

You can also check the license status of your copy of Windows:

```
Get-WmiObject SoftwareLicensingProduct |
  Select-Object -Property Description, LicenseStatus |
  Out-GridView
```

And you can find out which Windows SKU you are actually using:

```
PS> Get-WmiObject SoftwareLicensingProduct |
  Where-Object { $_.LicenseStatus -eq 1 } |
  Select-Object -Property Name, Description
```

```
Name                                    Description
----                                    -----------
Windows(R) 7, Ultimate edition          Windows Operating System - Windows(R...
```

## 8. Listing All WMI Namespaces

WMI is organized into namespaces which work similar to subfolders. The default namespace is “root\CIMV2”. You do not need to submit the namespace as long as the class is located inside the default namespace.

Here’s a line that lists all namespaces you have:

```
Get-WmiObject -Query “Select * from __Namespace” -Namespace Root |
  Select-Object -ExpandProperty Name
```

Next, you could investigate all classes that live in one of these particular namespaces:

```
Get-WmiObject -Namespace root\SecurityCenter2 -List
```

And then, once you know the classes, you could retrieve information, for example, about your installed antivirus protection:

```
PS> Get-WmiObject -Namespace root\SecurityCenter2 -Class AntivirusProduct


__GENUS                  : 2
__CLASS                  : AntiVirusProduct
__SUPERCLASS             :
__DYNASTY                : AntiVirusProduct
__RELPATH                : AntiVirusProduct.instanceGuid=”{641105E6-77ED-3F35-A3
                           04-765193BCB75F}”
__PROPERTY_COUNT         : 5
__DERIVATION             : {}
__SERVER                 : TOBIASAIR1
__NAMESPACE              : ROOT\SecurityCenter2
__PATH                   : \\TOBIASAIR1\ROOT\SecurityCenter2:AntiVirusProduct.in
                           stanceGuid=”{641105E6-77ED-3F35-A304-765193BCB75F}”
displayName              : Microsoft Security Essentials
instanceGuid             : {641105E6-77ED-3F35-A304-765193BCB75F}
pathToSignedProductExe   : C:\Program Files\Microsoft Security
                           Client\msseces.exe
pathToSignedReportingExe : C:\Program Files\Microsoft Security
                           Client\MsMpEng.exe
productState             : 397312
PSComputerName           : TOBIASAIR1
```

## 9. Listing Power Plans

It may be useful to automatically and temporarily switch to a “high performance” power plan from inside a script. Maybe you know that a script has to do CPU-intensive tasks, and you would like to speed it up a bit.

Here is how a script can change power plan settings:

```
# save current power plan
$PowerPlan = (Get-WmiObject -Namespace root\cimv2\power -Class Win32_PowerPlan -Filter
‘isActive=True’).ElementName
“Current power plan: $PowerPlan”
```

```
# turn on high performance power plan
(Get-WmiObject -Namespace root\cimv2\power -Class Win32_PowerPlan -Filter 'ElementName="High
Performance"').Activate()

# do something
'Power plan now is High Performance!'
Start-Sleep -Seconds 3

# turn power plan back to what it was before
(Get-WmiObject -Namespace root\cimv2\power -Class Win32_PowerPlan -Filter
"ElementName='$PowerPlan'").Activate()
"Power plan is back to $PowerPlan"
```

## 10. Determining Service Start Modes

By using WMI, you can enumerate the start mode that you want your services to use. To get a list of all services, try this:

```
Get-WmiObject Win32_Service | Select-Object Name, StartMode
```

If you want to find out the start mode of one specific service, try this instead:

```
([wmi]'Win32_Service.Name="Spooler"').StartMode
```

## 11. Identifying Computer Hardware

If you must identify computer hardware, you could do so on the hard drive serial number:

```
Get-WmiObject -Class Win32_DiskDrive | Select-Object -ExpandProperty SerialNumber
```

Unless the hard drive is exchanged, this number provides a unique identifier.

Another alternative is to use the following command:

```
Get-WmiObject -Class Win32_ComputerSystemProduct | Select-Object -ExpandProperty UUID
```

That is the ID that Windows uses to identify computers when booting via PXE, for example.

## 12. Get Logged-On User

You can use this code to find out which user is locally logged on a machine:

```
$ComputerName = 'localhost'

Get-WmiObject Win32_ComputerSystem -ComputerName $ComputerName |
    Select-Object -ExpandProperty UserName
```

Note that this will always return the user logged on to the physical machine. It will not return terminal service user or users inside a virtual machine. You will need administrator privileges on the target machine.

Get-WmiObject supports the -Credential parameter if you must authenticate as someone else.

## 13. Get All Logged-On Users

One way of finding logged-on users is to enumerate all instances of explorer.exe (which is responsible for the desktop UI), then determining the process owner:

```
$ComputerName = 'localhost'

Get-WmiObject Win32_Process -Filter 'name="explorer.exe"' -Computername $ComputerName |

ForEach-Object { $owner = $_.GetOwner(); '{0}\{1}' -f $owner.Domain, $owner.User } |
Sort-Object -Unique
```

## 14. Finding IP and MAC Addresses

When you query network adapters with WMI, it is not easy to find the active network card. To find the network card(s) that are currently connected to the network, you can filter based on NetConnectionStatus which needs to be "2" for connected cards. Then you can take the MAC information from the Win32_NetworkAdapter class and the IP address from the Win32_NetworkAdapterConfiguration class and combine both into one custom object:

```
PS> Get-WmiObject Win32_NetworkAdapter -Filter 'NetConnectionStatus=2'


ServiceName      : BCM43XX
MACAddress       : 68:A8:6D:0B:5F:CC
AdapterType      : Ethernet 802.3
DeviceID         : 7
Name             : Broadcom 802.11n Network Adapter
NetworkAddresses :
Speed            : 13000000
```

This gets you the network hardware but not the network configuration. To get the configuration data for this network card (like its IP address), get the related Win32_NetworkAdapterConfiguration instance:

```
function Get-NetworkConfig {
  Get-WmiObject Win32_NetworkAdapter -Filter 'NetConnectionStatus=2' |
    ForEach-Object {
      $result = 1 | Select-Object Name, IP, MAC
      $result.Name = $_.Name
      $result.MAC = $_.MacAddress
      $config = $_.GetRelated('Win32_NetworkAdapterConfiguration')
      $result.IP = $config | Select-Object -ExpandProperty IPAddress
      $result
    }
}

Get-NetworkConfig
```

## 15. Finding Services and Filtering Results

You can use a keyword search with WMI to find a specific service. The next line will get you all services with the keyword "cert" in their description:

```
Get-WmiObject Win32_Service -Filter 'Description like "%cert%"' |
    Select-Object Caption, StartMode, State, ExitCode
```

It uses the WMI filter and the WMI operator "like" so you should note that WMI uses "%" as wildcard.

## 16. WMI Server-Side Filtering

Whenever you use Get-WmiObject, be sure to minimize resources and maximize speed by using server-side filtering. The next line will use the slow client-side filtering with Where-Object:

```
Get-WmiObject Win32_Service |
    Where-Object {$_.Started -eq $false} |
    Where-Object { $_.StartMode -eq 'Auto'} |
    Where-Object { $_.ExitCode -ne 0} |
    Select-Object Name, DisplayName, StartMode, ExitCode
```

Note that you do not get back any results if no service is in the condition you asked for.

A faster and better approach is to filter on the WMI end by using the query parameter:

```
Get-WmiObject -Query ('select Name, DisplayName, StartMode, ' +
'ExitCode from Win32_Service where Started=false ' +
'and StartMode="Auto" and ExitCode<>0')
```

## 17. Remove Empty Results

One little known fact is that Where-Object is a simple way of removing objects with empty properties.

