PowerShell and WMI Covers 150 practical techniques 读书笔记

Google for powershell tutorial

By Richard Siddaway

Part 1 Tools of the trade

1 Solving administrative challenges

1.1 Administrative challenges
Hardware costs is decreasing,
while Infrastructure complexity and Ad ministration costs are increasing.

1.1.1 Too many machines

1.1.2 Too many changes

1.1.3 Complexity and Understanding

1.2 Automation: the way forward

1.3 PowerShell overview

1.3.1 PowerShell scope

1.3.2 PowerShell and .NET
PowerShell is based on .NET and can access most of the .NET Framework.
eg:
```
Get-Service wi* | Format-Table -AutoSize
```

1.3.3 Breaking the curve

1.4 WMI overview

1.4.1 What is WMI?
Windows Management Instrumentation.
automatically installed with Windows.
eg: to look at the WMI classes available for working with disks
```
Get-WmiObject -List *disk* | sort name | select name
```
Note: the CIM_class is the parent, corresponding to the definition supplied by the DMTF; the Win_32 classes are child classes that Microsoft has implemented.

1.4.2 Is WMI really too hard?

1.5 Automation with WMI and PowerShell
eg: VBScript to retrieve process information
```
set objWMIService = GetObject("winmgmts:" & "{impersonationlevel=impersonate}!\\" & ".\root\cimv2")
set colProcesses = objWMIService.ExecQuery ("SELECT * FROM Win32_Process")
for each objProcess in colProcesses
    WScript.Echo " "
    WScript.Echo "Process Name : " + objProcess.Name
    WScript.Echo "Handle   : " + objProcess.Handle
    WScript.Echo "Total Handles : " + Cstr(objProcess.HandleCount)
    WScript.Echo "ThreadCount : " + Cstr(objProcess.ThreadCount)
    WScript.Echo "Path"  : " + objProcess.ExecutablePath
```

next
Note: The script starts by creating an object, objWMIService, to enable interrogation of the WMI service. A list of active processes is retrieved by running a WQL querry. The collection of processes is iterated through, and you write a caption and the value of a particular property to the screen.

eg: PowerShell translation
```
$procs = Get-WmiObject Query "SELECT * FROM Win32_Process"
foreach ($proc in $procs) {
    Write-Host "Name   :" $proc.ProcessName
    Write-Host "Handle   :" $proc.Handle
    Write-Host "Total Handles :" $proc.Handles
    Write-Host "ThreadCount  :" $proc.ThreadCount
    Write-Host "Path    :" $proc.ExecutablePath
}
```
Note: You first run the WMI query to select the information you need and put the results into a variable. The variable is a collection of objects reresenting the different processes. You can then loop through the collection of processes (using the foreach command), and for each process in that collection use the Write-Host cmdlet to output a caption and the value of the properties you're interested in.

eg: PowerShell command in a single line
```
Get-WmiObject Win32_Process | Format-Table ProcessName, Handle, Handles, ThreadCount, ExecutablePath -AutoSize
```
Note: This final version uses the Get-WmiObject cmdlet directly. Get-WmiObject returns an object for each process, and you use the PowerShell pipeline to pass them into a Format-Table cmdlet. This combines the data selection and display functionality and roduces neatly formatted tabular outut. (to display the output in a list format, substitute Format-List for Format-Table.)

1.6 Putting PowerShell and WMI to work
eg1: Shutting down all the Windows machines in your data center.

eg2: auditing a large number of machines to discover their capabilities.

1.6.1 Example 1: Shutting down a data center
eg: Shut down a data center
```
Import-Csv computers.csv |
foreach {
    (Get-WmiObject -Class Win32_Operating System -ComputerName $_.Computer ).Win32Shutdown(5)
}
```
Note: This script uses a CSV file called computers.csv, which contains a list of computer names.
eg :
```
Computer
W08R2CS01
W08R2CS02
W08R2SQL08
W08R2SQL08A
WSS08
DC02
```
Note: The $_ symbol refers to the current object on the pipeline, and the Computer part comes from the CSV header.

