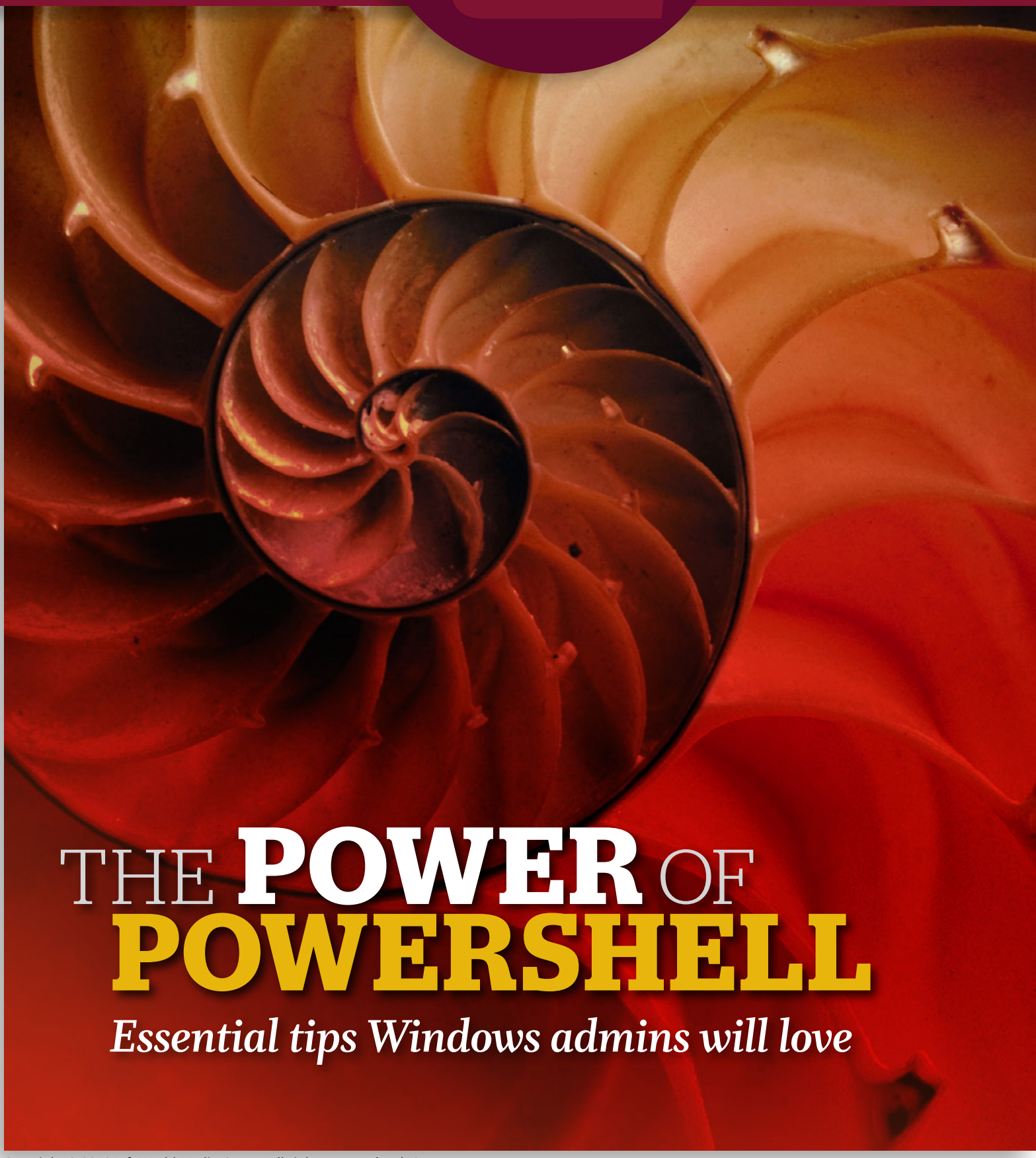
InfoWorld DeepDive



THE POWER OF POWERSHELL

Essential tips Windows admins will love

THINKSTOCK

Copyright © 2016 InfoWorld Media Group. All rights reserved. • $79

Deep Dive

The power of PowerShell: Essential tips Windows admins will love

Make the most of Microsoft's command line by mastering the nuances of the PowerShell language.

BY ADAM BERTRAM

Until recently, a clear delineation existed between Windows system administrators and developers. You'd never catch a system administrator writing a single line of code, and you'd never catch a developer bringing up a server. Neither party dared to cross this line in Windows environments. Nowadays, with the devops movement spreading like wildfire, that line is fading away.

A basic premise of devops is automation, which allows us to maintain consistent, repeatable processes while removing the error-prone ways of our being human. The only way to automate is through the command line. If you're in a Windows environment, the command line to use is PowerShell. Once considered an inferior command-line experience to Linux, Windows now touts a very powerful and functional command line through PowerShell.

PowerShell can be a daunting tool to master for Windows admins used to working with the GUI. But as I will show in this Deep Dive, adding even a little PowerShell into your daily mix can save a ton of effort.

Here, I'm going to concentrate on some of the fundamentals of PowerShell as a language to help you take your PowerShell skills to the next level. I won't cover technology-specific topics like how to manage Active Directory, Exchange, or IIS, nor will I cover specific PowerShell cmdlets. Instead, I will drill into the semantics of the language to show a few key techniques that you can put into your scripts today. Follow along and let's make your PowerShell scripts the best they can be.

Deep Dive

Splatting

PowerShell code is executed using commands such as cmdlets or functions. To increase their reuse value, commands typically employ parameters. Rather than requiring you to overspecify a custom function like `Get-ContentFromFileXYZ.TXT` and `Get-ContentFromFileABC.TXT`, PowerShell offers a Path parameter on its Get-Content cmdlet. Parameters, like Path, are typically passed to functions and cmdlets with a dash, followed by the parameter name, a space, and the parameter argument or value:

```
Get-Content –Path C:\FileABC.txt
```

But cmdlets and functions like `Get-Content` often support several parameters, and before long, you may be typing commands like this:

```
Get-Content -Path C:\MyText.txt -ReadCount 1 -TotalCount 3 -Force
-Delimiter “,” -Filter ‘*’ -Include * -Exclude ‘a’
```

Here, eight parameters are being passed to `Get-Content`, and it's clearly getting hard to read, to the point where confusion can arise in discerning the parameter from the value you're passing into the argument. Any time you use multiple parameters, it's wise to opt for another method of passing parameters to commands with the funny name of splatting.

Splatting, introduced in PowerShell Version 2, allows you to define all of your parameters up front in a much easier-to-read format.

Let's take that long, complicated set of parameters and see how all of them can be passed to `Get-Content` using splatting.

```
$getContentParameters = @{
    ‘Path’       = ‘C:\MyText.txt’
    ‘ReadCount’  = 1
    ‘TotalCount’ = 3
    ‘Force’      = $true
    ‘Delimiter’  = ‘,’
    ‘Filter’     = ‘*’
    ‘Include’    = ‘*’
    ‘Exclude’    = ‘a’
}
Get-Content @getContentParameters
```

Though I was forced to use more lines it's plain to see this method is much more readable than jamming them all on a single line. I also chose to line up the "`=`" statements by using tabs to have all of the parameter arguments lined up.

The `$getContentParameters` variable defined in the first line is a hashtable. You can see this by the `@{}`. Inside, each key represents the parameter, and each value represents the parameter argument or value. One item to note in this example is the `Force` parameter. You'll notice when `Force` was used on one line it wasn't technically equal to anything. This is because it is a type of switch. A switch parameter has no value and is simply used as a flag. Since hashtables require a value for each key, you must make any switch parameter used equal to a Boolean `$true` value.

Once you've defined the hashtable you then pass the entire variable to the command using the ampersand to represent a splatting variable. `Get-Content` will then process the commands exactly as if you'd passed them the traditional way with dashes.

Deep Dive

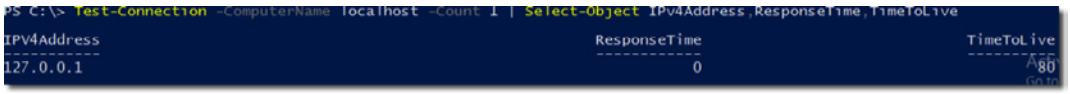
This is a much cleaner way to pass arguments to commands, and I encourage its use whenever multiple parameters are in play.

Calculated properties

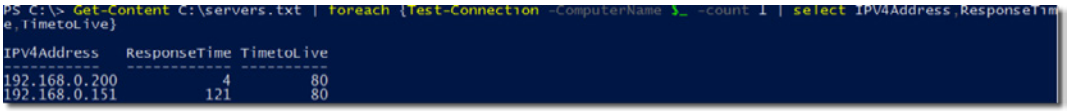
One of the biggest advantages of using PowerShell is that everything is an object. Objects have properties, but sometimes you may not want to get only the default property names on an object. Maybe you'd like to add another property, add text to an existing string property, or perform arithmetic on an integer property. You could send that property to a variable and make your necessary changes, but that requires another line and, depending on the complexity, can become unreadable real quick.