Let's say you want to list all network adapters that have an IP address. You can simply add Where-Object and specify the object property that needs to have content:

```
Get-WmiObject Win32_NetworkAdapterConfiguration | Where-Object { $_.IPAddress }
```

## 18. Getting IPv4 Addresses

You can use WMI to list all IPv4 addresses assigned to a computer:

```
Get-WmiObject Win32_NetworkAdapterConfiguration |
    Where-Object { $_.IPEnabled -eq $true } |
    ForEach-Object { $_.IPAddress } |
    ForEach-Object { [IPAddress]$_ } |
    Where-Object { $_.AddressFamily -eq 'InterNetwork' } |
    ForEach-Object { $_.IPAddressToString }
```

You can also query remote systems this way since Get-WmiObject supports the –ComputerName parameter. The trick to focusing on IPv4 only is to convert the IP address to the IPAddress type and then check that its AddressFamily is "InterNetwork."

## 19. Finding Unused Drives

You can use the cmdlet called Test-Path to test whether a given path exists. However, existing drives not currently in use, such as a CD-ROM drive, will be reported as not present. Therefore, Test-Path would never report a CD-ROM drive as present when no media is inserted.

There are numerous ways to work around this. One is the Win32_LogicalDisk WMI class. It gets you the information required:

```
Get-WmiObject Win32_LogicalDisk | Select-Object -Property *
Get-WmiObject Win32_LogicalDisk |
    Select-Object -Property DeviceID, Description, FileSystem, Providername
```

For example, to use this data to find existing drives by simply placing the drive letter into an array like this:

```
$drives = Get-WmiObject Win32_LogicalDisk |
    ForEach-Object { $_.DeviceID }
```

Next, you can use the -contains operator to check whether a specific drive exists:

```
PS> $drives -contains ‘C:’

True

PS> $drives -contains ‘P:’

False
```

## 20. List Local Groups

If you’d like to get a list of all local groups on your computer, you can use this line:

```
Get-WmiObject Win32_Group -Filter “domain=’$env:computername’” |
  Select-Object Name,SID
```

This uses WMI to retrieve all groups where the domain part is equal to your local computer name and returns the group name and SID.

## 21. Check for a Battery

If your script needs to know whether your computer has a battery, you can ask WMI. Here is a small function:

```
Function Test-Battery
{
  if ((Get-WmiObject Win32_Battery))
  {
    $true
  }
  else
  {
    $false
  }
}

Test-Battery
```

The “If” clause uses a little trick: if WMI returns anything, this is always converted to $true, so the condition is met. If WMI returns nothing, this is always converted to $false, so the condition is not met.

## 22. Displaying Battery Charge in Your Prompt

If you use a notebook and are on the road often, you may want a way to check the battery status so you can shut down your stuff in time. Here is a custom prompt function which displays the remaining charge in your prompt and also displays the current path in your console’s title bar:

```
Function Prompt
{
  $charge = (Get-WmiObject Win32_Battery |
    Measure-Object -Property EstimatedChargeRemaining -Average).Average

  $prompt = “PS [$charge%]> “

  if ($charge -lt 30)
  {
    Write-Host $prompt -ForegroundColor Red -NoNewline
  }
```

```
    elseif ($charge -lt 60)
    {
      Write-Host $prompt -ForegroundColor Yellow -NoNewline
    }
    else
    {
      Write-Host $prompt -ForegroundColor Green -NoNewline
    }

    $Host.UI.RawUI.WindowTitle = (Get-Location)
    ''
  }
```

Note how your prompt will change color from green over yellow to red, depending on the remaining battery charge. It could now look similar to this:

```
PS [48%]>
```

## 23. Averaging Multiple Objects

PowerShell is a dynamic language, and as Forest Gump put it, you never know what you get. For example, when you try and figure out the battery charge on a notebook, the following command may return null, one or multiple results depending on the number of batteries in your system:

```
Get-WmiObject Win32_Battery
```

To find out the overall charge of all batteries in your system, simply use Measure-Object. Measure-Object happily accepts an arbitrary number of (equal) objects and aggregates information. To find out the overall charge remaining in all of your batteries, use this:

```
Get-WmiObject Win32_Battery | Measure-Object EstimatedChargeRemaining -Average

(Get-WmiObject Win32_Battery |
  Measure-Object EstimatedChargeRemaining -Average).Average
```

Note that this call will fail if there is no battery at all in your system because then Measure-Object is unable to find the EstimatedchargeRemaining property.

## 24. Finding Computer Serial Number

Professionally-managed computer systems have an individual serial number that often is printed on the enclosure. WMI can then retrieve the serial number locally and remotely, which is a lot more convenient than climbing under the desk and reading the number from the physical enclosure:

```
$serial = Get-WmiObject Win32_SystemEnclosure | Select-Object -ExpandProperty serialnumber


"Computer serial number: $serial"
```

If no serial number is returned, then your computer hardware manufacturer did not include the information. This is sometimes the case with cheap off-the-shelf machines.

## 25. Renaming Computers

WMI can rename computers (provided you have Administrator privileges and know what you do).

The next example will read the serial number from the system enclosure class and rename the computer accordingly:

```
$serial = Get-WmiObject Win32_SystemEnclosure | Select-Object -ExpandProperty serialnumber
if ($serial -ne 'None') {
  $computer = Get-WmiObject Win32_ComputerSystem
  $computer.Rename("DESKTOP_$serial")
} else {
  Write-Warning 'Computer has no serial number'
}
```

## 26. Use WMI and WQL!

WMI is a great information resource, and Get-WmiObject makes it easy to retrieve WMI instances. First, use -List parameter to find WMI class names. For example, find classes that deal with network:

```
Get-WmiObject -List Win32_*network*
```

Next, pick one of the classes and enumerate its instances:

```
Get-WmiObject Win32_NetworkAdapterConfiguration
```

With WQL, a SQL-type query language for WMI, you can even create more sophisticated queries, such as:

```
Get-WmiObject -Query 'Select * FROM Win32_NetworkAdapterConfiguration WHERE IPEnabled=True'
```

To learn more about the specific WQL format and the keywords you can use, check this reference:
http://msdn.microsoft.com/en-us/library/windows/desktop/aa394606(v=vs.85).aspx

## 27. Accessing Individual Files and Folders Remotely via WMI

WMI is an excellent way of remotely checking files or folders since it has the ability to access individual files and folders, and also works locally as well as remotely. Files are represented by the WMI class CIM_DataFile whereas folders are represented by Win32_Directory.

However, you should never ask WMI for all instances of either class. Doing so would enumerate every single file or folder on any drive on the target computer, which not only takes a long time but also can easily let you run out of memory and lock down the CPU.

Always use specific filters when accessing files or folders. The next line gets you all folders in the root folder of C: drive. To use this remotely, add the -ComputerName property:

```
Get-WmiObject Win32_Directory -Filter 'Drive="C:" and Path="\\"' |
  Select-Object -ExpandProperty Name
```

Note the double-backslash: in WMI queries, you need to escape the backslash character with another one.

The next line retrieves all log files in your Windows folder, assuming the Windows folder is C:\Windows on the target system:

```
Get-WmiObject CIM_DataFile -Filter 'Drive="C:" and Path="\\Windows\\" and Extension="log"'|
  Select-Object -ExpandProperty Name
```

Note that both examples limit information to only the “Name” property. You can get back any kind of information, such as file size and write times:

```
Get-WmiObject CIM_DataFile -Filter ‘Drive=”C:” and Path=”\\Windows\\” and Extension=”log”’|
  Select-Object -Property *

Get-WmiObject CIM_DataFile -Filter ‘Drive=”C:” and Path=”\\Windows\\”’|
  Select-Object -Property Name, FileType, FileSize, EightDotThreeFileName
```

Avoid using the “like” operator. You can use it to find files recursively, but this can lead to very large result sets, and it can fail if a folder contains “illegal” path names (such as path names longer than 256 characters).