1.6.2 Example 2: Auditing hundreds of machines
The audit should return the following information:
Server make and model
CPU data (numbers, cores, logical processors, and speed)
Memory
Windows version and service pack level
eg: to gather basic information from many machines.
```
Import-Csv computers.csv |
foreach {
     $system = " " | select Name, Make, Model, CPUs, Cores, LogProc, Speed, Memory, Windows, SP
     $server = Get-WmiObject -Class Win32_ComputerSystem  -ComputerName $_.Computer
```

```
    $system.Name = $server.Name
    $system.Make = $server.Manufacturer
    $system.Model = $server.Model
    $system.Memory = $server.TotalPhysicalMemory
    $system.CPUs = $server.NumberOfProcessors
    $cpu = Get-WmiObject -Class Win32_Processor  -ComputerName $_.Computer | select -First 1
    $system.Speed = $cpu.MaxClockSpeed
    $os = Get-WmiObject -Class Win32_OperatingSystem  -ComputerName $_.Computer
    $system.Windows = $os.Caption
    $system.SP = $os.ServicePackMajorVersion
    if (($os.Version -split "\.")[0] -ge 6) {
        $system.Cores = $cpu.NumberOfCores
        $system.LogProc = $cpu.NumberOfLogicalProcessors
    }
    else {
        $system.CPUs = ""
        $system.Cores = $server.NumberOfProcessors
    }
    $system
} |
Format-Table -AutoSize -Wrap
```

1.7 Summary

2 Using PowerShell

PowerShell allows you to read from and write to files using the *-Csv, *-Content, and Out-File cmdlets.

2.1 PowerShell in a nutshell

What's special about PowerShell?
A shell
A set of command-line tools (cmdlets)
A scripting language
An automation engine that allows for remote access, asynchronous processing, and integration between products.

PowerShell allows you to do things such as:
Run PowerShell commands
Run the standard Windows utilities, such as ipconfig or ping
Work with the filesystem using standard commands
Run Windows batch files (with some provisos around environmental variables)
Run VBScripts

2.2 Cmdlets

A cmdlet name consists of two words separated by a hyphen, such as Get-Process.
The first part of the name is a verb, and the second part is a noun.
eg:
Get-Verb : a built-in function to discover the standard set of verbs in PowerShell.
Get-Command: to fetch a list of the cmdlets available in PowerShell.
eg: groups the results by the verb in the cmdlet name and sorts the verbs on the number of occurrence. The top 20 verbs are then displayed.

```
Get-Command -CommandType cmdlet | group verb | sort count -Descending | select name -First 20
```

2.2.1 Utility cmdlets

Utility cmdlets function as the glue that binds the working cmdlets together on the pipeline.
They enable you to fileter, sort, compare, and group data or even create new objects.
eg:
Compare-Object: (compare, diff) compares two sets of objects.

ForEach-Object: (foreach, %) performs an operation against each member of a set of input objects.
Group-Object: (group) groups objects that contain the same value for specified properties.
Measure-Object: (measure) calculates the numeric properties of objects, and the characters, words, and lines in string objects, such as files of text.
New-Object: creates an instance of a Microsoft .NET Framework or COM object.
Select-Object: (select) selects specified properties of an object or set of objects. It can also select unique objects from an array of objects, or it can select a specified number of objects from the beginning or end of an array of objects.
Sort-Object: (sort) sorts objects by property values.
Tee-Object: (tee) saves command output in a file or variable and displays it in the console.
Where-Object: (where, ?) creates a filter that controls which objects will be passed along a command pipeline.

2.2.2 Where-Object
eg:
Get-WmiObject -Class Win32_Service
eg:
Get-WmiObject -Class Win32_Service | select name, startmode, state
eg:
Get-WmiObject -Class Win32_Service | select name, startmode, state | where {$_.state -eq "stopped"}
or
Get-WmiObject -Class Win32_Service | where {$_.state -eq "stopped"} | select name, startmode, state
Note: When working against hundreds of machines, filtering as early as possible could improve performance.

eg: to display the status of services set to start automatically on a particular computer
$computername = "."
Get-WmiObject -Class Win32_Service -ComputerName $computername |
where {$_.state -eq "stopped" -and $_.startmode -eq "auto"} |
select name, startmode, state
Note: The first line defines a variable to hold the computer name. a period, "." signifies the local machine. (localhost or $env:COMPUTERNAME can also be used to denote the local system.)