For example, let's say I want to test the network connectivity of a computer and get some common properties. To do this, I'll use the `Test-NetConnection` cmdlet.

```
Test-Connection -ComputerName localhost -Count 1 |
    Select-Object IPv4Address,ResponseTime,TimeToLive
```



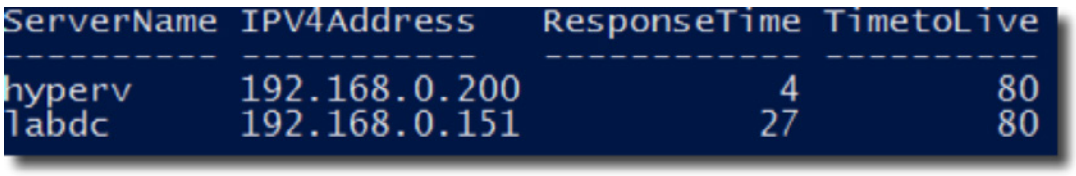
This is great, but maybe I want to put this into a CSV report. As part of the CSV report, I'm reading server names from a text file. I'd like to include the server name as well as the properties I've shown above. I start out by reading the text file, but I don't have the server names that were in the text file.



I need to add another property to this to correlate the IP with the server name. This is a job for calculated properties. Since this requires additional code, I will now move into a script rather than show you examples directly from the console.

```
Get-Content C:\servers.txt | foreach {
    $serverName = $PsItem
    $selectProperties = @{ Name = 'ServerName'; Expression = { $serverName } }, 'IPV4Address', 'ResponseTime', 'TimetoLive'
    Test-Connection -ComputerName $_ -count 1 | Select-Object $selectProperties
}
```

The output will now include our new custom property called `ServerName`.



I had to change the code quite a bit so allow me to break it down. First, notice how I moved the original properties being passed to `Select-Object` (`IPV4Address,ResponseTime,TimeToLive`) above, then passed `$selectProperties` to `Select-Object` that way. This was simply to reduce the length of the line. It behaves exactly the same as if I simply passed them directly to `Select-Object` as I did in the console previously.

What functionality *did* change was the addition of another property in `$selectProperties` to pass to `Select-Object`. Notice it wasn't a string like the others but a hashtable with two elements inside: `Name` and `Expression`. This is called a calculated property. This is how using `Select-Object` you can essentially create properties on the fly. Every property you'd like to add has to be in a hashtable with a `Name` and `Expression` as key names. The `Name` key's value is the name you'd like to call the object property. The `Expression` key's value always has to be a script block.

Deep Dive

Notice `$serverName` is enclosed in curly braces. This is how `$serverName` can be expanded to the actual server names as each server is tested in the text file.

This can be used not only to create new properties but to modify existing properties as well. Maybe I'd like to append a `ms` label to all the `TimeToLive` properties to signify the number is in milliseconds. Instead of specifying the name of the `TimeToLive` property I would instead create a hashtable and concatenate the `foreach` pipeline value of `TimeToLive` represented by `$_.TimeToLive` with `ms` to create a single string.

```
Get-Content C:\servers.txt | foreach {
    $serverName = $PsItem
    $selectProperties = 'IPV4Address', 'ResponseTime', @{ Name = 'TimeToLive'; Expression = { "$($_.TimetoLive) ms" } }
    Test-Connection -ComputerName $_ -count 1 | Select-Object $selectProperties
}
```

```
IPV4Address    ResponseTime TimeToLive
-----------    ------------ ----------
192.168.0.200             7 80 ms
192.168.0.151           142 80 ms
```

Objects abound in PowerShell, and it only makes sense for us to be able to create our own objects from scratch.

Using calculated properties with `Select-Object` are very convenient but beware: They come with a performance hit. I don't recommend using calculated properties if you're working with large data sets as it can drastically slow down your script. But if you have fairly small data sets with 100 or fewer elements the performance hit will be minimal.

Custom object creation

Objects abound in PowerShell, and it only makes sense for us to be able to create our own objects from scratch. Fortunately, PowerShell provides us with a few different ways to do that. In this tip, I'll cover three methods to create custom objects.

In PowerShell, an object is of a specific type. When creating custom objects, the most common type of object type is `System.Management.Automation.PSCustomObject`. This is the kind of object we'll create in this article. Also, an object has one or more properties of various types. In this article, we'll focus on `NoteProperty` types.

One of the oldest ways to create a custom object that works on all versions of PowerShell is via the `New-Object` cmdlet. To create a blank custom object of type `System.Management.Automation.PSCustomObject` with no properties using `New-Object` you'd simply call `New-Object` and specify the `TypeName` parameter of `PSObject`.

```
$object = New-Object –TypeName PSObject
```

```
PS C:\> $object = New-Object -TypeName PSObject
PS C:\> $object | Get-Member


   TypeName: System.Management.Automation.PSCustomObject

Name        MemberType Definition
----        ---------- ----------
Equals      Method     bool Equals(System.Object obj)
GetHashCode Method     int GetHashCode()
GetType     Method     type GetType()
ToString    Method     string ToString()
```

Deep Dive

This doesn't do us any good, however, because it contains no properties. To add properties, we can use the **Add-Member** cmdlet. This cmdlet essentially binds a new member (or property) to an existing object similar to the one we created.

```
$object | Add-Member –MemberType NoteProperty –Name MyProperty –Value SomeValue
```

```
PS C:\> $object = New-Object -TypeName PSObject -Property $properties
PS C:\> [pscustomobject]$properties

MyProperty2 MyProperty1
----------- -----------
Value2      Value1
```

You can see that **Add-Member** has a **MemberType** parameter. As I mentioned earlier, when creating your own objects you will typically use the **NoteProperty** type here. At this point, you simply need to specify the name of the property and the value that the property will hold.

From here, you can repeat adding as many properties as you'd like using **Add-Member**.

Next, rather than using **Add-Member** you can specify all your properties up front in a hashtable and pass those properties to **New-Object**.

```
$properties = @{ 'MyProperty1' = 'Value1'; 'MyProperty2' = 'Value2' }
$object = New-Object –TypeName PSObject –Property $properties
```

```
PS C:\> $properties = @{ 'MyProperty1' = 'Value1'; 'MyProperty2' = 'Value2' }
PS C:\> $object = New-Object -TypeName PSObject -Property $properties
PS C:\> $object

MyProperty2 MyProperty1
----------- -----------
Value2      Value1
```

Finally, as of PowerShell v3, we can use the **[pscustomobject]** type accelerator. Occasionally, the PowerShell team will create what's called type accelerators. These are convenient shortcuts to create objects of particular types. Since creating custom types are so common in PowerShell, they decided to create one for creating them. Nowadays, this is the most common way to create custom objects.

To create custom objects with the **[pscustomobject]** type accelerator you'll first need to create a hashtable with property names as key names and their values as the hashtable values. Let's reuse the hashtable we created earlier.

```
$properties = @{ 'MyProperty1' = 'Value1'; 'MyProperty2' = 'Value2' }
```

Now, instead of using the **New-Object** cmdlet and specifying the type and properties we can simply cast that hashtable directly to a custom object simply by "declaring" the hashtable as a custom object type.

```
[pscustomobject]$properties
```

```
PS C:\> $object | Add-Member -MemberType NoteProperty -Name MyProperty -Value SomeValue
PS C:\> $object

MyProperty
----------
SomeValue
```

You'll see it's a much faster way to create a custom object. I recommend this approach when working with PowerShell v3 or later. It's, by far, the easiest to remember and is the most readable.

In this tips and tricks article, we were able to cover a few language-specific concepts in PowerShell. Replicate what I've done, tinker around, use the **Get-Member** cmdlet to explore the custom objects further. PowerShell has so much more to offer from a language perspective. If you're new to PowerShell I recommend checking out the book "PowerShell in a Month of Lunches." It addresses the topics we've discussed here but covers much more of PowerShell and takes a well-rounded approach to learning the language. ■

Deep Dive

An intro for Windows Server admins

In this increasingly devops-minded world, automation is king. Here's how to get started with PowerShell.

BY ADAM BERTRAM

If you're a Windows system administrator you've probably been clicking buttons, dragging windows, and scrolling scroll bars for a long time. Using the GUI — even on a server — has been common in the Windows world. We thought nothing of firing up a remote desktop client and logging into a server to do our bidding. This might be OK for very small businesses with only a couple servers, but enterprises soon realized this approach doesn't scale. Something else had to be done. The solution was to turn to scripts to automate as much of this management as possible. With the introduction of PowerShell, sys admins now had a tool to make it happen.

Here we provide a hands-on introduction to what Windows system administrators can accomplish using PowerShell in an increasingly devops-minded world. So roll up your sleeves, fire up PowerShell, and follow along.

Event log querying across servers

One task I struggled with as an IT pro was automating the reading of event logs. Windows event logs contain a ton of useful information. The problem is getting to all the data and collating it into a meaningful form. Using PowerShell cmdlets like `Get-WinEvent` and `Get-EventLog`, I could finally gather information from one to 1,000 servers with relative ease.

Before we start writing our code, we need to figure out what events we want to retrieve. For the purposes of this introduction, I'll be searching for event ID 6005 in my servers' System event log. Why that ID? Because the ID is used to indicate when a server has started. If you want to search for a different event, you can easily switch out this ID for one that represents your desired event.

Before I can start pulling these events from all my servers, I first need to know how to do it on a single server. There are a few cmdlets that can do this, such as `Get-EventLog` and `Get-WinEvent`.

THINKSTOCK

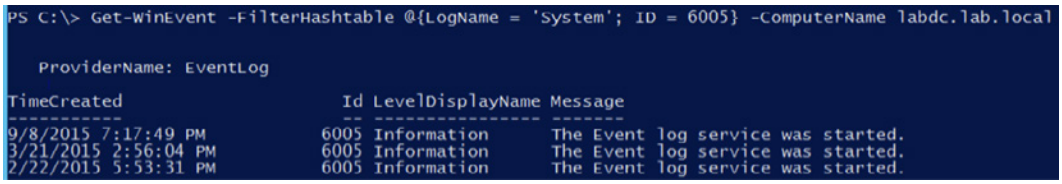
Deep Dive

I'm going to use `Get-WinEvent`. This cmdlet is typically faster and allows you to perform more advanced filtering. It is a little harder to get a handle on than `Get-EventLog`, but I believe a more thorough understanding of `Get-WinEvent` pays off in the long run.