The next line would get you all text files in the folder “c:\temp” (which must exist of course) and all of its subfolders

```
Get-WmiObject CIM_DataFile -Filter ‘Drive=”C:” and Path like “\\temp\\%” and Extension=”txt”’ |
  Select-Object -Property Caption, EightDotThreeFileName, FileSize
```

Note again that these WMI queries can fail or report cryptic errors if file paths exceed the 256 character file path length.

## 28. Detect DHCP State

You can use WMI and Win32_NetworkAdapterConfiguration to determine whether a computer uses DHCP to acquire an IP address. Simply look for all instances where IPEnabled is true and DHCPEnabled is also true.

The next line will provide you with the number of network adapters using DHCP:

```
$number = Get-WmiObject Win32_NetworkAdapterConfiguration -Filter ‘IPEnabled=true and
DHCPEnabled=true’ |
  Measure-Object |
  Select-Object -ExpandProperty Count

“There are $number NICs with DHCP enabled”

$DHCPEnabled = $number -gt 0
“DHCPEnabled? $DHCPEnabled”
```

## 29. Map Network Drive

Sure you can use the command net use to map a network drive. But this would not check for existing mapped drives. Here’s a small function that first checks to see that the URL you are mapping to does not yet exist, avoiding duplicate mapped drives:

```
function New-MapDrive {
    param($Path)

    $present = @(Get-WmiObject Win32_Networkconnection |
       Select-Object -ExpandProperty RemoteName)

    if ($present -contains $Path)
    {
        “Network connection to $Path is already present”
    }
    else
    {
        net use * $Path
    }
}
```

## 30. Use WMI to Create Hardware Inventory

You can use Get-WmiObject to create a high-level hardware report. Instead of submitting a specific device class name, you should use a generic device class that applies to all hardware devices:

```
Get-WmiObject -Class CIM_LogicalDevice |
  Select-Object -Property Caption, Description, __Class
```

The __Class column lists the specific device class. If you'd like to find out more details about a specific device class, you can then pick the name and query again:

```
Get-WmiObject -Class Win32_PnPEntity |
  Select-Object Name, Service, ClassGUID |
  Sort-Object ClassGUID
```

## 31. Check Active Internet Connection

If your machine is connected to the Internet more than once, let's say wired and wireless at the same time, which connection is used? Here's a function that can tell:

```
function Get-ActiveConnection {
    Get-WmiObject Win32_NetworkAdapter -Filter "AdapterType='Ethernet 802.3'" |
    ForEach-Object { $_.GetRelated('Win32_NetworkAdapterConfiguration') } |
    Select-Object Description, Index, IPEnabled, IPConnectionMetric
}

Get-ActiveConnection
```

## 32. Finding Out Video Resolution

You can use this function if you would like to check the current video resolution on a local or remote system:

```
Function Get-Resolution
{
  param
  (
    [Parameter(ValueFromPipeline=$true)]
    [Alias('cn')]
    $ComputerName = $env:COMPUTERNAME
  )

  Process
  {
    Get-WmiObject -Class Win32_VideoController -ComputerName $Computername |
      Select-Object *resolution*, __SERVER
  }
}
```

When you run Get-Resolution, you will retrieve your own video resolution. By adding a computer name, you will find the same information is returned from a remote system (provided you have sufficient privileges to access the remote system).

## 33. Converting WMI Date and Time

WMI uses a specific (DMTF) format for date and time.

You can easily convert regular date and time information into this format. This would convert the current date and time into DMTF format:

```
$date = Get-Date
$wmidate = [System.Management.ManagementDateTimeConverter]::ToDmtfDateTime($date)
```

The result looks similar to this:

```
PS> $wmidate
20130827132307.113358+120
```

Likewise, you can convert WMI date and time information back to a regular date and time object. This code tells you when your system was rebooted the last time:

```
$os = Get-WmiObject -Class Win32_OperatingSystem
$bootTime = $os.LastBootUpTime
$bootTimeNice = [System.Management.ManagementDateTimeConverter]::ToDateTime($bootTime)
```

The raw WMI date and time information would look like this:

```
PS> $bootTime
20130825072242.307198+120
```

The converted date and time however is much more readable:

```
PS> $bootTimeNice

Sunday, August 25, 2013 7:22:42 AM
```

## 34. Checking System Uptime

This piece of code returns the days a given system is running since the last reboot:

```
$os = Get-WmiObject -Class Win32_OperatingSystem
$boottime = [System.Management.ManagementDateTimeConverter]::ToDateTime($os.LastBootupTime)
$timedifference = New-TimeSpan -Start $boottime
$days = $timedifference.TotalDays
'Your system is running for {0:0.0} days.' -f $days
```

The result looks similar to this:

```
Your system is running for 2,2 days.
```

## 35. Free Space on Disks

You can use WMI to determine how much free space is available on any given disk:

```
Get-WmiObject Win32_LogicalDisk |
ForEach-Object {
  'Disk {0} has {1:0.0} MB space available' -f $_.Caption, ($_.FreeSpace / 1MB)
}
```

Here is how it works:

Get-WmiObject returns instances of a WMI class. The Win32_LogicalDisk WMI class represents disk drives. Each drive is then processed within the ForEach-Object script block and represented by the special variable $_.

Using the -f formatting operator, it is easy to fill in the requested information by reading the Caption and FreeSpace property. Note that you can convert the bytes to megabytes by dividing bytes by 1MB. If you wanted gigabytes, change 1MB to 1GB. Just make sure you don't have a space between number and unit. 1 MB will not work.

## 36. Backing Up Event Log Files

WMI provides a method to backup event log files as *.evt/*.evtx files. The code below creates backups of all available event logs:

```
Get-WmiObject Win32_NTEventLogFile |
    ForEach-Object {
      $filename = "$home\" + $_.LogfileName + '.evtx'
      Remove-Item $filename -ErrorAction SilentlyContinue
      $null = $_.BackupEventLog($filename).ReturnValue
    }
```

## 37. Translate EventID to InstanceID

Sometimes, you may need to have the event ID for a system event, though what you really need is the instance ID. For example, Get-EventLog will only support instance IDs, but no event IDs. Here is a function that can translate event IDs into instance IDs:

```
 Function ConvertTo-InstanceID
{
  param
  (
    [Parameter(Mandatory=$true)]
    $EventID
  )

  try {
    Get-WmiObject Win32_NTLogEvent -Filter "EventCode=$EventID" |
    ForEach-Object { $_.EventIdentifier; Throw 'Done' }
  } catch {}
}
```

## 38. Formatting a Drive

In Windows Vista and higher, there is a new WMI class called Win32_Volume that you can use to format drives.

Warning: You should be careful when formatting data on a drive. All data on that drive will be lost.

The following line will format drive G:\ using NTFS file system, provided you have appropriate permissions:

```
# WARNING: This will format drive G:\
# DATA WILL BE LOST!
$DriveLetter = 'G'
$drive = Get-WmiObject -Class Win32_Volume -Filter ("DriveLetter='{0}:'" -f $DriveLetter)
$drive.Format('NTFS',$true,4096,"",$false)
```

## 39. Disabling Automatic Page Files

If you would like to programmatically control page file settings, you can use WMI but must enable all privileges using -EnableAllPrivileges. The code below will disable automatic page files:

```
$computer = Get-WmiObject Win32_ComputerSystem -EnableAllPrivileges
$computer.AutomaticManagedPagefile = $false
$computer.Put()
```

## 40. Enumerating Network Adapters

Finding your network adapters with WMI isn't always easy because WMI treats a lot of "network-like" adapters like network adapters. To find only those adapters that are also listed in your Control Panel, you should make sure to filter out all adapters that have no NetConnectionID, which is the name assigned to a network adapter in your Control Panel:

```
Function Get-MyNetworkAdapter
{
  Get-WmiObject Win32_NetworkAdapter -Filter 'NetConnectionID!=null'
}

Get-MyNetworkAdapter | Select-Object -Property Name, ServiceName, DeviceID
```

With your new function, you can then easily list all network adapters and check their status.