2.2.3 Foreach-Object
eg:
$computername = "."
Get-WmiObject -Class Win32_Service -ComputerName $computername |
where {$_.state -eq "stopped" -and $_.startmode -eq "auto"} |
foreach { $_.StartService() }
Note: The select statement has been dropped, WMI will provide a return code of 0 if the action was successful.

2.2.4 Aliases
eg: A list of currently defined aliases can be obtained by using Get-Alias.
Get-Alias | where {$_.definition -like "*object"} | Format-Table Name, Definition -AutoSize
eg:
Get-Command *alias | select name
Note: The import and export commands are for reading and writing the alias information to a file so you can reuse it in other PowerShell sessions. Unless you do this, or you set the alias in your profile, it's lost when you close PowerShell.
eg:
Get-Help about_profiles: to get more details about profiles.
eg: to create an alias
New-Alias -Name filter -Value Where-Object
Set-Alias -Name sieve -Value Where-Object
eg: to confirm the creation with the code snippet you used earlier to list the aliases of the utility cmdlets.
Get-Alias | filter {$_.definition -like "*object"} | Format-table Name, Definition -AutoSize
Get-Alias | sieve {$_.definition -like "*object"} | Format-table Name, Definition -AutoSize
Note:
powershell.exe -noprofile : to test your snippetif you have defined the alias in your profile.
Note: aliases are fine at the command line, but don't publish code using them and don't include them in scripts.
eg: to display your two aliases.
"filter", "sieve" | foreach {dir alias:\$_} : this will

eg: to delete your two aliases.

```
"filter", "sieve" | foreach {Remove-Item alias:\$_}
```

Note: The list of installed PowerShell drives can be found using Get-PSDrive.

eg:

```
Get-Help about_Providers
Get-Help Get-PSdrive
```

2.3 Pipeline

The big difference is that DOS and UNIX commands produce (emit) text, whereas PowerShell cmdlets emit .NET objects.

2.4 .NET for administrators

PowerShell uses .NET objects.

2.4.1 Objects

An object is a package that contains both data and the information on how to use that data.

2.4.2 PowerShell objects

PowerShell puts a wrapper around .NET objects. Sometimes properties and methods can be added or removed. The pure .NET object can be accessed by usign a .psbase suffix.

2.4.3 Creating .NET objects

eg: to generate some (pseudo) random numbers.

```
$rand = New-Object -TypeName System.Random -ArgumentList 42
$rand.Next()
```

Note: PowerShell v2 has its own cmdlet for generating random number, Get-Random, but this is a nice simple example of using a .NET object.

2.4.4 Creating your own objects

Three ways to create new objects that you've designed:

1) select method

eg:

```
$myobject = " " | select name, number, description
$myobject.Name = "Object1"
$myobject.Number = 100
$myobject.Description = "Simplest object creation"
```

Note: create an object called myobject that has three properties.

The drawback to this method is that the resultant object is a string, as are all of its properties. Technically, it's a Select.System.String.

2) New-Object and the PSObject class

eg:

```
$myobject = New-Object System.Management.Automation.PSObject |
Add-Member -MemberType NoteProperty -Name "Name" -Value "object2" -PassThru |
Add-Member -MemberType NoteProperty -Name "Number" -Value 100 -PassThru |
Add-Member -MemberType NoteProperty -Name "Description" -Value "More complicated" -PassThru
```

eg: to simplify the previous snippet.

```
$myobject = NewObject PSObject -Property @{
    Name = "object2a"
    Number = 100
    Description = "More complicated"
}
```

Note: The Property parameter is used. It's a hash table of property names and values that are applied to the object as it's created. The drawback it that the order of the parameters on the object can't be guaranteed.

