The DSC Book

by Don Jones
and Steve Murawski
with Stephen Owen

The DSC Book

by Don Jones
and Steve Murawski
with contributions by Stephen Owen

Visit PowerShell.org to check for newer editions of this ebook.

PowerShell.org ebooks are works-in-progress, and many are curated by members of the community. We encourage you to check back for new editions at least twice a year, by visiting PowerShell.org. You may also subscribe to our monthly e-mail TechLetter for notifications of updated ebook editions. Visit PowerShell.org for more information on the newsletter.

Feedback and corrections, as well as questions about this ebook's content, can be posted in the PowerShell Q&A forum on PowerShell.org. Moderators will make every attempt to either address your concern, or to engage the appropriate ebook author.

About the Authors

Principal writing in this book was by Don Jones, President and CEO of PowerShell.org, Inc. and a multi-year recipient of Microsoft's MVP Award.

Additional writing, background information, and tech-checking were by Steve Murawski, possibly one of the earliest production adopters of DSC through his job at StackExchange, and one of Don's fellow PowerShell MVPs.

Several examples have been adapted, with permission, from http://foxdeploy.com/2014/02/25/desired-state-configuration-what-it-is-and-why-you-should-care/#more-474, and other excellent posts by Stephen Owen.

It's important for you to know that this guide is *very much a work in progress.* We appreciate feedback (use the PowerShell Q&A forum at PowerShell.org), and welcome additional contributors. Because we're treating this book as an open-source project, you may be reading it before a complete tech-check has been completed – so we appreciate your patience!

The Argument for DSC

Desired State Configuration is Microsoft's technology, introduced in Windows Management Framework 4.0, for declarative configuration of systems. At the risk of oversimplifying a bit, DSC lets you create specially formatted text files that describe how a system should be configured. You then deploy those files to the actual systems, and they magically configure themselves as directed. At the top level, DSC isn't programming – it's just listing how you want a system to look.

Microsoft is actually a little late to the game on declarative configuration game. The Linux and Unix world has had tools like Chef and Puppet, which at their very core perform a similar task, for a long time. What's important is that DSC isn't an add-on, it's a core part of the Windows OS, now. And while the toolset for DSC isn't yet as mature as tools like Chef and Puppet, DSC provides an inbuilt foundation that, in its v1 implementation, is incredibly robust and extensible.

DSC is the last major piece of functionality outlined in PowerShell inventor Jeffrey Snover's *Monad Manifesto,* a document in which he envisioned PowerShell itself, PowerShell Remoting, and a declarative configuration model. The implementation of that Manifesto has been happening since 2006, when PowerShell v1 was released to the public. For the most part (as you'll read in this guide), that implementation has been done in a way that's extremely cross-platform friendly. PowerShell Remoting, and even DSC, are based on open, third-party standards. With that standardized foundation, Microsoft can let other folks focus on tooling, knowing that those tools don't have to be Windows-dedicated since other operating systems can utilize these same technologies.

DSC is important – crucially important – to anyone who works with Windows servers. Even if that isn't the only OS in your environment, DSC is massively important, because it's going to gradually become *the only means by which you manage Windows-based servers.* That's a big statement, and it deserves some explanation.

> By the way, nothing in the following should be construed as an argument about Windows being better/same/worse as compared to any other OS. This is a discussion about Windows, in and of itself, without any of that being a value judgement or comparison.

The Evolution of Administration

In its beginning, Windows was administered mostly by graphical tools, and a few command-line ones, communicating directly with services on local or remote computers. You clicked a button in a dialog box, and the tool would talk to the Security Accounts Manager (SAM) and create a user account.

As environments grew large and IT teams smaller, folks needed to automate many of those processes. Graphical tools are inherently difficult to automate; the first phase in Snover's *Monad Manifesto* was a command-line shell that provided a consistent command-line language for automating administration. Command-line tools are inherently easy to automate, because they can be sequenced in simple batch or script files. A *consistent* and *all-encompassing* command-line language was something Windows had never had; management pressures within Microsoft were brought to bear to make PowerShell as all-encompassing as possible (an investment still underway). Guidelines were put into place to make it as consistent as possible (with largely positive results).

It's important to note that commands continued to communicate directly with services. Run **New-ADUser**, for example, and the command communicates directly with a service running on a domain controller to create the new user.

Scale-out was the next phase of the *Manifesto.* A command-line automation language is fine, but if you have to run around typing commands on individual computers, you still have an upper limit on efficiency. The ability to push commands to remote computers, in parallel, over a standardized communication protocol, was needed. PowerShell v2 delivered that in its Remoting feature. Now, one computer could easily tell a thousand others what to do, more or less at once. Remoting also reduced some of the burden on command developers. They could now assume that their commands always ran locally, on whatever computer was running their technology, and let PowerShell handle network communications. Not having to deal with communications made it significantly easier for command developers to pump out more commands, increasing PowerShell's technological "coverage."