Use `Get-WinEvent`

To retrieve information about events on remote computers we need to use the `-ComputerName` parameter. Because servers produce multiple event logs, we'll then need to narrow that down by the System event log and finally the event ID. `Get-WinEvent` offers a few options for filtering, but the easiest to use is the `-FilterHashTable` parameter. By using the `LogName` and `ID` as hashtable keys we can easily narrow down what kind of events will be retrieved.

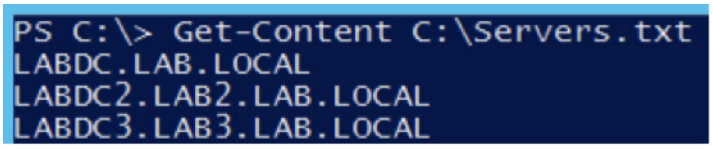
```
Get-WinEvent -FilterHashtable @{LogName = 'System'; ID = 6005}
-ComputerName labdc.lab.local
```



You can see in the screenshot above that I'm querying the `labdc.lab.local` computer and I can see three events with the `ID` of `6005`.

Expand the search to more servers

As you can see, finding all instances of an event ID on a single server is easy, but what about multiple servers? One way is to use a simple text file listing all of your server names. If you have Active Directory up and running, you can easily use `Get-AdComputer` to pull necessary server names as well. For this introduction, I'll use a text file, from which the `Get-Content` cmdlet can pull out all of my server names into a variable.



You can see I have three servers in the text file. I'd like to store all of these server names in a variable to reference in a minute, so I'll create one called `$Servers`.

```
$Servers = Get-Content -Path C:\Servers.txt
```

Now I can check each one of those servers by placing that `Get-WinEvent` line we defined earlier inside of a `foreach` loop where we iterate over each line in the `C:\Servers.txt` file. Although this time, instead of specifying the name of an individual server, I'm using the variable `$s`. This represents each server name as it's processed in `$Servers`.

```
foreach ($s in $Servers) {
   Get-WinEvent -FilterHashtable @{LogName = 'System'; ID = 6005}
-ComputerName $s
}
```

Deep Dive

When you run this you'll get something that looks like this. Not every helpful, right? Which servers did these events come from? We'll need to add a bit more code to get the output just right.

```
   ProviderName: EventLog

TimeCreated                     Id LevelDisplayName Message
-----------                     -- ---------------- -------
9/8/2015 7:17:49 PM           6005 Information      The Event log service was started.
3/21/2015 2:56:04 PM          6005 Information      The Event log service was started.
2/22/2015 5:53:31 PM          6005 Information      The Event log service was started.
9/12/2015 2:42:48 PM          6005 Information      The Event log service was started.
8/14/2015 4:30:06 PM          6005 Information      The Event log service was started.
4/7/2015 1:34:32 PM           6005 Information      The Event log service was started.
4/7/2015 12:25:10 AM          6005 Information      The Event log service was started.
3/21/2015 2:56:54 PM          6005 Information      The Event log service was started.
11/16/2014 8:08:28 PM         6005 Information      The Event log service was started.
11/3/2014 5:54:35 PM          6005 Information      The Event log service was started.
```

By default, PowerShell tries to be helpful by showing you only what it thinks are the essentials. However, sometimes you need to see more, as is the case here, because what you're seeing from **`Get-WinEvent`** is not the *true* output. You're only seeing what PowerShell is configured to output to the console. To see all of the properties that come out of **`Get-WinEvent`** you'll need to use the **`Select-Object`** cmdlet or **`select`** as it is aliased.

Be selective on properties with Select-Object

```
Get-WinEvent -FilterHashtable @{LogName = 'System'; ID = 6005} - ComputerName
labdc.lab.local | select -First 1 *
```

```
Message               : The Event log service was started.
Id                    : 6005
Version               :
Qualifiers            : 32768
Level                 : 4
Task                  : 0
Opcode                :
Keywords              : 36028797018963968
RecordId              : 102660
ProviderName          : EventLog
ProviderId            :
LogName               : System
ProcessId             :
ThreadId              :
MachineName           : LABDC.lab.local
UserId                :
TimeCreated           : 9/8/2015 7:17:49 PM
ActivityId            :
RelatedActivityId     :
ContainerLog          : system
MatchedQueryIds       : {}
Bookmark              : System.Diagnostics.Eventing.Reader.EventBookmark
LevelDisplayName      : Information
OpcodeDisplayName     :
TaskDisplayName       :
KeywordsDisplayNames  : {Classic}
Properties            : {System.Diagnostics.Eventing.Reader.EventProperty}
```

Using the **`-First`** parameter to **`Select-Object`** (here as the alias **`select`**) returns the first event record on the System event log of computer **`labdc.lab.local`**, and the asterisk at the end of the command allows me to see *all* the properties, not only the ones PowerShell displays to the console by default. Notice that there's a property called **`MachineName`**? This is exactly what we need in our output.

Deep Dive

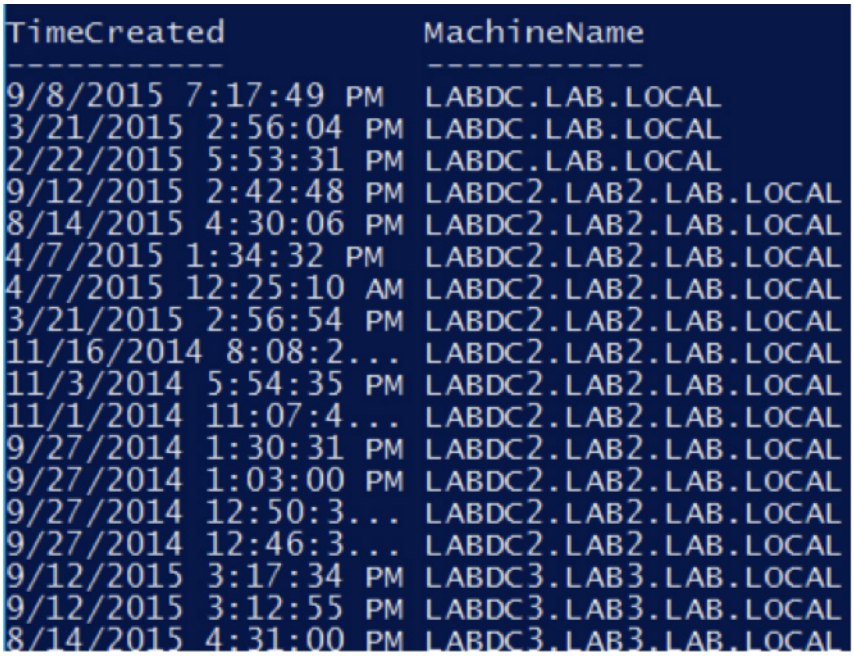
```
foreach ($s in $Servers) {
   $getWinEventParams = @{
       'FilterHashTable' = @{LogName = 'System'; ID = 6005}
       'ComputerName' = $s
   }
   Get-WinEvent @getWinEventParams | Select-Object
TimeCreated,MachineName
}
```

```
TimeCreated             MachineName
-----------             -----------
9/8/2015 7:17:49 PM     LABDC.lab.local
3/21/2015 2:56:04 PM    LABDC.lab.local
2/22/2015 5:53:31 PM    LABDC.lab.local
9/12/2015 2:42:48 PM    LABDC2.lab2.lab.local
8/14/2015 4:30:06 PM    LABDC2.lab2.lab.local
4/7/2015 1:34:32 PM     LABDC2.lab2.lab.local
4/7/2015 12:25:10 AM    LABDC2.lab2.lab.local
3/21/2015 2:56:54 PM    LABDC2.lab2.lab.local
11/16/2014 8:08:2...    LABDC2.lab2.lab.local
11/3/2014 5:54:35 PM    LABDC2.lab2.lab.local
11/1/2014 11:07:4...    LABDC2.lab2.lab.local
9/27/2014 1:30:31 PM    LABDC2.lab2.lab.local
9/27/2014 1:03:00 PM    LABDC2
9/27/2014 12:50:3...    WIN-85RADI8TKIS
9/27/2014 12:46:3...    WIN-DDM7D664Q0O
9/12/2015 3:17:34 PM    LABDC3.lab3.lab.local
9/12/2015 3:12:55 PM    LABDC3.lab3.lab.local
8/14/2015 4:31:00 PM    LABDC3.lab3.lab.local
4/7/2015 1:36:24 PM     LABDC3.lab3.lab.local
4/7/2015 12:27:08 AM    LABDC3.lab3.lab.local
3/21/2015 2:59:27 PM    LABDC3.lab3.lab.local
11/16/2014 8:09:2...    LABDC3.lab3.lab.local
11/3/2014 5:54:45 PM    LABDC3.lab3.lab.local
11/1/2014 11:07:5...    LABDC3.lab3.lab.local
9/27/2014 1:30:46 PM    LABDC3.lab3.lab.local
9/27/2014 1:01:31 PM    LABDC3
9/27/2014 12:51:4...    WIN-L3DH8QS5NHO
9/27/2014 12:48:4...    WIN-DDM7D664Q0O
```

By using the **Select-Object** cmdlet to manipulate the output from **Get-WinEvent** I can now limit our output to the **TimeCreated** and **MachineName** properties. Notice in my screenshot some odd-looking machine names? That was when the server I'm querying was named something else. I don't really care what the server was named a long time ago. It looks like I might not be able to use the **MachineName** property. Instead, I'll use the value of **$s** to ensure I get a consistent server name.