## 41. Resetting Network Adapters

Sometimes, it is necessary to reset network adapters, such as after you changed settings for that adapter in your Registry. Resetting a network adapter is done by first disabling and then enabling it again. Here are three functions you can use:

```
Function Disable-NetworkAdapter
{
  param
  (
    $NetworkName
  )

  Get-WmiObject Win32_NetworkAdapter -Filter "NetConnectionID='$NetworkName'" |
  ForEach-Object {
    $rv = $_.Disable().ReturnValue

    if ($rv -eq 0)
    {
      '{0} disabled' -f $_.Caption
    }
    else
    {
      '{0} could not be disabled. Error code {1}' -f $_.Caption, $rv
    }
  }
}

Function Enable-NetworkAdapter
{
  param
  (
    $NetworkName
  )

  Get-WmiObject Win32_NetworkAdapter -Filter "NetConnectionID='$NetworkName'" |
  ForEach-Object {
    $rv = $_.Enable().ReturnValue

    if ($rv -eq 0)
    {
      '{0} enabled' -f $_.Caption
    }
    else
    {
      '{0} could not be enabled. Error code {1}' -f $_.Caption, $rv
    }
  }
}
```

```
Function Restart-NetworkAdapter
{
  param
  (
    $NetworkName
  )

  Disable-NetworkAdapter $NetworkName
  Enable-NetworkAdapter $NetworkName
}
```

Try this to reset a network adapter:

```
Restart-NetworkAdapter LAN-Connection
```

A restart will require Administrator privileges. Otherwise, you will receive error code 5 (Access Denied).

## 42. Finding WMI Instance Path Names

WMI can access instances directly using their individual instance path. Here is how you can find that path for any WMI object:

```
Get-WmiObject Win32_Share | Select-Object __Path
```

You can simply retrieve WMI instances and look at the "__Path" property. Be sure to note that this property begins with two underscores, not just one.

## 43. Accessing WMI Instances Directly

If you know the path to a WMI instance, you can access it directly by converting the WMI path to a WMI object:

```
[WMI]'Win32_Service.Name="W32Time"'
[WMI]'Win32_Logicaldisk="C:"'
```

You can also specify the full WMI path, including a machine name to access WMI objects on remote systems (provided you have sufficient access rights):

```
[WMI]'\\SERVER5\root\cimv2:Win32_Service.Name="W32Time"'
```

This line will access the WMI object representing drive C: and will tell you a ton of interesting details about that drive:

```
[WMI]'Win32_LogicalDisk="C:"'
```

To query remote machines, append the full WMI instance path. This will get drive details for drive "C:" on the remote machine "storage1":

```
$ComputerName = 'storage1'
[WMI]"\\$ComputerName\root\cimv2:Win32_LogicalDisk='C:'"
```

## 44. View Object Inheritance

A hidden object property called "PSTypeNames" will tell you the object type as well as the inherited chain:

```
PS> (Get-WMIObject Win32_BIOS).PSTypeNames
System.Management.ManagementObject#root\cimv2\Win32_BIOS
System.Management.ManagementObject#root\cimv2\CIM_BIOSElement
System.Management.ManagementObject#root\cimv2\CIM_SoftwareElement
System.Management.ManagementObject#root\cimv2\CIM_LogicalElement
System.Management.ManagementObject#root\cimv2\CIM_ManagedSystemElement
System.Management.ManagementObject#Win32_BIOS
System.Management.ManagementObject#CIM_BIOSElement
System.Management.ManagementObject#CIM_SoftwareElement
System.Management.ManagementObject#CIM_LogicalElement
System.Management.ManagementObject#CIM_ManagedSystemElement
System.Management.ManagementObject
System.Management.ManagementBaseObject
System.ComponentModel.Component
System.MarshalByRefObject
System.Object
```

In contrast to GetType(), this property will work for all objects, including COM objects. The most specific type is always found at the beginning of that array:

```
(Get-WmiObject Win32_BIOS).PSTypeNames[0]
```

## 45. Calling ChkDsk via WMI

Some WMI classes contain methods that you can call to invoke some action. For example, the next line initiates a disk check on drive D:

```
([WMI]"Win32_LogicalDisk='D:'").Chkdsk($true, $false, $false, $false, $false, $true).ReturnValue
```

Make sure you have Administrator privileges or else you receive a return value of 2 which corresponds to "Access Denied".

The previous one-liner translates to these steps:

```
$drive = [WMI]"Win32_LogicalDisk='D:'"
$result = $drive.Chkdsk($true, $false, $false, $false, $false, $true)
$result.ReturnValue
```

If you'd like to know what the arguments are for, ask for a method signature:

```
PS> ([wmi]"Win32_LogicalDisk='D:'").Chkdsk

OverloadDefinitions
-------------------
System.Management.ManagementBaseObject Chkdsk(System.Boolean FixErrors,
System.Boolean VigorousIndexCheck, System.Boolean SkipFolderCycle,
System.Boolean ForceDismount, System.Boolean RecoverBadSectors, System.Boolean
OkToRunAtBootUp)
```

## 46. Calculating Server Uptime

Have you ever wanted to find out the effective uptime of your PC (or some servers)? The information can be found inside the event log. Here is an example on how to select and prepare event data and create a report. This even works remotely.

```
Function Get-UpTime
{
  param
  (
    $ComputerName='localhost'
  )

  Get-WmiObject Win32_NTLogEvent -Filter 'Logfile="System" and EventCode>6004 and EventCode<6009'
-ComputerName $ComputerName |
  ForEach-Object {
    $rv = $_ | Select-Object EventCode, TimeGenerated
    switch ($_.EventCode) {
      6006 { $rv.EventCode = 'shutdown' }
      6005 { $rv.EventCode = 'start' }
      6008 { $rv.EventCode = 'crash' }
    }
    $rv.TimeGenerated = $_.ConvertToDateTime($_.TimeGenerated)
    $rv
  }
}
```

The result will look similar to this:

```
PS> Get-UpTime

EventCode                               TimeGenerated
---------                               -------------
start                                   01.09.2013 16:14:15
shutdown                                01.09.2013 16:13:16
start                                   17.08.2013 13:39:27
shutdown                                17.08.2013 13:38:30
start                                   05.08.2013 20:03:35
shutdown                                05.08.2013 20:02:37
start                                   21.07.2013 10:15:05
crash                                   21.07.2013 10:15:05
(...)
```

## 47. Get Running Process Owners

If you need to filter running processes by owner, for example to terminate the ones owned by some user, you should use WMI and the GetOwner() method. This code will retrieve all processes from a local or remote system and add an Owner property, which you can then use to select or filter processes:

```
Get-WmiObject Win32_Process |
    ForEach-Object {
      $ownerraw = $_.GetOwner();
      $owner = '{0}\{1}' -f $ownerraw.domain, $ownerraw.user;
      $_ | Add-Member NoteProperty Owner $owner -PassThru
    } |
    Select-Object Name, Owner
```

Note that you can get owner information for other users only when you have Administrator privileges. The result may look like this:

```
(...)
TscHelp.exe                             TobiasAir1\Tobias
SnagPriv.exe                            TobiasAir1\Tobias
iPodService.exe                         NT AUTHORITY\SYSTEM
SnagitEditor.exe                        TobiasAir1\Tobias
splwow64.exe                            TobiasAir1\Tobias
wmpnetwk.exe                            NT AUTHORITY\NETWORK SERVICE
taskhost.exe                            TobiasAir1\Tobias
OSPPSVC.EXE                             NT AUTHORITY\NETWORK SERVICE
PresentationFontCache.exe               NT AUTHORITY\LOCAL SERVICE
svchost.exe                             NT AUTHORITY\LOCAL SERVICE
WmiPrvSE.exe                            NT AUTHORITY\NETWORK SERVICE
OUTLOOK.EXE                             TobiasAir1\Tobias
WINWORD.EXE                             TobiasAir1\Tobias
(...)
```

## 48. Forwarding Selected Parameters with PowerShell

With all the power found in WMI and CIM, you may want to create your own functions for selected tasks that internally use WMI and CIM.