But to this point, administrators still ran commands, and commands talked to services. Everything was very *imperative:* computers did what admins told them to do, more or less right then. If the environment changed, admins had to construct new scripts to implement the new changes. Admins then had to go back and make sure the changes *had been implemented* correctly, and continually re-check to make sure things *stayed that way.* If something got mis-configured, the fix was often a fallback to manual reconfiguration. You couldn't just run the same script, since it was designed to take a system from Point A to Point B, not from Point-A-Except-For-This-One-Thing to Point B.

DSC represents a significant break in administration, because it asks administrators to *no longer communicate directly with services on computers.* That means no graphical tools that talk directly to a service. It also means no running commands that talk directly to services. In other words, DSC asks administrators to *not actually configure anything themselves.*

Instead, DSC asks administrators to describe, in fairly simple text files, how they would like a computer to be configured. The computer, in turn, reads that text file, and configures itself accordingly. The configuration process is granular, so that if one item gets mis-configured, the computer can still bring itself "into compliance" by fixing that one thing.

DSC couldn't have existed without PowerShell, because PowerShell is what gives DSC the ability to implement changes. PowerShell is the one common language that all Microsoft products speak (to some degree or another, with constant improvement). In other words, without PowerShell, Microsoft products wouldn't have consistent enough automation capabilities for DSC to exist. Now that DSC does exist, administrators can actually focus less on PowerShell, because they'll be separated from it by a couple of less-complex layers.

Imagine a graphical tool where you click through a wizard to add a new user to your environment. Only that tool doesn't actually talk to a domain controller to create the user account, nor does it talk to a file server to create their home directory. Instead, the tool merely modifies a couple of text files that contain the configuration for a cluster of file servers, and for your domain. The file servers and domain controllers periodically re-read those files, and implement whatever they say. So, you click a button, and a few minutes later the user, and their home directory, magically exists. It's automated, but you didn't need to do any programming. The configuration files are where you, as an administrator, make your changes.

Growing the environment becomes simpler. "Hey, you are going to be a new domain controller. Go look at configuration file DCX34766578348 and make yourself look like what it says." Poof, done.

DSC represents a massive change in how Windows administrators think about their entire environment. Provided every configuration setting can be boiled down to a DSC setting – which will be true over time – then "administration" will essentially become "intelligent editing of text files." Pretty powerful stuff.

DSC Overview and Requirements

Desired State Configuration (DSC) was first introduced as part of Windows Management Framework (WMF) 4.0, which is preinstalled in Windows 8.1 and Windows Server 2012 R2, and is available for Windows 7, Windows Server 2008 R2, and Windows Server 2012. Because Windows 8.1 is a free upgrade to Windows 8, WMF 4 is not available for Windows 8.

You must have WMF 4.0 on a computer if you plan to author configurations there. You must also have WMF 4.0 on any computer you plan to manage via DSC. Every computer involved in the entire DSC conversation must have WMF 4.0 installed. Period. Check $PSVersionTable in PowerShell if you're not sure what version is installed on a computer.

> On Windows 8.1 and Windows Server 2012 R2, make certain that KB2883200 is installed or DSC *will not work.* On Windows Server 2008 R2, Windows 7, and Windows Server 2008, be sure to install the full Microsoft .NET Framework 4.5 package prior to installing WMF 4.0 or DSC may not work correctly.

To figure out what DSC is and does, it's useful to compare it to Group Policy. There are significant differences between the two, but at a high level they both set out to accomplish something similar. With Group Policy, you create a declarative configuration file called a Group Policy object (GPO). That file lists a bunch of configuration items that you want to be in effect on one or more computers. You target the GPO by linking it to domain sites, organizational units, and so on. Targeted machines pull, or download, the entire GPO from domain controllers. The machines use client-side code to read the GPO and implement whatever it says to do. They periodically re-check for an updated GPO, too.