Deep Dive

```
foreach ($s in $Servers) {
   $getWinEventParams = @{
        'FilterHashTable' = @{LogName = 'System'; ID = 6005}
        'ComputerName' = $s
   }
   Get-WinEvent @getWinEventParams | Select TimeCreated,@
{n='MachineName';e = {$s}}
}
```



TimeCreated	MachineName
9/8/2015 7:17:49 PM	LABDC.LAB.LOCAL
3/21/2015 2:56:04 PM	LABDC.LAB.LOCAL
2/22/2015 5:53:31 PM	LABDC.LAB.LOCAL
9/12/2015 2:42:48 PM	LABDC2.LAB2.LAB.LOCAL
8/14/2015 4:30:06 PM	LABDC2.LAB2.LAB.LOCAL
4/7/2015 1:34:32 PM	LABDC2.LAB2.LAB.LOCAL
4/7/2015 12:25:10 AM	LABDC2.LAB2.LAB.LOCAL
3/21/2015 2:56:54 PM	LABDC2.LAB2.LAB.LOCAL
11/16/2014 8:08:2...	LABDC2.LAB2.LAB.LOCAL
11/3/2014 5:54:35 PM	LABDC2.LAB2.LAB.LOCAL
11/1/2014 11:07:4...	LABDC2.LAB2.LAB.LOCAL
9/27/2014 1:30:31 PM	LABDC2.LAB2.LAB.LOCAL
9/27/2014 1:03:00 PM	LABDC2.LAB2.LAB.LOCAL
9/27/2014 12:50:3...	LABDC2.LAB2.LAB.LOCAL
9/27/2014 12:46:3...	LABDC2.LAB2.LAB.LOCAL
9/12/2015 3:17:34 PM	LABDC3.LAB3.LAB.LOCAL
9/12/2015 3:12:55 PM	LABDC3.LAB3.LAB.LOCAL
8/14/2015 4:31:00 PM	LABDC3.LAB3.LAB.LOCAL

Yay! Now I don't see those old names in there. I only have to change the **`MachineName`** property by using a calculated property. This allowed me to create a new **`MachineName`** property with a value of my choosing.

Also, you'll notice that I created a **`$getWinEventParams`** variable and passed that to **`Get-WinEvent`** instead of passing those parameters individually. This is a method called splatting that allows you to pass parameters to cmdlets, rather than having to pass all of them on a single line. It's a clean way of passing parameters to commands in PowerShell.

Build a server inventory report: A lesson in CIM

Dozens of tools on the market today can inventory your servers. These range from expensive, full-blown suites like Microsoft System Center Configuration Manager, Altiris, and the rest to absolutely free tools. These work, of course, but what if you don't want to spend time learning a new piece of software or simply need to quickly query something from a few servers? Maybe you have specific requirements and the tool you usually use can't meet that requirement. You can use PowerShell instead.

Before we begin coding, let's determine what we need pulled from each of the servers. For this example, I'll pull the following information from each server.

1. Operating system
2. Total memory
3. Processor name and speed
4. Total disk space on the C: drive

Deep Dive

The Active Directory module

Also, because the Active Directory cmdlets do not come with PowerShell out of the box I'll need to download and install Remote Server Administration Tools (RSAT). This will give me the Active Directory module.

Again, as with the event log report we gathered, we'll need a way to get a list of server names. Rather than use a text file, let's get these names from Active Directory (AD). For today, I want to gather some information on all of my global catalog servers in my AD forest.

This is a walk in the park using the **Get-ADforest** cmdlet.

```
PS C:\> (Get-ADForest).GlobalCatalogs
LABDC.lab.local
LABDC2.lab2.lab.local
LABDC3.lab3.lab.local
LABDC3-2.lab3.lab.local
```

Let's assign those servers to a variable again.

```
$Servers = (Get-ADForest).GlobalCatalogs
```

Introducing CIM

Now that we have our list of servers, how do we pull our target information? For this, we'll need to understand a little about Common Information Model (CIM). Every Windows machine has a CIM repository. This repository holds hundreds of classes. Each of these classes contains object properties. One way to query these classes is through the **Get-CimInstance** cmdlet. This is a newer cmdlet that uses PSRP (PowerShell remoting protocol). This means you must have WinRM enabled and available on all of your servers. I'm using all Windows Server 2012 R2 servers configured to have WinRM available in my demonstrations, so your mileage may vary.

The four attributes I am looking for exist in different CIM classes on each of my servers. To save time in tracking these down, here's the breakdown:

1. Operating system → **Win32_OperatingSystem**
2. Total memory → **Win32_PhysicalMemory**
3. Processor name and speed → **Win32_Processor**
4. Total disk space on C: drive → **Win32_LogicalDisk**

To query each of these classes I'll use the **Get-CimInstance** cmdlet. This command has two parameters we'll need to use: **-ClassName** and **-ComputerName**. You can use **Get-CimInstance** as demonstrated below.

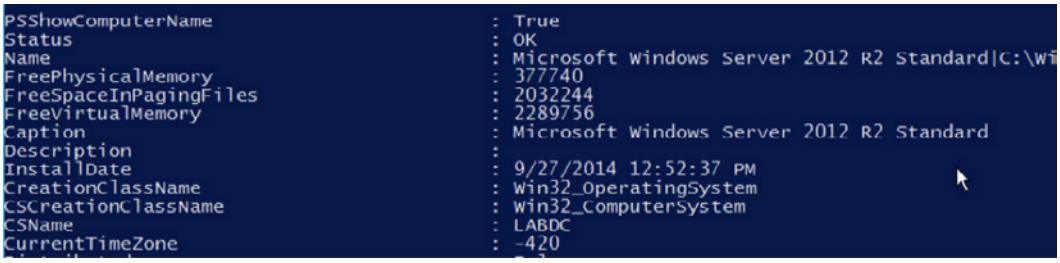
```
PS C:\> Get-CimInstance -ClassName Win32_OperatingSystem -ComputerName labdc.lab.local

SystemDirectory     Organization BuildNumber RegisteredUser SerialNumber           Version  PSComputerName
---------------     ------------ ----------- -------------- ------------           -------  --------------
C:\Windows\system32              9600        Windows User   00252-00108-14656-AA566 6.3.9600 labdc.lab.local
```

Let's see if we can get the operating system name. Notice how I queried the **Win32_OperatingSystem** class above, but you don't see the operating system name? What gives? As with the **Get-WinEvent** cmdlet we went over earlier, **Get-CimInstance** also does not show the "real" output. However, in this instance we won't use the **Select-Object** cmdlet. Instead, **Get-CimInstance** has a **-Property** parameter where we can specify an asterisk to see all of the properties.

```
Get-CimInstance -ClassName Win32_OperatingSystem -ComputerName labdc.lab.local
-Property *
```

Deep Dive

Once you do this, if all goes well, you should see an output with lots of different property names, including one called `Caption`, which contains the name of the operating system.



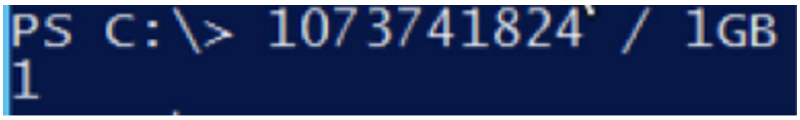
Now that I know the property name, I can limit our output to displaying the `Caption` property alone, with no need for the `-Property` parameter anymore. That was only needed to check out the values.

```
(Get-CimInstance -ClassName Win32_OperatingSystem -ComputerName
labdc.lab.local).Caption
```

This same methodology applies to all of the other attributes we need to retrieve. To find the processor we'll use the Name property on the `Win32_Processor` class.

```
(Get-CimInstance -ClassName Win32_Processor -ComputerName
labdc.lab.local).Name
```

The same goes for `Win32_PhysicalMemory` and `Win32_LogicalDisk` with one caveat. If you try to retrieve the total memory on a server using the `Capacity` property you'll get the total *bytes* back. For example, if I have 1GB in a demo server, it will show up as 1073741824. That's not very intuitive. I'd like to get that back in gigabytes. Luckily, this is easy to do in PowerShell. Simply divide total bytes by 1GB.



Put it all together

Now that you have an understanding of how to retrieve this information on a single server, let's put it all together in a `foreach` loop to query each of our servers.

```
$Servers = (Get-ADForest).GlobalCatalogs
foreach ($Server in $Servers) {
      $Output = @{'Name' = $Server }
      $Session = New-CimSession -Computername $Server
      if ($Session) {
          $Output.OperatingSystem = (Get-CimInstance -CimSession $Session
-ClassName Win32_OperatingSystem).Caption
          $Output.Memory = (Get-CimInstance -CimSession $Session
-ClassName Win32_PhysicalMemory).Capacity / 1GB
          $Output.CPU = (Get-CimInstance -CimSession $Session -ClassName
Win32_Processor).Name
          $Output.FreeDiskSpace = (Get-CimInstance -CimSession $Session
-ClassName Win32_LogicalDisk -Filter "DeviceID = 'C:'").FreeDiskSpace /
1GB
```

Deep Dive

```
            Remove-CimSession $Session
            [pscustomobject]$Output
        }
}
```

Notice that I used the `New-CimSession` cmdlet. Instead of the `-ComputerName` parameter on each `Get-CimInstance` line, I use the `-CimSession` parameter instead. This is for performance reasons. Every time `Get-CimInstance` runs with the `-ComputerName` parameter it creates a temporary new CIM session on the remote computer. Since I need to make multiple calls to CIM, I can simply create a single CIM session for each server and repeatedly use that single session. It's more efficient and faster.