To produce functions that can work locally and remotely as well, make sure you always add the parameters -Credential and -ComputerName to your functions. You then can forward those using splatting to the WMI/CIM cmdlet.

Here is an example:

```
Function Get-BIOS
{
  param
  (
    $ComputerName,

    $Credential,

    [switch]$Verbose
  )

  $PSBoundParameters.Remove('Verbose') | Out-Null

  $bios = Get-WmiObject Win32_BIOS @PSBoundParameters

  if ($verbose)
  {
    $bios | Select-Object *
  }
  else
  {
    $bios
  }
}


Get-BIOS -Verbose
Get-BIOS -ComputerName 'YOURSERVERNAME_HERE' -Verbose
```

The function supports three parameters, but only two should be forwarded to Get-WmiObject. The remaining parameter -Verbose is used internally by the function.

To prevent -Verbose from being forwarded to Get-WmiObject, you can remove that key from $PSBoundParameters before you splat it to Get-WmiObject.

## 49. Listing Processes and Process Ownership

Get-ProcessEx is a clever function that returns process information including ownership, and it works remotely, too.

```
Function Get-ProcessEx
{
  param
  (
    $Name='*',

    $ComputerName,

    $Credential
  )

  $null = $PSBoundParameters.Remove('Name')
  $Name = $Name.Replace('*','%')

  Get-WmiObject -Class Win32_Process @PSBoundParameters -Filter "Name like '$Name'" |
  ForEach-Object {
    $result = $_ | Select-Object Name, Owner, Description, Handle
    $Owner = $_.GetOwner()
```

```
      if ($Owner.ReturnValue -eq 2)
      {
        $result.Owner = 'Access Denied'
      }
      else
      {
        $result.Owner = '{0}\{1}' -f ($Owner.Domain, $Owner.User)
      }

      $result
    }
  }
```

So if you wanted to know who is running PowerShell on your system, check this out:

```
PS> Get-ProcessEx power*exe

Name                 Owner                Description          Handle
----                 -----                -----------          ------
powershell_ise.exe   TobiasAir1\Tobias    powershell_ise.exe   6056
powershell_ise.exe   TobiasAir1\Tobias    powershell_ise.exe   7392
```

Likewise, you can now check who is currently visiting your computer through PowerShell Remoting. Just look for processes named "wsmprovhost.exe".

## 50. Get Process Owners

One way to find out the owner of a process is to add the missing Owner property to process objects. You will get regular process objects that now have an additional property called "Owner".

This code will look for any process name that starts with "power" and ends with ".exe", then find out the process owner, add it to the process object, and it return it to you:

```
$processes = Get-WmiObject Win32_Process -Filter "name like 'power%.exe'"

$appendedprocesses =
ForEach ($process in $processes)
{
  Add-Member -MemberType NoteProperty -Name Owner -Value (
    $process.GetOwner().User) -InputObject $process -PassThru
}

$appendedprocesses | Select-Object -Property name, owner
```

A loop adds the necessary Owner property. You could then use the information in Owner to selectively stop all processes owned by a specific user.

## 51. Change Service Account Password

Ever wanted to automatically change the password a service uses to log on to its account? WMI has a solution. Have a look:

```
$localaccount = '.\sample-de-admin-local'
$newpassword = 'secret@Passw0rd'

$service = Get-WmiObject Win32_Service -Filter "Name='Spooler'"
$service.Change($null,$null,$null,$null,$null,$null,$localaccount, $newpassword)
```

These lines assign a new user account and password for the Spooler service. Note that your account will need special privileges to be able to do that. If you get an “access denied” error, open services.msc and try to change the account manually using the GUI. You will get a message if your account lacks the necessary privileges and the privileges are added.

## 52. Using Clear Text Descriptions

Sometimes, WMI results use cryptic numeric IDs to describe various information. If you know the meaning of these cryptic constants, you can easily replace them with clear text descriptions.

Whenever you pipe objects through Select-Object, you actually get a copy of the original object. All properties are now readable and writeable, so you can change the object properties in any way you like. This example reads memory information and then replaces the cryptic form factor and memory type ID numbers with clear text values:

```
$memorytype = ‘Unknown’, ‘Other’, ‘DRAM’, ‘Synchronous DRAM’,
‘Cache DRAM’, ‘EDO’, ‘EDRAM’, ‘VRAM’, ‘SRAM’,
‘RAM’, ‘ROM’, ‘Flash’, ‘EEPROM’, ‘FEPROM’,
‘EPROM’, ‘CDRAM’, ‘3DRAM’, ‘SDRAM’, ‘SGRAM’,
‘RDRAM’, ‘DDR’, ‘DDR-2’
$formfactor = ‘Unknown’, ‘Other’, ‘SIP’, ‘DIP’, ‘ZIP’, ‘SOJ’,
‘Proprietary’, ‘SIMM’, ‘DIMM’, ‘TSOP’, ‘PGA’,
‘RIMM’, ‘SODIMM’, ‘SRIMM’, ‘SMD’, ‘SSMP’, ‘QFP’,
‘TQFP’, ‘SOIC’, ‘LCC’, ‘PLCC’, ‘BGA’, ‘FPBGA’, ‘LGA’

# raw results with cryptic ID numbers
Get-WmiObject Win32_PhysicalMemory |
Select-Object BankLabel, FormFactor, MemoryType
‘-’ * 40

# clear text results
Get-WmiObject Win32_PhysicalMemory |
Select-Object BankLabel, FormFactor, MemoryType |
ForEach-Object {
  $_.FormFactor = $formfactor[$_.FormFactor]
  $_.MemoryType = $formfactor[$_.MemoryType]
  $_
}
```

Here is the result, first with raw ID number, second with clear text:

```
BankLabel                                   FormFactor           MemoryType
---------                                   ----------           ----------
BANK 0                                              12                    0
BANK 1                                              12                    0
----------------------------------------
BANK 0                                          SODIMM              Unknown
BANK 1                                          SODIMM              Unknown
```