DSC is similar… but not exactly the same. For one, it has no dependencies whatsoever on Active Directory Domain Services (ADDS). It's also a lot more flexible and more easily extended. A comparison is perhaps a good way to get a feel for what DSC is all about:

Feature	Group Policy	DSC
Configuration specification	GPO file	Configuration script (which produces a MOF file – more on those after this table)
Targeting machines	Link GPO to sites, OUs, etc.	Specify target nodes in the configuration script itself
Configuration implemented by	Client-side OS components	DSC *resources*, which are special PowerShell script modules
Extend the things that can be configured	Client-side GP extensions – usually written in native code, somewhat difficult to write	Simply write new DSC resources in PowerShell
Primary configuration target	Windows registry	Anything PowerShell can touch
Persistence	Settings "disappear" and re-applied each time for most GPO	Configuration changes are permanent (don't automatically

	settings	"undo" themselves)
Number of configurations	As many GPOs as you want	One MOF per computer

With DSC, you start by writing a *configuration script* in Windows PowerShell. This script doesn't *do* anything. That is, it doesn't install stuff, configure stuff, or anything else. It simply lists the things you want configured, and how you want them configured. The configuration also specifies the machines that it applies to. When you run the configuration, PowerShell produces a Management Object Format (MOF) file for each targeted machine, or *node.*

That's an important thing to call out: You (step 1) write a configuration script in PowerShell. Then you (step 2) run that script, and the result is one or more MOF files. If your configuration is written to target multiple nodes, you'll get a MOF file for each one. MOF stands for *Managed Object Format,* and it's basically a specially formatted text file. Then, (step 3), the MOF files are somehow conveyed to the machines they're meant for, and (step 4) those machines start configuring themselves to match what the MOF says.

In terms of conveying the MOF files to their target machines, there are two ways to do so: *push* mode is a more-or-less manual file copy, performed over PowerShell's Windows Remote Management (WinRM) remoting protocol; the *pull* mode configures nodes to check in to a special web server and grab their MOF files. Pull mode is a lot like the way Group Policy works, except that it doesn't use a domain controller. Pull mode can also be configured to pull MOF files from a file server by using Server Message Blocks (SMB; Windows' native file-sharing protocol).

> You'll see the term "node" instead of "computer" or "machine" a lot, because DSC envisions a time when you might be sending configurations to devices other than computers, or even to services running on computers. "Node" is just a bit more generic.

Once a node has its MOF file (and it's only allowed to have one; that's another difference from Group Policy, where you can target several GPOs to a single machine), it starts reading through the configuration. Each section in the configuration uses a *DSC resource* to actually implement the configuration. For example, if the configuration includes some kind of registry specification, then the registry DSC resource is called upon to actually check the registry and make the change if necessary.

You do have to deploy those DSC resources to your target nodes. In push mode, that's a manual task. In pull mode, nodes can realize that they're missing a resource needed by their configuration, and grab the necessary resource from the pull server (if you've put it there). For that reason, pull mode is the most flexible, centralized, and convenient way to go if you're managing a bunch of machines. Pull mode is something you can set up on any Windows Server 2012 R2 computer, and it doesn't even need to belong to a domain. If you're using the usual web server style of pull server (as opposed to SMB), you can configure either HTTP or HTTPS at your leisure (HTTPS merely requires an SSL certificate on the server).

In this guide, we're going to go through pretty much every aspect of DSC. The things we configure will be simple, so that we're not distracting from the discussion on DSC itself. This guide will evolve over time; if you notice blank sections, it's because those haven't yet been written. Errors, requests for more information, and so on should be reported in the PowerShell Q&A forum at PowerShell.org.

Why MOF?

You'll notice that DSC has a heavy dependency on MOF files, and there's a good reason for it.

The Managed Object Format (MOF) was defined by the Distributed Management Task Force (DMTF), a vendor-neutral industry organization that Microsoft belongs to. The purpose of the DMTF is to supervise standards that help enable cross-platform management. In other words, MOF is a cross-platform standard. That means a couple of important things:

- You don't necessarily have to write a PowerShell configuration script to produce a MOF file that DSC can use. So long as you give it a valid MOF file, DSC is happy, no matter who produced that MOF file. This opens up the possibility of using third-party management tools.

- Because a PowerShell configuration script produces standard MOF file, you can potentially write configuration scripts *that manage non-Windows computers.* Remember, your PowerShell script doesn't get sent to target nodes. The script produces a MOF, which is sent to target nodes. If the target node is a Linux computer that knows what to do with a MOF file, then you're good to go.

So the idea behind MOFs is to create a configuration management system that's cross-platform compatible. Existing configuration management systems in the Linux world (think Chef and Puppet, for example) already use MOFs the same way. So Microsoft isn't tying you to using their technology: you can manage Windows servers using anything that's capable of producing a valid MOF.

MOFs are also closely related to the Common Information Model (CIM), another DMTF standard that Microsoft originally implemented as Windows Management Instrumentation (WMI). The MOF format is used to define the classes that appear in the CIM repository – another Microsoft attempt to work and play well with others, since non-Windows computers can also implement a CIM repository that's cross-platform compatible.

