The Administrator Crash Course

Windows PowerShell v2

Realtime publishers

Don Jones

Copyright Statement

PowerShell Crash Course Week 3

Hopefully, you're ready for week 3 of your crash course. The previous ten lessons have covered the majority of Windows PowerShell's core techniques and patterns. This week we're going to start learning about some of its cooler embedded technologies, building on those techniques and patterns.

Don't forget: I encourage you to continue exploring beyond this crash course, too. For example, visit http://windowsitpro.com/go/DonJonesPowerShell to find tips and tricks and FAQs and to ask questions, or drop by the PowerShell team's own blog at http://blogs.msdn.com/powershell for "insider" information. You'll also find a lot of in-person PowerShell instruction at events like TechMentor (http://www.techmentorevents.com) and Windows Connections (http://www.winconnections.com). If you want to get a few PowerShell tips per week, you're welcome to subscribe to my Twitter feed, @concentrateddon. I focus almost exclusively on PowerShell, and I'll pass along any tips or articles that I find especially useful. I've also put up a new resource to help you find all the *other* good PowerShell resources: http://shellhub.com. It's a hand-picked list of things guaranteed to be useful and educational.

How to Use this Crash Course

I suggest that you tackle a single crash course item each day. **Spend some time practicing whatever examples are provided and trying to complete tasks that make sense in your environment.** Don't be afraid to fail: Errors are how we learn. Just do it in a virtual environment (I recommend a virtualized domain controller running Windows Server 2008 R2) so that you don't upset the boss! Each "Day" in this crash course is designed to be reviewed in under an hour, so it's a perfect way to spend lunch for a few weeks. This book will be published in five-day increments, so each chapter corresponds to a single week of learning.

Week 3, Day 1: Remote Control

One of the coolest new features in Windows PowerShell v2 is its built-in remoting. With it, *every* command can be run on one or more remote computers—even simultaneously—without the commands' authors having to do any extra work to make it possible.

Pre-Requisites and Setup

First, you need to get remoting enabled. It requires that PowerShell v2 be installed on each and every computer that you want to remote to. Then run **Enable-PSRemoting** on those computers to turn on the ability to receive incoming connections. You can also control that through GPO: Run **help about_remote_troubleshooting** for details on how to set that up and for help on topics like connecting non-domain computers, remoting across a proxy server or firewall, and so on.

Authentication, by default, is done using Kerberos—so your password stays safe. By default, you'll remote using whatever credential was used to launch PowerShell—normally, it'll need to be an Administrator account. The service involved in remoting is Windows Remote Management (WinRM), and the protocol is Web Services for Management (WS-MAN), which rides atop HTTP. Enable-PSRemoting will set up the necessary exceptions in the Windows Firewall for you.

1:1 Remoting

If you just need to do something with one computer, then you can remote to it interactively—kind of like SSH in Unix. PowerShell does provide encryption, and you can have the entire connection run over HTTPS (rather than the default HTTP) for greater security. Just run **Enter-PSSession -computer *computername*** to start a session; you'll see the PowerShell prompt change appropriately, and you'll be running commands on the remote computer. When you're done, run **Exit-PSSession** to return to your local prompt and close the remote connection.

Enter-PSSession offers a number of options. Review the help and see if you can figure out how you would:

- Add SSL (HTTPS) to the connection
- Specify an alternate IP port (used if you've changed the port that WinRM listens to on the remote machine)
- Specify an alternate credential for the connection

1:Many Remoting

Managing one computer at a time can be useful, but it's not *automation.* What's often better is the ability to have the same command run on *multiple* computers, all at once. You do that with **Invoke-Command.** Here's a simple example using three computers:

```
Invoke-Command -scriptblock { Get-EventLog -Log Security -newest 100 }
  -computername server1,server2,server3
```

There's a lot of ways to get computer names into the -computername parameter. What I did is provide a comma-separated list, which PowerShell sees as an array of items. Another way is to read names from a text file, which contains one name per line:

```
... -computername (get-content names.txt)
```