Also notice the `$Output` hashtable variable. Since it's not possible for me to gather all of these attributes with a single command I have to store the results in a variable for each server. Then, once I am done with the server I convert the $Output hashtable to a custom object.

If the stars align, you should get an output similar to this.

```
FreeDiskSpace   : 0
Memory          : 1
Name            : LABDC.lab.local
CPU             : Intel(R) Core(TM) i5 CPU         760  @ 2.80GHz
OperatingSystem : Microsoft Windows Server 2012 R2 Standard

FreeDiskSpace   : 0
Memory          : 1
Name            : LABDC2.lab2.lab.local
CPU             : Intel(R) Core(TM) i5 CPU         760  @ 2.80GHz
OperatingSystem : Microsoft Windows Server 2012 R2 Standard

FreeDiskSpace   : 0
Memory          : 1.46484375
Name            : LABDC3.lab3.lab.local
CPU             : Intel(R) Core(TM) i5 CPU         760  @ 2.80GHz
OperatingSystem : Microsoft Windows Server 2012 R2 Standard
```

This was a tiny sample of how you can use PowerShell as a Windows sys admin. There are tons of other opportunities where PowerShell can be used. I encourage you to see them out, learn more about PowerShell, and find out where it can take you and your career.

If you are interested in any of the code demonstrated in today's article, feel free to download it from my GitHub repository. GitHub is where I share all of my PowerShell content. You'll find everything we covered today there and much more about PowerShell. ■

Deep Dive

The power of PowerShell: An intro for Exchange admins

Forgo the GUI in favor of the console and free yourself of the drudgery of administering Exchange.

BY ADAM BERTRAM

Managing Exchange can be a significant time sink, and you may be surprised to find that much of the time spent administering Exchange is pure waste — unless, that is, you're already tapping PowerShell.

"Waste?!?" you may be thinking. "I don't waste any time managing Exchange!" But if you're diligently attached to the Exchange Admin Center building mailboxes, changing routing groups, restoring email, and so on, you're spending far too much time clicking around in a GUI. Repetitive tasks are the bread and butter of the PowerShell console. If you aren't already deeply versed in PowerShell, this hands-on tutorial will convert you.

Windows PowerShell has its roots in Exchange Server. Way back in Exchange Server 2007, the Microsoft Exchange team decided to take a gamble on this new scripting language, and PowerShell has since expanded its reach to nearly every facet of IT. If you're an Exchange administrator, you've probably used PowerShell *some*. Here, we delve a little deeper, going over a few examples of what you can do in PowerShell with Exchange that will cut down the time you spend on administrative drudgery.

Get connected without RDPing into your Exchange server

To tap the power of PowerShell, you first have to connect to your Exchange server. You *could* remote desktop onto one of your Exchange servers and fire up the Exchange Management Shell, but I don't recommend that approach at all. Servers are meant to process workloads, not to power GUIs. Instead, there's a better way to connect, called implicit remoting.

All the PowerShell functionality that comes bundled with Exchange Server is wrapped up in a PowerShell module. When Exchange is installed, the module is installed on the server, not your admin workstation. Thus, you will need a way to make all of those Exchange PowerShell functions available on your admin workstation. Implicit remoting allows you to do that.

THINKSTOCK

Deep Dive

To make use of implicit remoting, you must first establish a remote session to the Exchange server. Once your session is established, you will then virtual import this remote session into your local session, which will make it seem like you're on the server's PowerShell console itself.

To create that remote session, we'll use the **New-PSSession** cmdlet. Exchange provides specific configuration options for remote sessions, so we'll need to specify a few more parameters than simply the name of the server and a credential that typically applies to new remote sessions. This means specifying the **ConfigurationName** of **Microsoft.Exchange** and a **ConnectionUri** of **http://MYEXCHANGESERVER.MYDOMAIN.LOCAL/PowerShell/?SerializationLevel=Full**, subbing in your environment's server info for what's in all caps in the URI.

```
$Session = New-PSSession –ConfigurationName Microsoft.Exchange
–ConnectionUri 'http://MYEXCHANGESERVER.MYDOMAIN.LOCAL/PowerShell/?SerializationLevel
=Full'
```

This will create a remote session inside the **$Session** variable. Once you have this, it's then a matter of importing that remote session into your current local session by using the **Import-PSSession** cmdlet.

```
Import-PSSession -Session $Session
```

```
PS C:\> Import-PSSession -Session $Session

ModuleType Name                                ExportedCommands
---------- ----                                ----------------
Script     tmp_7c3100f5-789e-40d9...           {Set-MailboxCalend
```

If all goes well, you should see your module output as shown above. You should now have all of the cmdlets available to the Exchange server now available on your local admin workstation. Here is where the real power of managing Exchange with PowerShell begins.

Manage Exchange mailboxes with PowerShell

One of the most common Exchange administrative tasks is managing mailboxes. Your organization probably has more than a few mailboxes, right? Mailboxes have a bunch of common attributes. For example, every mailbox has an assigned username, a quota, set policies, sending limits, and so on. This is where use of PowerShell shines. Let's say you need to change the quota on 100 mailboxes. That would be painful using the GUI, but it's pretty easy using PowerShell.

Wherever you find yourself having to perform a certain action on a common set of objects the first place you should look is PowerShell. Let's go over a few examples.

Find mailboxes

Let's say you have thousands of mailboxes spread across multiple mailbox servers. Perhaps you need to find some information about one of your user's mailboxes. Let's pick on Virginie Jean. To do this, we'd simply use the **Get-Mailbox** cmdlet, which returns limited, but useful information about mailboxes.

```
Get-Mailbox -Identity 'Virginie Jean'
```

```
[PS] C:\>Get-Mailbox -Identity 'Virginie Jean'

Name                 Alias                ServerName           ProhibitSendQuota
----                 -----                ----------           -----------------
Virginie Jean        ViJe                 ex01                 unlimited
```

Deep Dive

Limited information indeed — what if I'm looking for an attribute of her mailbox that I don't see among the above defaults? By using the PowerShell pipeline and the **Select-Object** cmdlet with an asterisk for the **-Property** parameter, I can get the full extent of what the **Get-Mailbox** cmdlet pulls about the mailbox, not only Microsoft's "friendly" defaults. The screenshot below is a tiny fraction of the information you can glean from **Get-Mailbox**. There are actually dozens of attributes about the mailbox you can find.

```
Get-Mailbox -Identity 'Virginie Jean' | SelectObject -Property *
```

```
[PS] C:\>Get-Mailbox -Identity 'Virginie Jean' | Select-Object -Property *

RunspaceId                          : dc703423-b31f-4282-8ed4-7f39b2b48137
Database                            : Mailbox Database 1945161784
DeletedItemFlags                    : DatabaseDefault
UseDatabaseRetentionDefaults        : True
RetainDeletedItemsUntilBackup       : False
DeliverToMailboxAndForward          : False
LitigationHoldEnabled               : False
SingleItemRecoveryEnabled           : False
RetentionHoldEnabled                : False
EndDateForRetentionHold             :
StartDateForRetentionHold           :
RetentionComment                    :
```

Pretty easy — but what if you need to find information about a lot of users? If all of those users match some kind of pattern you can use the **-Filter** parameter.

Maybe I want to find all of the mailboxes owned by someone named Adam.

```
Get-Mailbox -Filter {Name -like 'Adam *'}
```

```
[PS] C:\>Get-Mailbox -Filter {Name -like 'Adam *'}

Name                 Alias                ServerName       ProhibitSendQuota
----                 -----                ----------       -----------------
Adam Barr            AdBa                 ex01             unlimited
Adam Carter          AdCa                 ex01             unlimited
Adam Kodeda          AdKo                 ex01             unlimited
Adam Trukawka        AdTr                 ex01             unlimited
```

Change mailboxes in bulk

Finding mailboxes is rarely the end of your task. Instead you'll likely want to *do* something with those mailboxes after you've found them. Here, piping your set of mailboxes to another cmdlet like **Set-Mailbox** or **Remove-Mailbox** is key.

What if the owners of all of those mailboxes that start with "Adam" went on vacation and we needed to change the forwarding address? Simple:

```
Get-Mailbox -Filter {Name -like 'Adam *'} | Set-Mailbox -ForwardingAddress
'somecontact@domain.local'
```

Or maybe their owners left the organization, and I want to remove the mailboxes. Simply change **Set-Mailbox** to **Remove-Mailbox** and they're gone.

```
Get-Mailbox -Filter {Name -like 'Adam *'} | Remove-Mailbox
```

Chaining a number of commands together by piping them directly from one cmdlet to another in this way is a powerful way of addressing Exchange admin tasks through PowerShell.

Deep Dive

Mailbox statistics and CSV reports

Another common activity that Exchange administrators must do is finding where all that hard-earned storage is allocated. Which mailbox takes up the most space on the SAN, and which folders in that mailbox are taking up so much space? This kind of information can easily be found with PowerShell using the **Get-MailboxFolderStatistics** cmdlet.

While we're at it, let's add another PowerShell cmdlet called **Export-Csv** to our arsenal. Recall that piping thing we did earlier? That isn't limited to mailboxes. The pipeline is one of the most powerful features of PowerShell for good reason. In this section, I'll show you how you can use the pipeline to take data returned from any cmdlet and immediately turn it into a nicely formatted CSV file with little effort.