It's important to understand that DSC consists of three layers:

- Top: The domain-specific language (DSL) that you use to write declarative configuration scripts. This is a PowerShell language subset, and PowerShell compiles the scripts into MOFs.

- Middle: The node-side functionality that accepts MOFs and accomplishes the configuration process.

- Bottom: The node-side DSC resources that are called on by the MOFs to actually perform configuration tasks.

The point of this is that you can swap out the top layer for *anything* capable of producing the right MOF – and that the MOF format is vendor-neutral and not Microsoft-proprietary. Since anyone can provide elements of the bottom layer (and Microsoft will provide a lot), *any* management tool can leverage DSC. So if you've got a cross-platform management tool, and it can produce the right MOFs, then you don't necessarily need that tool's agent software installed on your computers. Instead, the tool can produce MOFs that tell the Windows DSC components what to do.

Where to Find Resources

WMF 4.0 ships with some basic DSC resources. As time goes on, it's likely that many Microsoft server products will include their own DSC resources for configuring those products. Also, Microsoft is releasing "out of band" waves of additional resources via the web.

- Wave 1: http://blogs.msdn.com/b/powershell/archive/2013/12/26/holiday-gift-desired-state-configuration-dsc-resource-kit-wave-1.aspx

- Wave 2: http://blogs.msdn.com/b/powershell/archive/2014/02/07/need-more-dsc-resources-announcing-dsc-resource-kit-wave-2.aspx?utm_source=tuicool

You'll find some open-source community resources at http://github.com/powershellorg/dsc, and a Google or Bing search will likely turn up other folks' DSC projects.

A word on naming: the Microsoft "waves" of resources all have resource names that start with the letter "x," which stands for "experimental, unsupported, use at your own risk." Keep in mind that these are *just PowerShell script modules,* which means they're basically open-source. Microsoft has asked that any community derivatives or original work carry the letter "c" as a resource name prefix. The idea is that non-prefixed names (e.g., "WindowsFeature") is reserved for Microsoft. By not using non-prefixed names, you will avoid coming up with a resource that conflicts with later Microsoft releases. Within your own organization, adopt a resource name prefix that's unique to you, like "contosoBusinessApp" for a "BusinessApp" resource owned by Contoso.

> Keep in mind that you need resources on whatever computer you're using to author configurations, as well as on whatever nodes will implement those configurations.

Microsoft-Provided Resources

Microsoft provides the following resources in WMF 4:

- Registry
- Script (runs custom scripts; kind of a catch-all when there isn't a resource to do what you want)
- Archive (zip and unzip files)
- File
- WindowsFeature
- Package (install MSI or Setup.exe)
- Environment
- Group (local groups)
- User (local users)
- Log (logging mechanism for troubleshooting DSC)
- Service
- WindowsProcess

DSC Resource Kit Wave 1 included the following:

- xComputer (renaming and domain join)
- xVHD (superceded by Wave 2 version)
- xVMHyperV (create VMs)
- xVMSwitch

- xDNSServerAddress (binding DNS addresses to NICs, not managing DNS)
- xIPAddress
- xDSCWebService (deploys a pull server)
- xWebsite (superceded by Wave 2 version)

DSC Resource Kit Wave 2 included the following:

- xADDomain
- xADDomainController
- xADUser
- xWaitForADDomain (pauses configuration until a domain is available)
- xSqlServerInstall
- xSqlHAService (SQL high availability)
- xSqlHAEndpoint
- xSqlHAGroup
- xWaitForSqlHAGroup
- xCluster
- xWaitForCluster
- xSmbShare
- xFirewall
- xVhdFile (files copied to a VHD image)
- xWebsite
- xVhd

There isn't really a single download for the DSC Resource Kit; as of this writing, each resource is a separate download from the TechNet Script Gallery.

> Please don't rely on this guide to be an ongoing, reliable directory of Microsoft-released DSC resources. You'll need to do some research to decide if there are later waves, bug fixes, or community versions of these resources. We won't be updating this portion of the guide constantly.