## 53. How Much RAM Do You Have?

By using hash tables, you can easily convert numeric IDs to clear text plus rename the resulting property (column):

```
$memorytype = ‘Unknown’, ‘Other’, ‘DRAM’, ‘Synchronous DRAM’, ‘Cache DRAM’,
‘EDO’, ‘EDRAM’, ‘VRAM’, ‘SRAM’, ‘RAM’, ‘ROM’, ‘Flash’, ‘EEPROM’, ‘FEPROM’,
‘EPROM’, ‘CDRAM’, ‘3DRAM’, ‘SDRAM’, ‘SGRAM’, ‘RDRAM’, ‘DDR’, ‘DDR-2’

$formfactor = ‘Unknown’, ‘Other’, ‘SIP’, ‘DIP’, ‘ZIP’, ‘SOJ’, ‘Proprietary’,
‘SIMM’, ‘DIMM’, ‘TSOP’, ‘PGA’, ‘RIMM’, ‘SODIMM’, ‘SRIMM’, ‘SMD’, ‘SSMP’,
‘QFP’, ‘TQFP’, ‘SOIC’, ‘LCC’, ‘PLCC’, ‘BGA’, ‘FPBGA’, ‘LGA’

$col1 = @{Name=’Size (GB)’; Expression={ $_.Capacity/1GB } }
$col2 = @{Name=’Form Factor’; Expression={$formfactor[$_.FormFactor]} }
$col3 = @{Name=’Memory Type’; Expression={ $memorytype[$_.MemoryType] } }
```

```
Get-WmiObject Win32_PhysicalMemory | Select-Object BankLabel, $col1, $col2, $col3
```

## 54. Create HTML System Reports

ConvertTo-Html is a great way of creating system reports because you can combine information gathered from different sources into one report. The following code will create a report with service information that is gathered from Get-Service, operating system overview returned by WMI, and installed software read directly from the Registry:

```
$style = @'
<style>
body { background-color:#EEEEEE; }
body,table,td,th { font-family:Tahoma; color:Black; Font-Size:10pt }
th { font-weight:bold; background-color:#AAAAAA; }
td { background-color:white; }
</style>
'@

& {
  "<HTML><HEAD><TITLE>Inventory Report</TITLE>$style</HEAD>"
  "<BODY><h2>Report for '$env:computername'</h2><h3>Services</h3>"

  Get-Service |
      Sort-Object Status, DisplayName |
      ConvertTo-Html DisplayName, ServiceName, Status -Fragment

  '<h3>Operating System Details</h3>'

  Get-WmiObject Win32_OperatingSystem |
      Select-Object * -ExcludeProperty __* |
      ConvertTo-Html -As List -Fragment

  '<h3>Installed Software</h3>'

  Get-ItemProperty
Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\* |
      Select-Object DisplayName, InstallDate, DisplayVersion, Language |
      Sort-Object DisplayName | ConvertTo-Html -Fragment

  '</BODY></HTML>'
} | Out-File $env:temp\report.hta
Invoke-Item $env:temp\report.hta
```

## 55. Creating HTML Reports

By adding a bit of custom formatting, your reports can be colorful and cool. The following example shows how to retrieve all error event log entries from all event logs and nicely output them as HTML report.

Note that the code will only include the local system. Need to include more than one system in your report? Simply add more IP addresses or computer names as a comma-separated list:

```
$head = '
<style>
BODY {font-family:Verdana; background-color:lightblue;}
TABLE {border-width: 1px;border-style: solid;border-color: black;border-collapse: collapse;}
TH {font-size:1.3em; border-width: 1px;padding: 2px;border-style: solid;border-color:
black;background-color:#FFCCCC}
TD {border-width: 1px;padding: 2px;border-style: solid;border-color: black;background-color:yellow}
</style>'

$header = '<H1>Last 24h Error Events</H1>'
$title = 'Error Events Within 24 Hrs'
```

```
'127.0.0.1' |
ForEach-Object {
  $time = [System.Management.ManagementDateTimeConverter]::ToDmtfDateTime((Get-Date).AddHours(-24))
  Get-WmiObject Win32_NTLogEvent -ComputerName $_ -Filter "EventType=1 and TimeGenerated>='$time'" |
  ForEach-Object {
    $_ | Add-Member NoteProperty TimeStamp (
[System.Management.ManagementDateTimeConverter]::ToDateTime($_.TimeWritten)) ; $_
  }
} |
Select-Object __SERVER, LogFile, Message, EventCode, TimeStamp |
ConvertTo-Html -Head $head -Body $header -Title $title |
Out-File $home\report.htm

Invoke-Item "$home\report.htm"
```

## 56. Printing Results

Out-Printer can print results to your default printer. You can also print to any other printer when you specify its name:

```
Get-Process | Out-Printer -Name 'Microsoft XPS Document Writer'
```

You can ask WMI to find out the names of your installed printers:

```
Get-WmiObject Win32_Printer |
    Select-Object -ExpandProperty Name |
    Sort-Object
```

If you install a PDF printer, Out-Printer then can even create PDF reports. Simply choose the PDF printer for output.

## 57. Combining Objects

Sometimes it becomes necessary to consolidate two or more objects into one. In PowerShell 3.0, Add-Member is now able to take a hash table and add its contents as note properties to another object.

Here is sample code. It gets BIOS information and operating system information from the WMI. These are two separate objects. Now, the BIOS object is converted into a hash table and then added to the other object.

$os now contains the properties of both objects (conflicting properties are removed which is why errors are suppressed):

```
$bios = Get-WmiObject -Class Win32_BIOS
$os = Get-WmiObject -Class Win32_OperatingSystem
$hashtable = $bios |
  Get-Member -MemberType *Property |
  Select-Object -ExpandProperty Name |
  Sort-Object |
  ForEach-Object -Begin { $rv=@{} } -Process {
    Write-Warning "Adding $_"
    $rv.$_ = $bios.$_
    } -End {$rv}

$os | Add-Member ($hashtable) -ErrorAction SilentlyContinue
```

When you output $os, at first sight it did not seem to change because PowerShell continues to display the default properties. However, the BIOS information is now added to it:

```
PS> $os | Select-Object *BIOS*


SMBIOSPresent        : True
SMBIOSMajorVersion   : 2
BiosCharacteristics  : {7, 11, 12, 15...}
SMBIOSBIOSVersion    : MBA41.88Z.0077.B0E.1110141154
(...)
```

So you can now use Select-Object to combine information from both WMI classes into one tabl So you can now use Select-Object to combine information from both WMI classes into one table:

```
PS> $os | Select-Object Caption, OSArchitecture, SerialNumber, SMBIOSBIOSVersion

Caption              OSArchitecture      SerialNumber        SMBIOSBIOSVersion
-------              --------------      ------------        -----------------
Microsoft Windows... 64-bit              00426-069-126489... MBA41.88Z.0077.B...
```

When you send $os to Select-Object and specify “*”, you get a list of all of the combined properties:

```
$os | Select-Object -Property *
```

## 58. Removing Empty Object Properties

Ever wanted to remove empty object properties? This example illustrates how it can be done:

```
$bios = Get-WmiObject -Class Win32_BIOS

$biosNew = $bios |
Get-Member -MemberType *Property |
  Select-Object -ExpandProperty Name |
  Sort-Object |
  ForEach-Object -Begin { $obj=New-Object PSObject } {
    if ($bios.$_ -eq $null)
    {
      Write-Warning “Removing empty property $_”
    }
    else
    {
      $obj | Add-Member -memberType NoteProperty -Name $_ -Value $bios.$_
    }
  }{$obj}

$biosNew
```

The result is a BIOS object where all empty properties are removed. In addition, all properties are now alphabetically sorted.

This is a good thing if you want to use the object for reporting purposes, for example.

## 59. Consolidating Multiple Information in One Object

In the next example, PowerShell retrieves information from two different WMI classes. It creates a custom object by expanding the simple data object “1” by two new properties. Then, it saves the WMI information into those properties and returns the object.

```
$result = 1 | Select-Object Version, Build

$result.Version = (Get-WmiObject Win32_BIOS).Version
$result.Build = (Get-WmiObject Win32_OperatingSystem).BuildNumber

$result
```

Note that you can sort and group and measure this object in just the same way as any other object.

## 60. Working Remotely with WMI

WMI natively is able to work on local or remote machines. Simply use the -ComputerName parameter to access remote systems. You can provide an IP address, a NetBIOS name and even a comma-separated list of multiple names or a list of computer names that you read in from file using Get-Content.