The parentheses force the shell to execute the Get-Content command first, then feeds its output to the parameter. Parentheses are a useful trick, and there are several ways in which you can use them. Perhaps you have a CSV file that contains a column named Server? Here's how you'd use that column:

```
... -computername (import-csv servers.csv | select -expand Server)
```

The -expand parameter of Select-Object tells the command to grab the *contents* of the property designated—in this case, the Server column of your CSV, and feed just those values as the command output. That gets stuck onto the -computername parameter, and voila! You can use the same trick to query computers from Active Directory (AD), assuming you've loaded the ActiveDirectory module:

```
... -computername (get-adcomputer -filter *
  -searchbase "ou=servers,dc=mycompany,dc=local" | select -expand Name)
```

The -scriptblock parameter of Invoke-Command is where you put the commands you want to run remotely; multiple commands can be separated by a ";" semicolon. You can feed in an entire pipeline, and you should have as much processing done remotely as possible. For example, this:

```
Invoke-Command -scriptblock { ps | sort vm -desc | select -first 10 }
  -computername server1,server2,server3
```

Is better than this:

```
Invoke-Command -scriptblock { ps } -computername server1,server2,server3 |
  Sort vm -desc | select -first 10
```

The second example has far too much data being transmitted from the remote machines—and it won't produce the same results, anyway. See if you can figure out why (if not, drop by http://connect.concentratedtech.com and ask—I'll be happy to give you a clue. If you're registering for the first time, enter "Realtime" as the place where you took a class from me and I'll make sure your registration is approved).

By default, Invoke-Command talks to as many as 32 computers at once. Give it more than that, and the extras get queued up and completed in order. You can change that with the -Throttle parameter. Other parameters you should look for (read the help—no hints from me!):

- How to specify an alternate credential
- How to specify an alternate IP port
- How to enable SSL (HTTPS) for the connections
- How to specify a script file path instead of an individual command

Notice, too, that the results of Invoke-Command have an extra property attached. The PSComputerName property contains the name that each result came from so that you can keep track of them. You could sort and group that output by computer name once the output is back to your computer:

```
Invoke-Command -scriptblock { Dir c:\ } -computername (gc names.txt) |
  Sort pscomputername | format-table -groupby pscomputername
```

Also, keep in mind that you can only invoke commands *that are available on the remote computer.* Let's say all of the remote computers are Windows Server 2008 R2 machines, and you want to see what Best Practices Analyzer (BPA) models are available on each. To do that, each remote computer needs to load the BestPractices module:

```
Invoke-Command -scriptblock { Import-Module BestPractices; Get-BPAModel }
  -computername server1,server2,server3
```

That would load the module on each, then execute a command from that module. When the command finishes executing, the connection will close and the module will unload automatically.

Re-Using Connections

When you give a computer name (or computer names) to Invoke-Command or Enter-PSSession, the commands create a connection, then automatically tear it down when you're done with it. That means every time you open a new connection, you might have to specify credentials, alternate ports, and other options—which can be tedious.

Another option is to create a reusable connection, called a PSSession. When you do so, you'll probably want to save the session (or sessions) in a variable because that makes it easier to re-use it. For example:

```
$webservers = New-PSSession -computername www1,www2,www3
 -credential Administrator
```

When you're ready to use the session, just provide it to the -session parameter of Invoke-Command:

```
Invoke-Command -script { Get-Process } -session $webservers
```

Enter-PSSession only works with a single computer, so you can use an index number to specify which of those sessions you want. For example, the second session (www2) would be:

```
Enter-PSSession -session $webservers[1]
```

To close the sessions:

```
$webservers | Remove-PSSession
```

That also happens automatically when you close your shell. Leaving a session open does occupy some processor and memory overhead on both your machine and the remote one, but it's not a massive amount. Just don't get every admin in the habit of holding a bunch of sessions open, and you'll be fine.

Week 3, Day 2: In the Background

You may often have times when you want to kick off some long-running task, but you don't want to wait for it to complete. PowerShell provides background jobs as a way of moving a task into the background, where the shell will continue to execute and monitor it. The shell will also cache any results that the job generates, enabling you to receive those results once you're ready.