Where to Put Resources

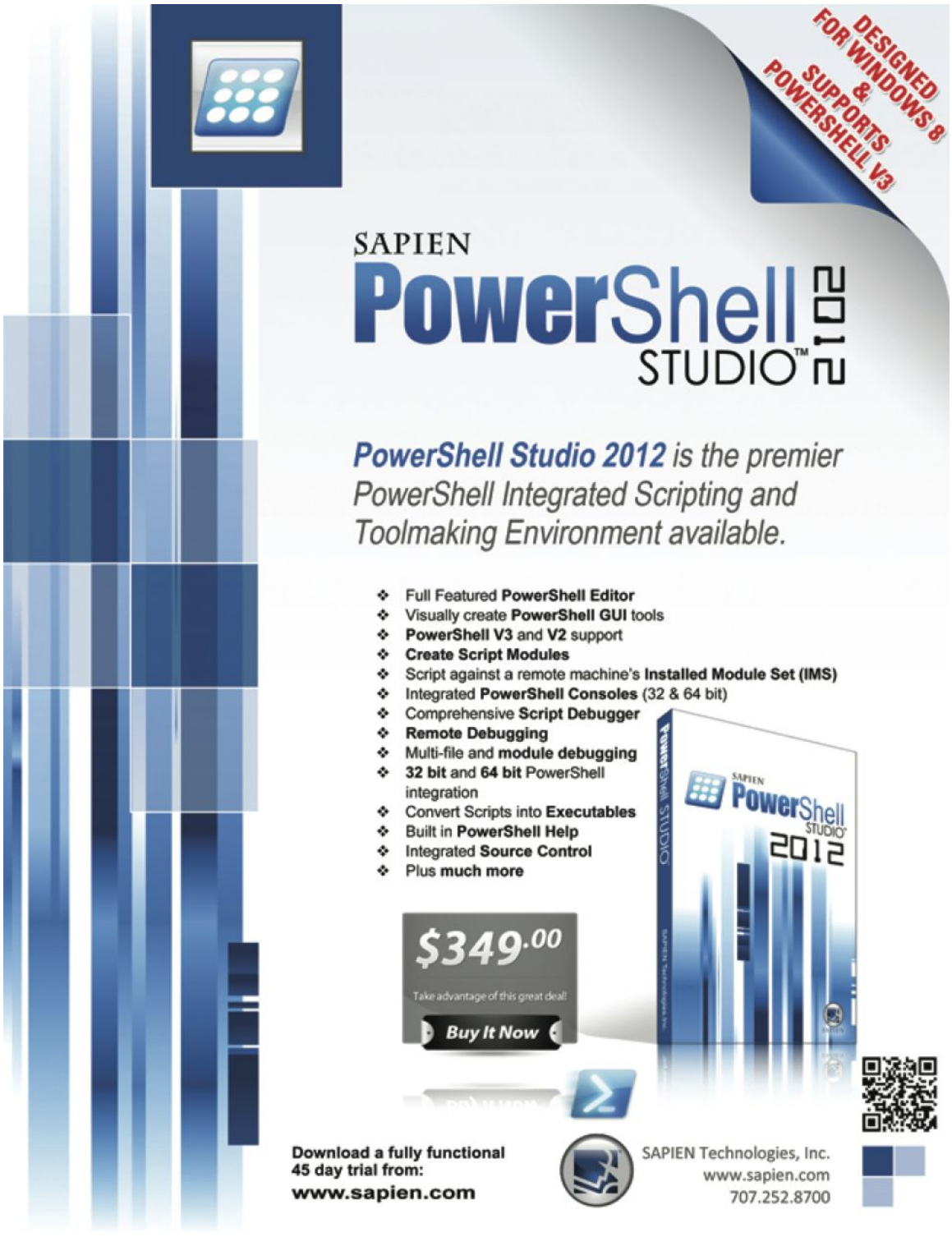
We'll discuss it more later in this guide, but in general you should save new resources in \Program Files\WindowsPowerShell\Modules. Each resource consists of a *root module,* and then has a DSCResources sub-folder for the actual working code. It's possible for a single root module to actually offer more than one named resource.

For example, suppose you have a DSC resource module named CorpApp. You would create the following folder:

```
\Program Files\WindowsPowerShell\Modules\CorpApp
```

The entire module would go into that folder. The root module file will be named something like CorpApp.psd1, and there will be a DSCResources sub-folder that contains additional script files for the actual resources. There's a whole section in this guide about deploying resources, so you'll find more detail there.

Note that you'll need to install resources on any computer where you plan to author configurations using PowerShell, and on any nodes that will use those resources in their configurations.



Writing Configuration Scripts

A configuration script consists of three main items:

- The main **configuration** container, which contains everything else
- One or more **node** sections that specify specific target computers and contain the configuration items
- One or more **configuration items** per node section, which specify whatever configuration you want to be in effect on those nodes

The PowerShell ISE understands configuration syntax, and is a good editor. You do need to be careful with it: when you run a configuration, it's going to need to create a folder and one or more MOF files. It'll need to do that in whatever folder the ISE is currently pointed at. So before you begin writing, you may want to switch to the ISE's console pane and change directories. Otherwise, you may end up writing files to \Windows\System32, which might not be desirable.