3) using c#.

eg:

```
$source = @"
public class pawobject
```

```
{
    public string Description { get; set;}
    public string Name {get; set;}
    public int Number {get; set}
}
"@
Add-Type $source -Language CSharpVersion3
$myobject = New-Object -TypeName pawobject -Property @{
    Name = "myobject3";
    Number = 200;
    Description = "More complicated again"
    }
```

Note: a here-string (a multilined string that can be used to embed chunks of text into your script) starts with @" and ends with "@. Add-Type is used to compile the class. The main advantage of using this method is that the properties become strongly typed.

2.5 PowerShell scripting language

The PowerShell consists of four things:

a shell

the command-line tools

a scripting language

an automation engine

PowerShell language is case insensitive.

2.5.1 Loops

foreach

eg:

```
$lower = "a","b","c","d"
foreach ($letter in $lower) { Write-Host $letter.ToUpper() }
```

Note: An array (collection) of letters is defined. For each leter in the collection, you convert the letter to uppercase and use Write-Host to write it tot the screen.

eg:

```
"a","b","c","d" | foreach { Write-Host $_.ToUpper() }
```

Note: foreach in this example is the alias for ForEach-Object cmdlet. You don't need the ($letter in $lower) structure because it's implicit in the use of the pipeline. $_ represents the current object on the pipeline.

If foreach is on the pipeline, it' s an alias for the ForEach-Object cmdlet. If foreach is followed by a ($letter in $lower) type of structure, it' s a language statement.

for loop

eg: to convert $i into a character (using ASCII codes) and write it out.

```
for ($i=65; $i -le 68; $i++) { Write-Host $([char] $i) }
```

eg:

```
65..68 | foreach { Write-Host $([char] $_) }
```

Note: The ... or range operator is used to define a range of numbers.

other loops

while loop

syntax:

```
while (<condition>) {
    < PowerShell code >
}
```

do looop

syntax:

```
do{
    <PowerShell code >
} until (<condition>)
```

NOTE:

Get-Help about-While

Get-Help about-Do

2.5.2 Branching

if statement

eg:

```
$date = Get-Date
if ($date.DayOfWeek -eq "Friday") {
    "The weekend starts tonight"}
elseif ($date.DayOfWeek -eq "Saturday" -or $date.DayOfWek -eq "Sunday") { "It's the weekend" }
else { "Still working!" }
```

Note: If you have a string all by itself on a line, PowerShell will treat it as something you want to output and will write it to the screen or whatever output mechanism you've define.

switch statement

eg:

```
switch (Get-Date).DayOfWeek) {
    "Sunday" {It's the weekend but work tomorrow"; break}
    "Monday" {"Back to work"; break}
     “Tuesday" {"Long time until Friday"; break}
    "Wednesdya" {"Half way through the week"; break}
    "Thursday" {"Friday tomorrow"; break}
    "Friday" {"It's the weekend tomorrow"; break}
    "Saturday" {"It's the weekend"; break}
    default {"Something's gone wrong"}
}
```

2.5.3 Input and output

Three main areas:

Writing to the screen and accepting data typed in response to a prompt

Writing to or reading from a file on disk.

Writing to or reading from a specialized data store, such as Active Directory, the registry, or SQL Server.

Note: XML is well supported in PowerShell.

eg:

```
Get-Help xml
```

Common cmdlets that provide Input and output functionality

```
Write-Host
Read-Host
Out-Host
Out-GridView
Write-Output
```

2.6 Finding help

2.6.1 Get-Help

eg: to view the help function definition

```
(Get-Item -Path function:\help).Definition
```

eg:

```
Get-Help @PSBoundParameters | more
```

eg:

```
Get-Help Get-Command
```

eg: to get the full information on using the help system

```
Get-Help Get-Help -Full
```

eg: to get help online (This will open your default Internet Browser to get the help info about the cmdlet)

```
Get-Help Get-Command -Online
```

eg: to get help files that give conceptual information about PowerShell.

```
Get-Help about*
```

eg: to update the help information

```
Update-Help
```

2.6.2 Get-Command

Get-Command can look beyond PowerShell.

eg:

Get-Command ipconfig | f1 *

Note:

Get-Help will tell you how to use a particular command, but Get-Command will discover what commands are available.

eg:

Get-Command *wmi*

eg:

Get-Command *wmi* -CommandType cmdlet | select name

eg: The other use for Get-Command is finding the cmdlets that are loaded by a particular PowerShell snap-in or module.