Starting a Job

There are three ways to start a job:

- If it's a local task that won't connect to remote computers, use the **Start-Job** cmdlet. Give it a -command parameter, which tells it what commands to run in the background:

```
Start-Job –command { Get-EventLog –log Security }
```

- Get-WmiObject has an –AsJob parameter, which forces the WMI processing into the background—especially useful when dealing with many remote computers.

```
Get-WmiObject -class Win32_BIOS -computer server1,server2 –asjob
```

- Invoke-Command also has an -AsJob parameter, which forces it to invoke commands in the background—again, useful when you're running against multiple remote computers:

```
Invoke-Command -scriptblock { Get-EventLog -log Security }
 -computername server1,server2,server3 -asjob
```

By default, you'll create a job with a generic name like "Job1." Both Start-Job and Invoke-Command provide parameters (-name and -jobname, respectively) that let you specify a custom job name—often making it easier to manage multiple jobs.

Managing Jobs

Run **Get-Job** to see a list of jobs and their states—Running, Completed, Failed, and so forth. Note that every job has an ID number, along with a name. You can get the status for a particular job by using that ID or name. I like to pipe the results to Format-List so that I can see all of the job's details:

```
Get-Job -name MyJob | Format-List -property *
```

A key thing to know is that the shell always creates a "parent" job, plus one "child" job for each computer. So a job created by using Start-Job will have a parent and one child—which is the local computer. An Invoke-Command job targeting four computers will have the parent job and four children. The Format-List trick will allow you to see the names of each child job—enabling you to get those jobs individually:

```
Get-Job -name Job2 | Format-List -property *
```

When you're done with a job, you can remove it to clean up the list:

```
Get-Job -name MyOldJob | Remove-Job
```

There's also Stop-Job, to kill a stuck job, and Wait-Job, which is useful when a script needs to start a job and then wait until it completes before running more commands.

Getting Results from Jobs

When a job is finished, it'll likely have some results for you. Here's how to get them:

```
Receive-Job -id 4
```

Or

```
Receive-Job -name MyDoneJob
```

By default, the received results are delivered to you and then no longer cached—meaning you can't receive them again. If you want to leave them cached in memory, add the -Keep parameter to Receive-Job.

Week 3, Day 3: Implicit Remoting

Microsoft has always had versioning challenges. For example, Windows Server 2008R2 ships with a new ActiveDirectory module for managing AD—but that module can only run on 2008R2 and Windows 7. What if you're stuck on XP? *Implicit remoting* is designed to help solve the problem.

Here's the scenario: You have a 2008R2 domain controller, which has the ActiveDirectory module already installed (it comes along with the domain controller role). You enable remoting (Enable-PSRemoting) on that domain controller, then settle down in front of your Windows XP machine that has PowerShell v2 installed. Run these commands, assuming DC2008 is the name of that domain controller:

```
$session = New-PSSession -computername DC2008
Invoke-Command -script { Import-Module ActiveDirectory } -session $session
Import-PSSession -session $session -module ActiveDirectory -prefix Rem
```

Now, all of the commands in the ActiveDirectory module are available as local commands on your Windows XP machine. A prefix, "Rem," has been added to each command's noun, helping remind you that the commands will actually *execute* remotely. You can even ask for help on them. For example, all of these will now run fine:

```
Get-RemADUser -filter *
Help New-RemADUser
```

Neat trick, right? The idea is that the server providing the functionality can now also provide the administrative capability, and you don't need to install every single admin tool locally on your computer. This also helps mitigate version mismatches by not caring what OS you're running on your workstation—tools simply run on the server.