Here's a basic configuration script, which you might save as MyConfig.ps1:

```
Configuration MonitoringSoftware
{
   param(
     [string[]]$ComputerName="localhost"
  )
      Node $ComputerName
      {
         File MonitoringInstallationFiles
               {
                    Ensure          ="Present"
                    SourcePath      ="\\dc01\Software\Monitoring"
                    DestinationPath="C:\Temp\Monitoring"
                    Type            ="Directory"
                    Recurse         =$true
               }
      }
}
MonitoringSoftware
```

There are several important things happening here:

- The top-level **Configuration** construct acts a lot like a PowerShell function. It contains the entire configuration, and it's actually considered a kind of PowerShell command. The name of the configuration, **MonitoringSoftware,** is arbitrary – you can name it whatever you like.

- The configuration accepts a **–ComputerName** parameter, which defaults to localhost. So, when you run the configuration, you specify the computer name that you want it to apply to. This is one alternative to hardcoding a list of computer names into the configuration. Note that, like parameters in functions, configuration parameters can have a [Parameter()] attribute and various validation attributes. These can be used to validate parameter input, make parameters mandatory, and so on. Read the about_functions_advanced_parameters help file in PowerShell for more information.

- The **Node** element specifies the computer(s) that this configuration will apply to. Here, we're inserting the value of the –ComputerName parameter.

- The **File** keyword refers to a specific DSC resource – the File resource that ships with WMF 4. This configuration item has a name, **MonitoringInstallationFiles**, that you can make up. We

could have called it **Fred**, for example, and it would still work.

- The configuration item includes 5 settings: Ensure, SourcePath, DestinationPath, Type, and Recurse. These are specific to the **File** DSC resource; other resources would have different settings. In this case, we're asking DSC to ensure that a specific folder hierarchy is present on the target node. If it isn't, we've told DSC where to find it, so that it can copy it over. You could also copy individual files, if you needed to.

A configuration is a command, so the last line of the script actually runs the command. We don't provide a –ComputerName parameter, so it'll default to creating a MOF file for localhost. We'll actually get a folder named **MonitoringSoftware** (the name of the configuration), and it'll contain **Localhost.mof**.

Notice that the –ComputerName parameter is declared as **[string[]]**, meaning it accepts multiple values. We could provide multiple values, and generate a MOF for each. For example:

```
MonitoringSoftware –ComputerName (
  Get-ADComputer –Filter * -SearchBase "ou=clients,dc=domain,dc=pri" |
  Select-Object –Expand Name
)
```

That would produce a MOF file for each computer in the Clients OU of the Domain.pri domain.

Multiple Nodes in a Configuration

Most online examples you see will only include a single Node section inside the Configuration section. But there's a valid scenario to include multiple nodes in a configuration. For example, suppose you're setting up a lab environment that includes several servers, a few clients, and so on. You could include *every computer in the lab* inside one configuration script, giving you a single source for managing the entire configuration. When you run the configuration, PowerShell will produce a MOF for each node that's mentioned inside. We've included an example of a multi-node configuration later in this guide.

Discovering Resources

The above example should bring up one, or maybe two questions:

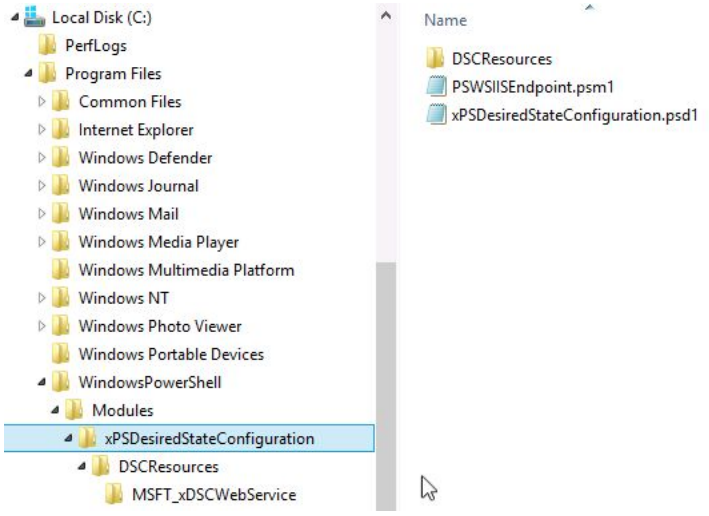
- How do I know what resources I have installed?
- How do I know what settings a DSC resource requires?