This line will read BIOS information from your local machine and another one called “storage1”:

```
Get-WmiObject Win32_BIOS -ComputerName ‘localhost’,’storage1’
```

If you receive a “RPC server not available” exception, then the system is blocked by a firewall or not online. If you receive an “access denied” exception, then you do not have local admin rights on the target machine.

If you query multiple remote systems, you need to know where the results came from. The computer name can be found in the __SERVER property available in any WMI result:

```
Get-WmiObject Win32_BIOS -ComputerName ‘localhost’,’storage1’ |
  Select-Object -Property __SERVER, Manufacturer, SerialNumber
```

You can authenticate yourself with different credentials:

```
Get-WmiObject Win32_BIOS -ComputerName storage1  -Credential Administrator
```

## 61. Unattended WMI: Exporting and Importing Credentials

You may have to use alternate credentials when you connect to different systems. You’ll find that a lot of cmdlets provide the -Credential parameter which accepts a credential object.

Typically, you would create such an object by using Get-Credential and then interactively entering user name and password.

Use these two functions if you’d like to save your credential to file and re-use this saved version for unattended scripts:

```
Function Export-Credential
{
  param
  (
    $Credential,

    $Path
  )

  $Credential = $Credential | Select-Object *
  $Credential.Password = $Credential.Password | ConvertFrom-SecureString
  $Credential | Export-Clixml $Path
}

Function Import-Credential
{
  param
  (
    $Path
  )

  $Credential = Import-Clixml $Path
  $Credential.Password = $Credential.Password | ConvertTo-SecureString
  New-Object System.Management.Automation.PSCredential($Credential.UserName, $Credential.Password)
}
```

Export-Credential can save a credential to XML:

```
Export-Credential (Get-Credential) $env:temp\cred.xml
```

Whenever you need to use this credential, you should import it:

```
$cred = Import-Credential -Path $env:temp\cred.xml
Get-WmiObject Win32_BIOS -ComputerName storage1 -Credential $cred
```

Note that your password is encrypted and can only be imported by the same user that exported it.

## 62. HTML Reporting: Create a System Report

In a previous tip you learned how WMI can deliver system profile information. Let’s take this data and turn it into a HTML report. Here’s a function Get-SystemReport that creates the HTML for local and/or remote systems (provided you have remote access permissions):

```
Function Get-SystemReport
{
  param
  (
    $ComputerName = $env:ComputerName
  )

  $htmlStart = “
  <HTML><HEAD><TITLE>Server Report</TITLE>
  <style>
  body { background-color:#EEEEEE; }
  body,table,td,th { font-family:Tahoma; color:Black; Font-Size:10pt }
  th { font-weight:bold; background-color:#AAAAAA; }
  td { background-color:white; }
  </style></HEAD><BODY>
  <h2>Report listing for System $Computername</h2>
  <p>Generated $(Get-Date -Format ‘yyyy-MM-dd hh:mm’) </p>
  “

  $htmlEnd = ‘</body></html>’

  $htmlStart
  Get-WmiObject -Class CIM_PhysicalElement |
  Group-Object -Property __Class |
  ForEach-Object {
    $_.Group |
    Select-Object -Property * |
    ConvertTo-Html -Fragment -PreContent (‘<h3>{0}</h3>’ -f $_.Name )
  }
  $htmlEnd
}
```

And this is how you would call the function and create a report:

```
$path = “$env:temp\report.hta”
Get-SystemReport | Out-File -Filepath $path
Invoke-Item $path
```

## 63. Deleting Shares

If you’d like to get rid of a file share, this is how you can do it:

```
$share = Get-WmiObject Win32_Share -Filter “Name=’$sharename’”
$share.Delete()
```

As with share creation, you need admin rights to delete a share.

## 64. Finding CD-ROM Drives

You can find out whether a system has CD-ROM drives by using a little function that returns all drive letters from all CD-ROM drives in your system. To get the list of CD-ROM drives, simply call the function:

```
Function Get-CDDrive
{
  Get-WmiObject Win32_LogicalDisk -Filter ‘DriveType=5’ |
  Select-Object -ExpandProperty DeviceID
}

Get-CDDrive
```

To find out how many CD-ROM drives are available, you should use the Count property:

```
(Get-CDDrives).Count
```

If you want to exclude CD-ROM drives with no media inserted, check the Access property. It is 0 for no media, 1 for read access, 2 for write access and 3 for both:

```
Get-WmiObject Win32_LogicalDisk -Filter ‘DriveType=5 and Access>0’ |
  Measure-Object | Select-Object -ExpandProperty Count
```

## 65. Turning Multi-Value WMI Properties into Text

When you read multi-valued information from WMI or any other source, for example, network adapter IP addresses, this information is returned as array:

```
Get-WmiObject -Class Win32_NetworkAdapterConfiguration -Filter ‘IPEnabled=true’ |
  Select-Object -ExpandProperty IPAddress
```

Most likely, this will return IPv4 and IPv6 addresses.

If you want to turn this into a list, use the -join operator:

```
$array = Get-WmiObject -Class Win32_NetworkAdapterConfiguration -Filter ‘IPEnabled=true’ |
  Select-Object -ExpandProperty IPAddress

$array -join ‘, ‘
```

## 66. Finding Network Adapter Data Based on Connection Name

Sometimes it would be nice to be able to access network adapter configuration based on the name of that adapter as it appears in your Network and Sharing Center. To find the network configuration data for any network card with a “LAN” in its name, use this code:

```
# finding LAN connections
Get-WmiObject Win32_NetworkAdapter -Filter ‘NetConnectionID like “%LAN%”’ |
ForEach-Object { $_.GetRelated(‘Win32_NetworkAdapterConfiguration’) }

# finding WiFi connections
Get-WmiObject Win32_NetworkAdapter -Filter ‘NetConnectionID like “%Wireless%”’ |
ForEach-Object { $_.GetRelated(‘Win32_NetworkAdapterConfiguration’) }
```

If you want to further narrow this down and cover only NICs that are currently connected to the network, extend the WMI filter. This finds all active WiFi connections (unless they have been renamed and use a non-default NetConnectionID):

```
Get-WmiObject Win32_NetworkAdapter -Filter ‘(NetConnectionID like “%Wireless%”) and
(NetConnectionStatus=2)’ |
  ForEach-Object { $_.GetRelated(‘Win32_NetworkAdapterConfiguration’) }
```

## 67. Finding and Deleting Orphaned Shares

Maybe you never noticed but when you delete folders that were shared on the network, the share may be left behind. To locate shares that have no folder anymore, use WMI and the Win32_Share class to give you the folder path that the share is pointing toward. Next, use Test-Path to see if the path still exists:

```
Get-WmiObject Win32_Share | Where-Object { $_.Path -ne '' }  |
  Where-Object { $_.Path -like '?:\' } |
  Where-Object { -not (Test-Path $_.Path) }
```

Your system is in good shape if this line does not return any results. Orphaned shares can occur when a shared folder is deleted in a low-level way, or when you disconnect hard drives.