Using the **Get-MailboxFolderStatistics** cmdlet you can easily see what items are in a folder, how much space each item takes up, and other useful information. Let's try the cmdlet on Virginie Jean's mailbox again.

```
Get-MailboxFolderStatistics -Identity 'Virginie Jean'
```

```
[PS] C:\>Get-MailboxFolderStatistics -Identity 'Virginie Jean'

RunspaceId                    : 357f211c-619e-4c9a-b654-b0619af0e9d6
Date                          : 11/25/2009 8:55:34 PM
Name                          : Top of Information Store
FolderPath                    : /Top of Information Store
FolderId                      : LgAAAACrvkl4122jTbMzx/n1+K4sAQD22ZH19SOjTozv/QOrtwMHBDrRpyxuAAAB
FolderType                    : Root
ItemsInFolder                 : 0
FolderSize                    : 0 B (0 bytes)
ItemsInFolderAndSubfolders    : 0
FolderAndSubfolderSize        : 0 B (0 bytes)
OldestItemReceivedDate        :
NewestItemReceivedDate        :
ManagedFolder                 :
Identity                      : Virginie Jean\Top of Information Store
IsValid                       : True
```

The above screenshot shows a single object returned from the command, which is the top of the information store. In total, Virginie's mailbox returned 15 different folders from her calendar, notes, email folders, and so on. Suppose you want to see the mailbox's calendar only. Simply append the **-FolderScope** parameter at the end of this command and use the argument **Calendar**.

You can see that you have a lot of control over the types of folders to look for.

```
help Get-MailboxFolderStatistics -Parameter FolderScope
```

```
[PS] C:\>help get-mailboxfolderstatistics -Parameter folderscope

-FolderScope <Nullable>
    The FolderScope parameter specifies the scope of the search by folder type. Valid parameter values include:
    * All
    * Calendar
    * Contacts
    * ConversationHistory
    * DeletedItems
    * Drafts
    * Inbox
    * JunkEmail
    * Journal
    * ManagedCustomFolder
    * Notes
    * Outbox
    * RecoverableItems
    * RssSubscriptions
    * SentItems
    * SyncIssues
    * Tasks
    If the ManagedCustomFolder value is entered, the command returns the output for all managed custom folders.
    ue is entered, the command returns the output for the Recoverable Items folder and the Deletions, Purges, a

    Required?                    false
    Position?                    Named
    Default value
    Accept pipeline input?       False
    Accept wildcard characters?  false
```

Deep Dive

You've seen an example of a single mailbox, so let's take this global. How hard it is to search *all* mailbox folders and get a report on who's hogging up storage? **Get-MailboxFolderStatistics** doesn't have an option to find all mailboxes, so we'll have to go back to the pipeline again.

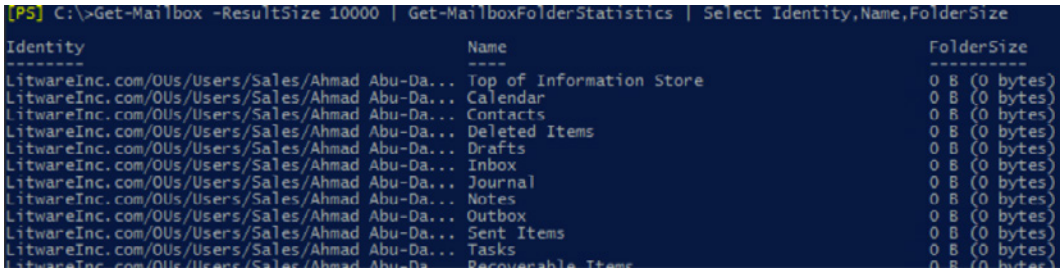
```
Get-Mailbox | Get-MailboxFolderStatistics
```

This simple command will begin enumerating all mailboxes in your organization. As it's finding each mailbox, it will then "pipe" the mailbox to **Get-MailboxFolderStatistics**, where it will begin enumerating all the folders inside each mailbox. Note, however, that by default Get-Mailbox will return only the first 1,000 mailboxes it sees. If your organization has more than 1,000 mailboxes you'll have to use the **-ResultSize** parameter to increase the limit. My demo organization has more than 1,000 mailboxes, so I will use this parameter when running the report.

We now have the ability to get all mailboxes and all folders inside the mailboxes. **Get-MailboxFolderStatistics** returns a lot of different properties for the folders. Since we're building a space report we don't need all of them. I only need the mailbox name, folder name, and the size of each folder. To limit my output to these properties I'll use the **Select-Object** cmdlet and the **-Property** parameter. You'll see in the screenshot below that I'm using the **Select-Object** alias **Select** and I'm excluding the **-Property** parameter, instead choosing what PowerShell calls positional binding.

```
Get-Mailbox -ResultSize 10000 |
  Get-MailboxFolderStatistics |
    Select Identity,Name,FolderSize
```

(Note: This is typed as one line in PowerShell. Due to the line length, it is broken into different lines for formatting reasons.)



I included a few of the objects to give you an example of what you might see. Now we're getting only the information necessary for the report. Next, I'd like to sort all of the folders by **FolderSize**. No problem — we'll simply pipe the folders into the **Sort-Object** cmdlet and use the **-Descending** parameter to ensure we see the biggest folders first.

```
Get-Mailbox -ResultSize 10000 |
  Get-MailboxFolderStatistics |
    Select Identity,Name,FolderSize |
      Sort-Object FolderSize -Descending
```

(Note: This is typed as one line in PowerShell. Due to the line length, it is broken into different lines for formatting reasons.)

Deep Dive

Finally, we have the data we're looking for, but perhaps our manager wants this information and we need to email him. Since this is tabular data, a CSV would be perfect. How can we get this to output as a CSV file? With one more cmdlet and pipe:

```
Get-Mailbox -ResultSize 10000 |
  Get-MailboxFolderStatistics |
    Select Identity,Name,FolderSize |
      Sort-Object FolderSize -Descending |
        Export-Csv -Path C:\FolderSizes.csv -NoTypeInformation
```

(Note: This is typed as one line in PowerShell. Due to the line length, it is broken into different lines for formatting reasons.)

All I had to do was add **Export-Csv** to the end to send all the output to a CSV file rather than the console. With **Export-Csv** you'll use the **-NoTypeInformation** parameter a lot. If you do not include this parameter, a small line will be included at the top of your CSV that contains text that has nothing to do with the output.

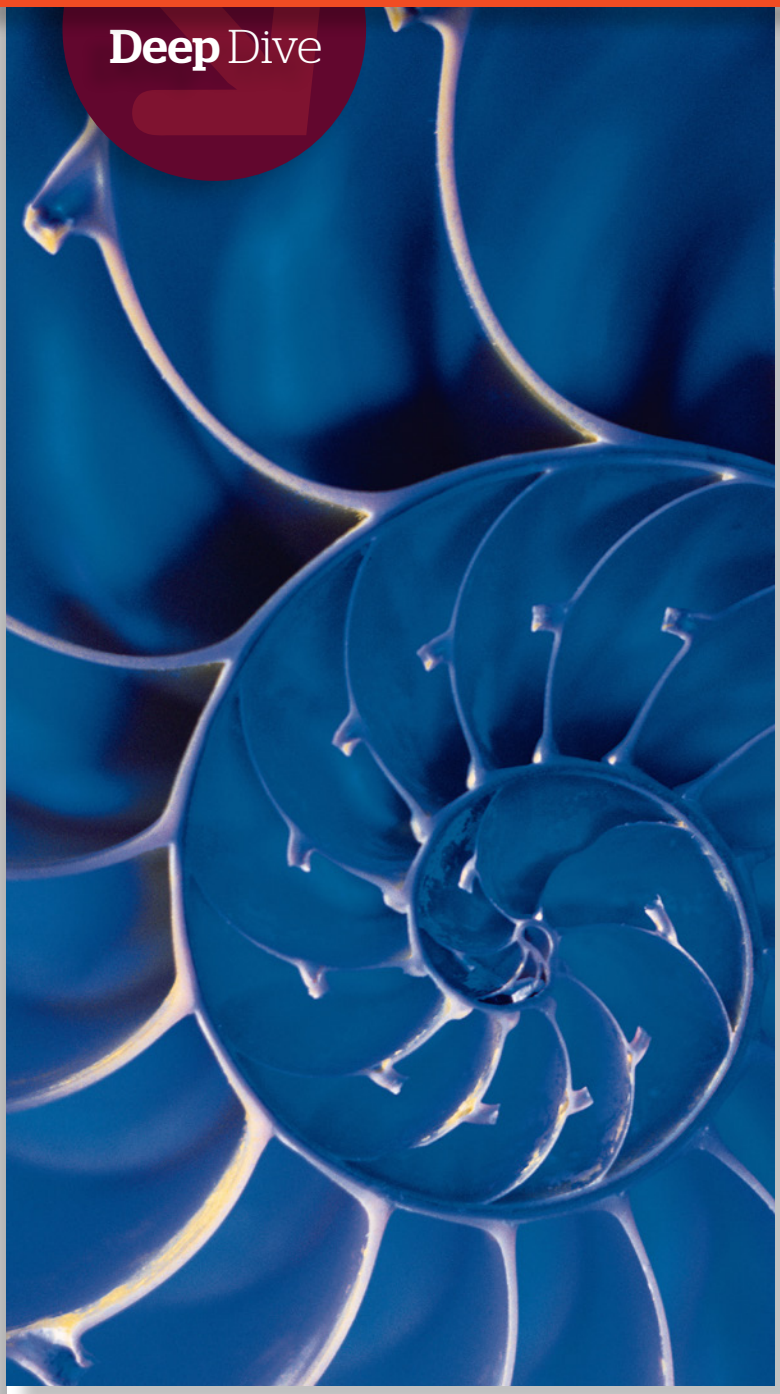
Once it's done you should then have a CSV file that contains every mailbox identity, name, and the size sorted by largest to smallest.