Both are excellent questions, and easily answered. Start by running **Get-DSCResource** (if the command doesn't work, you're probably not running PowerShell v4). That will list all resources that are loaded into memory, which will include any that *were shipped with Windows.* The output of the command includes the **Name** that you use to refer to the resource in a configuration. Once you've narrowed down a resource you want – let's say, "Package," – you can get more details about it. Run **Get-DSCResource –Name Package | Select –Expand Properties** (swapping out "Package" for the name of the resource you're interested in). You'll get a complete list of properties, and the type of data each one expects. In some cases, you'll see allowable values. For example, the "Ensure" property usually accepts "Absent" and "Present," depending on whether you want the item there, or want it to *not* be there.

Another option, using the **File** module as an example, would be to run **Get-DSCResource –Name File –Syntax**. This produces similar output, showing you all of the resource's settings, the kind of value they expect, and also including accepted values for some settings.

Beyond that, PowerShell doesn't necessarily offer a ton of documentation on every resource, so you may end up doing some Googling, but at least Get-DSCResource provides you with a good starting point.

Things get tricky with resources that you've added yourself. Although they're technically packaged as PowerShell script modules, they don't behave quite like modules, so you have to take a few extra steps. First of all, they should go into \Program Files\WindowsPowerShell\Modules. Here's an example folder hierarchy for a single DSC resource:



Here's what you're looking at:

- The *module name* is **xPSDesiredStateConfiguration**. That folder contains a script module and a module manifest – although not every resource you look at will have all of that. There's also a **DSCResources** subfolder.

- The **DSCResources** subfolder doesn't contain any files. It contain a subfolder for each resource that the module provides.

- The **MSFT_xDSCWebService** subfolder represents a single DSC resource. It contains, at a minimum, a .psm1 script module file and a .schema.mof file. Both are required to use the resource.

Things get a little tricky because you don't really import this into the shell as a normal module. There's an **Import-DSCResource** keyword, but it only works inside Configuration scripts – you can't use it right at the shell prompt. So part of your "finding what resources I have" process is going to necessarily involve browsing the file system a bit to see what's installed.

Sequencing Configuration Items

It's possible that configuration items might need to happen in a particular order. For example, suppose you're installing an application on a computer. You might need to first copy the install source to the computer, *then* run the installer. That's easy to do, as demonstrated in this configuration item:

```
Package MonitoringSoftware
{
    Ensure     ="Present"
    Path       ="$Env:SystemDrive\Temp\Monitoring\7z920-x64.msi"
    Name       = "7-Zip"
    ProductId  = "23170F69-40C1-2702-0920-000001000000"
    DependsOn  = "[File]MonitoringInstallationFiles"
```

```
}
```

Notice the **DependsOn** setting? That setting is common to most resources, and in this case it's saying, "hey, go to the **File**-type setting named **MonitoringInstallationFiles** before you do me." In this case, that makes sure the application's installer has been copied over before the Package resource is called upon to actually run it.

By the way, the **ProductID** setting isn't something to worry about for the purposes of understanding DSC. That's a setting that's unique to the **Package** resource. It contains the unique identifier for the particular piece of software (7-Zip, in this example) we're asking DSC to install. The **Package** resource probably uses the product ID to check and see if the thing is already installed or not. The point is that *it's necessary for the Package resource to work,* but it doesn't have anything to do with DSC per se.

Similarly, **Path** and **Name** are other settings required by the **Package** resource. The **Path** setting lets you specify the path to the installer, so that DSC knows where to find it if the software isn't already installed on the target node. In this case, we're pointing to a local file path – so we need to make darn well sure that the MSI file exists in that location. That's why this configuration items **DependsOn** our earlier **File** configuration item. The **File** item copies the MSI file over, so that the **Package** item can run the MSI if needed.

Forcing a Configuration Evaluation

There's a quick trick to force a computer to re-evaluate its configuration. If configured to pull its config, this will also force the pull to happen. Just run this on the computer:

```
$params = @{
    Namespace  = 'root/Microsoft/Windows/DesiredStateConfiguration'
    ClassName  = 'MSFT_DSCLocalConfigurationManager'
    MethodName = 'PerformRequiredConfigurationChecks'
    Arguments  = @{
        Flags = [uint32] 1
    }
}

Invoke-CimMethod @params
```

Pushing Configurations

Conceptually, *pushing* configurations is the easiest to talk about, so we'll start there. It also requires the least setup in most environments. There's really only one prerequisite, which is that your target nodes must have PowerShell remoting enabled. If they don't, you should read through *Secrets of PowerShell Remoting,* another free ebook at http://PowerShell.org/wp/newsletter. In a domain environment, running **Enable-PSRemoting** on the target nodes is a quick and easy way to get remoting enabled. It's pre-enabled on Windows Server 2012 and later server operating systems.