## 68. Changing Computer Description

Only a few properties in WMI objects are actually writeable although Get-Member insists they are all "Get/Set":

```
Get-WmiObject -Class Win32_OperatingSystem | Get-Member -MemberType Properties
```

That's because WMI only provides "copies" of information to you, and you can do whatever you want. WMI won't care. To make WMI care and put property changes into reality, you need to send back your copy to WMI by calling Put() method.
This will change the description of your operating system (provided you run PowerShell with full Administrator privileges):

```
$os = Get-WmiObject -Class Win32_OperatingSystem
$os.Description = 'I changed this!'
$result = $os.PSBase.Put()
```

If you change a property that in reality is not writeable, you'll get an error message only when you try to write it back to WMI. The truly changeable WMI properties need to be retrieved from WMI:

```
$class = [wmiclass]'Win32_OperatingSystem'
$class.Properties |
  Where-Object { $_.Qualifiers.Name -contains 'write' } |
  Select-Object -Property Name, Type
```

## 69. Using Get-CimInstance instead Get-WmiObject

In PowerShell 3.0, while you still can use the powerful Get-WmiObject cmdlet, it is slowly becoming replaced by the family of CIM cmdlets.

If you use Get-WmiObject to query for data, you can easily switch to Get-CimInstance. Both work very similar. The results from Get-CimIn-stance, however, do not contain any methods anymore.

If you must ensure backwards compatibility, on the other hand, you may want to avoid CIM cmdlets. They require PowerShell 3.0 and won't run on Windows XP and Vista/Server 2003.

## 70. New DateTime Support in CIM Cmdlets

In PowerShell 3.0, to work with WMI you can still use the old WMI cmdlets like Get-WmiObject. There is a new set of CIM cmdlets, though, that pretty much does the same - but better.

For example, CIM cmdlets return true DateTime objects. WMI cmdlets returned the raw WMI DateTime format. Compare the different results:

```
Get-CimInstance -ClassName Win32_OperatingSystem |
  Select-Object -ExpandProperty LastBoot*

Get-WmiObject -ClassName Win32_OperatingSystem |
  Select-Object -ExpandProperty LastBoot*
```

## 71. Using CIM Cmdlets Against PowerShell 2.0

By default, you cannot remotely access a system that has no WinRM capabilities. With a little trick, however, you can convince CIM cmdlets to fall back and use the old DCOM protocol instead. Here is an example:

```
Get-CimInstance -ClassName Win32_BIOS
Get-CimInstance -ClassName Win32_BIOS -ComputerName storage1
```

This will first get BIOS information from the local computer. Next, it tries and read the information from a remote system called “storage1”. This may fail if that system won’t support WinRM, for example, because it might be some old server 2003 with PowerShell 2.0 on it. In this case, you’d get an ugly red exception message, complaining that some DMTF resource URI wasn’t found.

So next are the lines you can use to force Get-CimInstance to use the old DCOM technology:

```
$option = New-CimSessionOption -Protocol DCOM
$session = New-CimSession -ComputerName storage1 -SessionOption $option

Get-CimInstance -ClassName Win32_BIOS -CimSession $session
```

This time, the server could be contacted and returns the data.

## 72. Mixing DCOM and WSMan in WMI Queries

Using the new CIM cmdlets in PowerShell 3.0, you can run remote WMI queries against multiple computers using multiple remoting protocols.

The sample code below gets WMI BIOS information from five remote machines. It uses DCOM for the old machines in $OldMachines and uses WSMan for new machines in $NewMachines. Get-CimInstance gets the BIOS information from all five machines (make sure you adjust the computer names in both lists so that they match real computers in your environment, and that you have admin privileges on all of these machines):

```
# list of machines with no WinRM 3.0/no PowerShell 3.0
$OldMachines = ‘pc_winxp’, ‘pc_win7’, ‘server_win2003’

# list of new machines with WinRM 3.0 (use Test-WSMan to find out)
$NewMachines = ‘pc_win8’, ‘server_win2012’

$useDCOM = New-CimSessionOption -Protocol DCOM
$useWSMan = New-CimSessionOption -Protocol WSMan

$Session = New-CimSession -ComputerName $OldMachines -SessionOption $useDCOM
$Session += New-CimSession -ComputerName $NewMachines -SessionOption $useWSMan

# get WMI info from all machines, using appropriate protocol
Get-CimInstance -CimSession $Session -ClassName Win32_BIOS
```

## 73. Calling WMI Methods with CIM Cmdlets

It can be very useful to call WMI methods, for example to create new shares, but in PowerShell 2.0 you had to know the names and exact order of arguments to submit which was a bit like black magic:

```
$rv = Invoke-WmiMethod -Path ‘Win32_Share’ -ComputerName $ComputerName -Name Create -ArgumentList
$null, $Description, $MaximumAllowed, $Name, $null, $Path, $Type
```

In PowerShell 3.0, you can use Get-CimClass to discover methods and arguments, and use Invoke-CimMethod instead. It takes the arguments as a hash table, so order is no longer important:

```
$class = Get-CimClass -ClassName Win32_Share
$class.CimClassMethods
```

The result looks similar to this:

```
PS> $class.CimClassMethods

Name                          ReturnType Parameters          Qualifiers
----                          ---------- ----------          ----------
Create                            UInt32 {Access, Descrip... {Constructor, Im...
SetShareInfo                      UInt32 {Access, Descrip... {Implemented, Ma...
GetAccessMask                     UInt32 {}                  {Implemented, Ma...
Delete                            UInt32 {}                  {Destructor, Imp...
```

Next, pick a method--like "Create" in this example--and get detail information:

```
PS> $class.CimClassMethods['Create']

Name                          ReturnType Parameters          Qualifiers
----                          ---------- ----------          ----------
Create                            UInt32 {Access, Descrip... {Constructor, Im...



PS>
PS> $class.CimClassMethods['Create'].Parameters

Name                             CimType Qualifiers         ReferenceClassName
----                             ------- ----------          ------------------
Access                          Instance {EmbeddedInstanc...
Description                       String {ID, In, Mapping...
MaximumAllowed                    UInt32 {ID, In, Mapping...
Name                              String {ID, In, Mapping...
Password                          String {ID, In, Mapping...
Path                              String {ID, In, Mapping...
Type                              UInt32 {ID, In, Mapping...
```

Note that some of the information is not displayed. Send the information to Out-GridView to see everything.

From this, you can now construct the command that would create new shares:

```
$args = @{
 Name='myTestshare'
 Path='c:\'
 MaximumAllowed=[UInt32]4
 Type=[UInt32]0
}

Invoke-CimMethod -ClassName Win32_Share -MethodName Create -Arguments $args
```

Note that you may need Administrator privileges. A return code of "0" indicates success. A return code of "2" reports "Access Denied", and a code of "22" indicates that there is already a share under the name you picked. The meaning of these error codes is specific to the WMI class and method you use.

## 74. Creating "Mini Modules"

Did you know that every PowerShell function can be turned into a script module with just one line of code? This way, you can easily turn your functions into "mini modules", and your new "cmdlets" will then be available in any PowerShell environment you open.

To test drive this, open the ISE editor and create a function:

```
function Get-BIOS
{
  param($ComputerName, $Credential)

  Get-WmiObject -Class Win32_BIOS @PSBoundParameters
}
```

This Get-BIOS function will get the computer BIOS information and supports -ComputerName and -Credential for remote access, too. Make sure you run the function and test it.

Next, turn the function into a module:

```
$name = ‘Get-BIOS’
New-Item -Path $home\Documents\WindowsPowerShell\Modules\$name\$name.psm1 -ItemType File -Force
-Value “function $name { $((Get-Item function:\$name).Definition) }”
```

Just make sure $name contains the name of your function, and you ran your function so it is in memory.

Now why is this conversion important? Because PowerShell 3.0 auto-detects functions in modules! So if you use PowerShell 3.0 and have created the “mini module”, open up a new PowerShell and type:

```
PS> Get-BIOS


SMBIOSBIOSVersion : MBA41.88Z.0077.B0E.1110141154
Manufacturer      : Apple Inc.
Name              : Default System BIOS
SerialNumber      : C02GW04RDRQ4
Version           : APPLE  - 60
```

Bam! Your function is now available automatically, and you can easily add new functionality.