If you're curious as to how well it went without opening up Excel, you can easily read the top few rows of the CSV using the **Import-Csv** cmdlet and **Select-Object** again. Since the **Identity** property can get pretty long you can force the rows to fit nicely by ending the pipeline with the **Format-Table** cmdlet using the **-AutoSize** parameter.

```
Import-Csv C:\FolderSizes.csv | select -first 10 | Format-Table -AutoSize
```

```
[PS] C:\>Import-Csv C:\FolderSizes.csv | select -first 10 | Format-Table -AutoSize

Identity                                                    Name       FolderSize
--------                                                    ----       ----------
LitwareInc.com/OUs/Users/Information Technology/Hao Chen\Inbox          Inbox      9.031 KB (9,248 bytes)
LitwareInc.com/OUs/Users/Information Technology/Lucio Iallo\Inbox       Inbox      7.238 KB (7,412 bytes)
LitwareInc.com/OUs/Users/Information Technology/Vivian Atlas\Sent Items Sent Items 6.664 KB (6,824 bytes)
LitwareInc.com/OUs/Users/Information Technology/Garrett Young\Calendar  Calendar   5.576 KB (5,710 bytes)
LitwareInc.com/OUs/Users/Information Technology/Hao Chen\Calendar       Calendar   5.479 KB (5,611 bytes)
LitwareInc.com/OUs/Users/Information Technology/Vivian Atlas\Inbox      Inbox      5.195 KB (5,320 bytes)
LitwareInc.com/OUs/Users/Information Technology/Garrett Young\Inbox     Inbox      4.998 KB (5,118 bytes)
LitwareInc.com/OUs/Users/Information Technology/Lucio Iallo\Calendar    Calendar   2.748 KB (2,814 bytes)
LitwareInc.com/OUs/Users/Information Technology/Lucio Iallo\Sent Items  Sent Items 2.59 KB (2,652 bytes)
LitwareInc.com/OUs/Users/Information Technology/Vivian Atlas\Calendar   Calendar   2.057 KB (2,106 bytes)
```

We covered a tiny fraction of what's possible when using PowerShell to manage Exchange. If you haven't been using PowerShell to manage Exchange, I hope this tutorial has inspired you to start. If you'd like to dive deeper into PowerShell and Exchange I recommend checking out "Microsoft Exchange Server 2013 PowerShell Cookbook" by Mike Pfeiffer and Jonas Andresson. It's a learn-by-example book that gives you tons of recipes for making the most of PowerShell. ■



Get started with Windows PowerShell DSC

It's easy to automate Windows Server configuration management with PowerShell; here's how.

BY ADAM BERTRAM

In today's cloud-centric world, we're seeing an explosion in the number of servers under IT management. Virtual machines made servers cheap, and containers will push prices down further. As a result, businesses can afford to deploy a server for every new need, but they can no longer afford to manage servers individually. Your servers no longer garner individual attention but are simply soldiers in a huge resource pool, dutifully fulfilling the resource requests of the data center.

This dramatic increase in server population requires a new method of resource management, called configuration management. Products like Chef, Puppet, Ansible, Salt, and CFEngine have automated configuration management in the Linux world for many years now. It wasn't until recently that the companies behind these products started taking Windows seriously. This is in part due to Windows PowerShell Desired State Configuration (DSC). Introduced with PowerShell 4, DSC provides a way to manage Windows servers declaratively through the same configuration management practices the open source community has been using for years.

At first glance, you might think that DSC is a competitor to these trusted configuration management solutions. Technically, you can bring all of your servers under DSC management and forgo the other products. However, this is not Microsoft's intention, and in an environment of any size, you'd soon see the downfalls of such an attempt. Microsoft did not build DSC to be a competitor but to be used as a platform for others to build upon. DSC was built to provide a standardized method that *any* solution could use to manage Windows (and Linux) systems however they choose.

If you're looking into managing your Windows systems with a configuration management product, why would you choose to use DSC anyway? After all, it's a bare-bones platform. In fact, DSC is still

THINKSTOCK

Deep Dive

extremely valuable for smaller businesses that don't require features, such as reporting and Web management, that larger enterprises might require. One of the big advantages DSC has for smaller businesses is that it's built right into the operating system as of Windows Server 2012 R2. There's no need to download, deploy, and manage third-party software. It comes baked into Windows already.

What DSC is made of

DSC is built to manage configuration baselines using three main components: resource providers (Windows Server features, processes, files, registry keys, and so on), resources (the building blocks of configuration scripts), and a Local Configuration Manager (LCM), the DSC "client" or "agent" that applies configuration scripts on each target node. These components work in tandem to deliver a configuration to a particular server.

Resource providers are the "API" between the operating system and other services in which DSC tests, gets, and sets (if applicable) each configuration item on each server. Resource providers then consume DSC resources in the form of PowerShell scripts. Resources are the component of DSC that control the features, services, and applications installed on the operating system. Microsoft provides you with about a dozen kinds of resources built into Windows, including **`WindowsFeature`**, **`WindowsProcess,`** and **`File`**, to name a few.

```
PS C:\Windows\system32> Get-DscResource

ImplementedAs   Name                Module                          Properties
-------------   ----                ------                          ----------
Binary          File                                                {DestinationPath, Attributes, Checksum, Content...
Composite       InstallSNMPService                                  {Computername}
PowerShell      Archive             PSDesiredStateConfiguration     {Destination, Path, Checksum, DependsOn...}
PowerShell      Environment         PSDesiredStateConfiguration     {Name, DependsOn, Ensure, Path...}
PowerShell      Group               PSDesiredStateConfiguration     {GroupName, Credential, DependsOn, Description...}
Binary          Log                 PSDesiredStateConfiguration     {Message, DependsOn}
PowerShell      Package             PSDesiredStateConfiguration     {Name, Path, ProductId, Arguments...}
PowerShell      Registry            PSDesiredStateConfiguration     {Key, ValueName, DependsOn, Ensure...}
PowerShell      Script              PSDesiredStateConfiguration     {GetScript, SetScript, TestScript, Credential...}
PowerShell      Service             PSDesiredStateConfiguration     {Name, BuiltInAccount, Credential, DependsOn...}
PowerShell      User                PSDesiredStateConfiguration     {UserName, DependsOn, Description, Disabled...}
PowerShell      WindowsFeature      PSDesiredStateConfiguration     {Name, Credential, DependsOn, Ensure...}
PowerShell      WindowsProcess      PSDesiredStateConfiguration     {Arguments, Path, Credential, DependsOn...}
```

Figure 1: Built-in resources for Windows Server 2012 R2.

These are components that allow you to control the presence and configuration of Windows features, processes, files, and other components of the operating system. A downside of using native DSC is that only a few DSC resources come built in. This forces users to build their own resources in PowerShell. Users can download and consume custom-built resources from others in the PowerShell community as well.

```
PS C:\Windows\system32> Get-DscLocalConfigurationManager


AllowModuleOverwrite            : False
CertificateID                   :
ConfigurationID                 :
ConfigurationMode               : ApplyAndMonitor
ConfigurationModeFrequencyMins  : 30
Credential                      :
DownloadManagerCustomData       :
DownloadManagerName             :
RebootNodeIfNeeded              : False
RefreshFrequencyMins            : 15
RefreshMode                     : PUSH
PSComputerName                  :
```

Figure 2: An example of LCM properties.

The various resource providers take their marching orders from scripts managed by the LCM, which is in charge of such details as how often resource providers consume resources (in the form of PowerShell scripts) and how they are applied. The LCM is the mediator that controls the resource providers.

Lastly, resources can be delivered to resource providers in two ways: push and pull. Push mode operates in the traditional way you'd expect to run a script. You simply copy the resource to the server and execute it. Once the script is executed, the LCM sends the resource to the resource provider, which then follows the instructions in the resource.

The pull method entails a server downloading the resource from a central server running a small Web service known as a pull server. This pull server is in charge of storing all of the resources. Once a

Deep Dive

server requests all resources assigned to it, the pull server delivers the resource, the server pulls them down locally and executes them.

Most businesses choose to start out delivering resources via the push method. This allows them to build resources as they would PowerShell scripts and forgo building a pull server. However, push does not scale. It is recommended to build a pull server if you are deploying DSC to production in order to manage many servers.

Creating a DSC configuration

To manage a set of configuration items on a server first requires building a DSC configuration script. A DSC configuration script is a combination of various references to DSC resources that, when combined, form the desired state of a server.

DSC configuration scripts look similar to simple PowerShell modules and follow a domain specific language similar to that of Puppet. They also follow a similar naming convention with PowerShell functions; instead of the word "function" they use the word "configuration," for example. This makes it easy for people who already know PowerShell to start writing DSC modules.

```
1   Configuration InstallSNMPService
2   {
3       param (
4           [Parameter()]
5           [string]$Computername = 'localhost'
6       )
7
8       Node $Computername {
9
10          WindowsFeature SNMPService {
11              Name = 'SNMP-Service'
12              Ensure = 'Present'
13          }
14      }
15  }
```

Figure 3: A DSC configuration to install a Windows Server feature.

Configurations contain one or more resources. These resources can be built-in resources, like the `WindowsFeature` resource mentioned above, or custom resources, which allow you to incorporate any kind of PowerShell code you would like into the configuration. You can write custom resources yourself or draw on those created by the broader PowerShell community. Microsoft provides many resources that do not come installed by default in its GitHub repository. There you can find dozens of DSC resources like `xHyper-V`, `xCertificate`, and `xDnsServer` to use in your configurations.