Get-Command -Module BitsTransfer

eg:

Get-Command -Syntax

eg:

Get-Command -Noun wmi*

eg:

Get-Command -Verb get

eg:

Get-Command -CommandType alias

Get-Command -CommandType application

Get-Command -CommandType cmdlet

Get-Command -CommandType externalscript

Get-Command -CommandType function

Get-Command -CommandType scripts

2.6.3 Get-Member

eg: return the process's type to investigate into the cmdlet.

Get-Process powershell | Get-Member

2.6.4 PowerShell community

PowerShell MVPs

2.7 Code reuse

eg: to get the PowerShell execution policy

Get-ExecutionPolicy

eg: to change the execution policy into RemoteSigned

Set-ExecutionPolicy -ExecutionPolicy RemoteSigned

Note: In remote-signed state, PowerShell allows scripts to run from the local drive but expects scripts on remote drives to be signed with a recognized code-signing certificate.

Note: Strongly not recommended to use the Unrestricted execution policy setting.

advanced functions

PowerShell modules

2.7.1 Editors

Notepad

PowerShell ISE

PowerGUI Script Editor

PowerShell Plus

2.7.2 Scripts

eg: Script to investigate physical disks

```
param (
    [string] $computername = "localhost"
)
    Get-WmiObject -Class Win32_DiskDrive -ComputerName $computername |
```

```
    Format-List DeviceID, Status,
    Index, InterfaceType,
    Partitions, BytesPerSector, SectorsPerTrack, TrackPerCylinder,
    TotalHeads, TotalCylinders, TotalTracks, TotalSectors,
    @{Name="Disk Size (GB)"; Expression={"{0:F3}" -f $($_.Size/1GB)}}
```

Note: In many WMI scripts, a period (.) is substituted for localhost, both refer to the local machine. However, in a very few instances, these values can cause issues, so use $env:COMPUTERNAME instead.

eg: If you save this script into a file called Get-DiskInfo.ps1, The script is run by typing the following:

```
.\Get-DiskInfo.ps1
```

eg:

```
.\Get-DiskInfo.ps1 -computername "rslaptop01"
```

2.7.3 Functions

eg: Function to investigate physical disks

```
function get-disk {
param (
    [string] $computername = "$env:COMPUTERNAME"
)
    Get-WmiObject -Class Win32_DiskDrive -ComputerName $computername |
    Format-List DeviceID, Status,
    Index, InterfaceType,
    Partitions, BytesPerSector, SectorsPerTrack, TracksPerCylinder,
    TotalHeads, TotalCylinders, TotalTracks, TotalSectors,
    @{Name="Disk Size (GB)"; Expression={"{0:F3}" -f $($_.Size/1GB)}}
}
Import-Csv computers2.csv | foreach {get-disk $_.computer}
```

or

```
. .\Listing2.3a.ps1
```

Note: The first dot tells PowerShell to keep any functions or variables in memory, rather than discarding them after execution. This is referred to as dot sourcing.

eg: for more information about function

```
get-help *function*
```

eg: Advanced function to investigate physical disks

```
function get-disk {
pCmdletBinding()]
param (
    [Parameter(ValueFromPipelineByPropertyName=$true)]
    [string]
    [ValidateNotNullOrEmpty()]
    $computername
)
PROCESS {
    Write-Debug $computername
    Get-WmiObject -Class Win32_DiskDrive -ComputerName $computername |
    Format-List DeviceID, Status,
    Index, InterfaceType,
    Partitions, BytesPerSector, SectorsPerTrack, TracksPerCylinder,
    TotalHeads, TotalCylinders, TotalTracks, TotalSectors,
    @{Name="Disk Size (GB)"; Expression={"0:F3}" -f $($_.Size/1GB)}}
}
```