Week 3, Day 4: Making a Simple Reusable Command

Let's say you've created a really kick-butt command that accomplished some great stuff. It's a lot of typing, but you love the way it works:

```
Get-WmiObject -class Win32_LogicalDisk -filter "DriveType=3"
 -computername server1,server2,server3 | Select-Object DeviceID,
 @{n='ComputerName';e={$_.__SERVER}},
 @{n='Size(GB)';e={$_.Size / 1GB -as [int]}},
 @{n='FreeSpace(MB)';e={$_.FreeSapce / 1MB -as [int] }}
```

You want to share that with some other techs in your organization, but you don't want them to have to retype all that—or even edit it because you *know* they'll mess up something. No problem. Put this into a text file with a .ps1 filename extension:

```
Function Get-DriveInventory {
 Param($computername)
 Get-WmiObject -class Win32_LogicalDisk -filter "DriveType=3"
  -computername $computername | Select-Object DeviceID,
  @{n='ComputerName';e={$_.__SERVER}},
  @{n='Size(GB)';e={$_.Size / 1GB -as [int]}},
  @{n='FreeSpace(MB)';e={$_.FreeSapce / 1MB -as [int] }}
}
```

The three lines in boldface are what I added, along with the boldfaced $computername in the command itself. Let's say you put this in a file named c:\scripts\utilities.ps1. Your other admin friends could then add the following to their PowerShell profile (run **help about_profiles** for more info):

```
. c:\scripts\utilities
```

That leading dot will load the contents of Utilities.ps1 into the shell's global scope, making your command available just like any other. To use it:

```
Get-DriveInventory -comp server-r2
```

The shell even supports abbreviating the -computername parameter, as shown here. Neat, right?

Week 3, Day 5: SELECTing

The example from the previous section utilized a little-known feature of the Select-Object cmdlet. This command has four really useful features:

- You can tell it what properties you want to keep for an object, helping to whittle down output to exactly what you want to see:

```
Get-Process | Select -property ID,Responding,VM,PM
```

- You can add custom properties, using a hashtable—just like you did with Format-Table in Chapter 2. The hashtable needs two keys: "N" is the name of the new property, and "E" is the expression used to create the new property's values:

```
Get-WmiObject -class Win32_LogicalDisk -filter "DriveType=3"
 -computername server1,server2,server3 | Select-Object -property DeviceID,
 @{n='ComputerName';e={$_.__SERVER}},
 @{n='Size(GB)';e={$_.Size / 1GB -as [int]}},
 @{n='FreeSpace(MB)';e={$_.FreeSapce / 1MB -as [int] }}
```

- You can also select a subset of the objects piped in—grabbing the first or last however many:

```
Get-Service | Select -first 10
```

- You can tell it to just grab the *values* from a given property. This forces the command to output simple values instead of a more complex object with one or more properties:

```
Get-ADComputer -filter * | Select -expand Name
```

Give these techniques a try with various other commands and see what happens.

Week 3 Wrap-Up

This week, you've learned about powerful embedded technologies within PowerShell—and you've also learned some new patterns. For example, the various ways in which I showed you to feed computer names to the -computername parameter of Invoke-Command are all patterns you'll find uses for with many other cmdlets. Pay attention to these different patterns, and try to think of ways in which you might use them elsewhere within the shell.

I'll leave you with a final super-complex example. Look through this carefully, and see if you can figure out why it does what it does. This is safe to run on a production system or your workstation, too, so that you can see the results. How would you modify this to run against a remote computer?

```
Get-WmiObject -class Win32_OperatingSystem | Select-Object
 @{n='OSVersion';e={$_.Caption}},
 @{n='SPVersion';e={$_.ServicePackMajorVersion}},
 @{n='OSBuild';e={$_.BuildNumber}},
 @{n='ComputerName';e={$_.__SERVER}},
 @{n='BIOSSerial';e={
  (Get-WmiObject -class Win32_BIOS -computername $_.__SERVER |
  Select -Expand SerialNumber)
 }} | Format-Table -property * -wrap
```

Good luck!

Download Additional Books from Realtime Nexus!

Realtime Nexus—The Digital Library provides world-class expert resources that IT professionals depend on to learn about the newest technologies. If you found this book to be informative, we encourage you to download more of our industry-leading technology books and video guides at Realtime Nexus. Please visit http://nexus.realtimepublishers.com.