Let's go over how to build your first DSC configuration. In this example, we will build the DSC configuration script to ensure a Windows server has the SNMP feature enabled. Because the DSC resource `WindowsFeature` comes built into Windows Server 2012, we don't have to download any external files. We will build this configuration on a Windows Server 2012 R2 server and execute it on the same machine. This means we will use the push mode, as our DSC client will not pull a configuration from an external server.

Deep Dive

Build the script

The first task is building the configuration script. To do this, we'll open up our editor of choice and get started.

In the first line you'll notice that we have a block called **Configuration**. This is how every configuration script must begin. The name of the script can be anything we'd like. Inside of that we'll typically have a parameter block with various parameters to the configuration. **$ComputerName** is a typical parameter, and you'll see that we're defaulting to the local computer here. Then, we have a Node block. This represents the name of the computer, or group of computers, that this configuration will be applied to. Since I'm specifying **$ComputerName** here, this script will build our configuration for the localhost machine. Next, we have the **WindowsFeature** block. This is a reference to the WindowsFeature DSC resource. I chose to give this reference a name of **SNMPService**, but it can be anything you'd like.

Inside of our DSC resource, we have two attributes: Name and Ensure. For the **WindowsFeature** resource, the name attribute signifies the name of the Windows feature. The Ensure attribute signifies that we want to ensure that this feature is present. We could just as easily choose to ensure that something isn't present, if that's what we wanted.

Create the MOF file

Once the configuration script is created, we then build a Managed Object Format (MOF) file. To do this requires executing our configuration block, which is as simple as referencing it in that same script. This will create a MOF file with the same name as the computer it will be run on in a subfolder with the same name as the configuration. You can see in Figure 4 below that executing our **InstallSNMPService** configuration created a subfolder of the same name with a localhost.mof file inside.

```
PS C:\Users\adam\Documents> InstallSNMPService


    Directory: C:\Users\adam\Documents\InstallSNMPService


Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         11/19/2015   2:24 PM       1096 localhost.mof
```

Figure 4: Creating the configuration MOF directory and file.

Once we have the MOF file generated, we can then call **Start-DscConfiguration** and specify the path to the folder we created. In the example below, I'm also using **–Verbose** to get more information as it applies the configuration and **–Wait** in order for the LCM to wait until the configuration has been applied. By default, LCM will create a PowerShell job and continue on before it is done.

Figure 5: Applying the DSC configuration.

```
PS C:\Windows\system32> Start-DscConfiguration -Path .\InstallSNMPService -Verbose -Wait
VERBOSE: Perform operation 'Invoke CimMethod' with following parameters, ''methodName' = SendConfigurationApply,'className' = MSFT_DSC
LocalConfigurationManager,'namespaceName' = root/Microsoft/Windows/DesiredStateConfiguration'.
VERBOSE: An LCM method call arrived from computer MEMBERSRV1 with user sid S-1-5-21-4117810001-3432493942-696130396-500.
VERBOSE: [MEMBERSRV1]: LCM:   [ Start  Set      ]
VERBOSE: [MEMBERSRV1]: LCM:   [ Start  Resource ]  [[WindowsFeature]SNMPService]
VERBOSE: [MEMBERSRV1]: LCM:   [ Start  Test     ]  [[WindowsFeature]SNMPService]
VERBOSE: [MEMBERSRV1]:                             [[WindowsFeature]SNMPService] The operation 'Get-WindowsFeature' started: SNMP-Servi
ce
VERBOSE: [MEMBERSRV1]:                             [[WindowsFeature]SNMPService] The operation 'Get-WindowsFeature' succeeded: SNMP-Ser
vice
VERBOSE: [MEMBERSRV1]: LCM:   [ End    Test     ]  [[WindowsFeature]SNMPService]  in 5.1250 seconds.
VERBOSE: [MEMBERSRV1]: LCM:   [ Start  Set      ]  [[WindowsFeature]SNMPService]
VERBOSE: [MEMBERSRV1]:                             [[WindowsFeature]SNMPService] Installation started...
VERBOSE: [MEMBERSRV1]:                             [[WindowsFeature]SNMPService] Prerequisite processing started...
VERBOSE: [MEMBERSRV1]:                             [[WindowsFeature]SNMPService] Prerequisite processing succeeded.
WARNING: [MEMBERSRV1]:                             [[WindowsFeature]SNMPService] Windows automatic updating is not enabled. To ensure t
hat your newly-installed role or feature is automatically updated, turn on Windows Update.
VERBOSE: [MEMBERSRV1]:                             [[WindowsFeature]SNMPService] Installation succeeded.
VERBOSE: [MEMBERSRV1]:                             [[WindowsFeature]SNMPService] successfully installed the feature SNMP-Service
VERBOSE: [MEMBERSRV1]: LCM:   [ End    Set      ]  [[WindowsFeature]SNMPService]  in 16.1470 seconds.
VERBOSE: [MEMBERSRV1]: LCM:   [ End    Resource ]  [[WindowsFeature]SNMPService]
VERBOSE: [MEMBERSRV1]: LCM:   [ End    Set      ]
VERBOSE: [MEMBERSRV1]: LCM:   [ End    Set      ]    in  22.0690 seconds.
VERBOSE: Operation 'Invoke CimMethod' complete.
VERBOSE: Time taken for configuration job to complete is 22.471 seconds
```

Once the configuration is applied, we can run the script again and you'll see that LCM simply skipped over the install since SNMP was already installed (Figure 6). The beauty of DSC is that you don't have to account for the Windows feature being installed or absent. DSC takes care of all

Deep Dive

that conditional logic for you. All you need to do is define how you want the state of the server to be -- that's it!

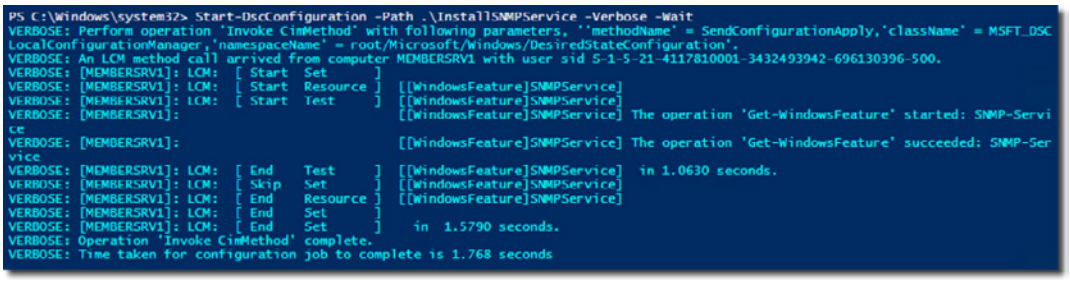


Figure 6: A subsequent run of the DSC configuration.

Once the configuration is applied, the LCM will take over and run again on its own every 15 minutes (by default). This means that once a configuration is set it will stay that way until you change the script. If someone should change the server's configuration manually, the LCM will soon set things right again. Thus DSC helps you to enforce standard configuration policies at regular intervals.

Building your own DSC resources

We've now covered building a configuration script using the built-in `WindowsFeature` resource. If the built-in resources don't cover all the functionality you're seeking, you can build custom DSC resources. If you choose to create your own, you'll need to know that DSC resources include three functions: `Get`, `Set`, and `Test`.

When the LCM runs, the resource provider first initiates the Test function to determine whether the configuration specified in the resource already matches the server's current configuration. If not, it then initiates the Get function to retrieve the current configuration. Depending on the LCM configuration, DSC will then either report that the configuration has drifted or will run the Set function to bring the server back into compliance.

With PowerShell 4, custom resources are created using MOF. On the plus side, MOF is a simple text file that could be easily consumed on the server. It is also an industry-standard format. In short, MOF was a simple way for Microsoft to get DSC shipped and immediately available to customers. Using MOF-based DSC resources, an admin goes through a four-step process:

1. Build the DSC module
2. Execute the DSC module to create a MOF file
3. The MOF file would get copied to a server somehow
4. The MOF file would be consumed by the DSC resource provider to check for the state of the server

This process works, but it is not the elegant method of resource consumption that Microsoft envisioned. As of PowerShell 5, Microsoft recommends creating custom resources via classes.

Classes are one of the big features introduced in PowerShell 5. Classes have been the mainstay of every object-oriented programming language, and now PowerShell can be considered part of the club. Classes have many uses in PowerShell 5, but one of the main use cases is class-based DSC resources. Using class-based resources, we no longer have to fool with MOF files anymore. Our previous four-step process can now be whittled down to three steps.

1. Build the DSC module
2. Copy the DSC module to the server
3. Execute the DSC module

Deep Dive

Building resources from classes is not only faster than using MOF files, but it also adheres more closely to how traditional PowerShell modules work. This makes it even easier for people to get started writing DSC resources.

Get to know DSC

If you're looking at configuration management options, it's well worth your time to take a look at DSC to manage your Windows servers. DSC has been around for only two years, but it is garnering ever-increasing support from Microsoft and others in the community. Keep a close eye on Microsoft's PowerShell GitHub repository, where you'll see contributors from Microsoft regularly fixing bugs, adding features, and creating new resources for everyone to use.

Still in its infancy, DSC is far behind products like Chef, Puppet, Ansible, and Salt. Even if you don't choose to use native DSC for configuration management, it's well worth your time to learn how DSC works.

The companies behind the leading configuration management solutions are only now beginning to adopt DSC in their products. You're soon going to find that every configuration management product that manages Windows systems is using DSC under the covers. Although you might not have to manage DSC directly, it will be important to understand how it works in order to better manage and troubleshoot the Windows configuration manager you do use. ■