> As of this writing, Remoting isn't enabled on any client computers by default. So if you plan to push a configuration to a client – even if you've authored the configuration right on that client – you first need to enable remoting.

So, let's say you've written your configuration script, and run it to produce MOF files. All you need to do now is run:

```
Start-DSCConfiguration –Path c:\windows\system32\MonitoringSoftware
```

Remember, when we created the sample configuration, we named it **MonitoringSoftware**. Running it will create a folder named **MonitoringSoftware**, with one MOF file per targeted node. So **Start-DSCConfiguration** only needs that path – it'll figure out the rest on its own. Add the **–Verbose** switch to get blow-by-blow progress reports as the command runs. This command will copy the MOF file to the targets, and tell their Local Configuration Manager (LCM) software to start processing the configuration.

> By default, you need to be an administrator to use Remoting – and so you must be an administrator to run Start-DscConfiguration. On a client with User Account Control (UAC) enabled, make sure you see the word "Administrator" in the Windows PowerShell window title bar. If you don't, you're not really Administrator.

DSC push happens by default as a PowerShell remoting job, and has a –ThrottleLimit parameter that controls how many machines it communicates with in parallel. That defaults to 32; each simultaneous connection puts a bit of extra memory and CPU overhead on the machine where you run the command, so on a well-equipped machine you can safely bump that up. All the more reason to get a quad-core Xeon box with 64GB of RAM as your next admin workstation, right?

What this command will *not* do is take care of deploying any resources that your configuration requires. That's on you, and it's pretty much a manual process if you're using push mode. Here's why: the target machine will attempt to evaluate the *entire* MOF file. If there are *any* resources in that MOF file that aren't present, then the evaluation will fail – meaning the *entire* configuration will fail. This is one reason that pull mode can be so attractive, since with pull mode the target node can copy over any resources it needs from the pull server.

Configuring an HTTP(S) Pull Server

There are bunches of ways to create a DSC pull server. The easiest is probably to use the resources Microsoft provided in their DSC Resource Kit Wave 1 and Wave 2 web releases, since they provide resources that can configure a pull server.

For this walkthrough, we used a Windows Server 2012 R2 computer. We downloaded the xPSDesiredStateConfiguration module from http://gallery.technet.microsoft.com/xPSDesiredStateConfiguratio-417dc71d; it was up to version 1.1 when we got it. This is actually a module, and it contains the **xDSCWebService** resource, so that's what we'll refer to in the configuration we create.

Any server that has WMF 4.0 installed can be a pull server; be sure you're not installing preview bits of WMF 4.0, though.

Remember, a *module* can contain one or more *resources.* Once you've installed the module in the proper location, you don't necessarily need to know the module name anymore. You just need to know the names of the resources that the module contains, and **Get-DSCResource** can provide you with the resource names. However, if you install the module elsewhere, you'll need to use **Import-Module** to import the module, thus making the resources available to PowerShell.

Setting Up the Pull Server

We unzipped the contents of the download to Program Files\WindowsPowerShell\Modules on our Windows Server 2012 R2 computer. The computer is running server core, and to begin with was a completely fresh install of the OS. We also copied the module to the same folder on a Windows 8.1 computer. We plan to write the configuration there, and run it to produce a MOF, and so wanted to make sure everything is in place.

Because the DSC resource modules are script modules, we needed to ensure that the execution policy on the Windows 8.1 computer was at least RemoteSigned. Windows Server 2012 R2 defaults to that setting.

Following the example on the module's download page, we used the Windows 8.1 computer, and the PowerShell ISE, to create the following configuration file:

```
configuration CreatePullServer
{
    param
    (
        [string[]]$ComputerName = 'localhost'
    )
```

```
    Import-DSCResource -ModuleName xPSDesiredStateConfiguration

    Node $ComputerName
    {
        WindowsFeature DSCServiceFeature
        {
            Ensure = "Present"
            Name   = "DSC-Service"
        }

        xDscWebService PSDSCPullServer
        {
            Ensure                  = "Present"
            EndpointName            = "PSDSCPullServer"
            Port                    = 8080
            PhysicalPath            = "$env:SystemDrive\inetpub\wwwroot\PSDSCPullServer"
            CertificateThumbPrint   = "AllowUnencryptedTraffic"
            ModulePath              = "$env:PROGRAMFILES\WindowsPowerShell\DscService\Modules"
            ConfigurationPath       = "$env:PROGRAMFILES\WindowsPowerShell\DscService\Configuration"
            State                   = "Started"
            DependsOn               = "[WindowsFeature]DSCServiceFeature"
        }

        xDscWebService PSDSCComplianceServer
        {
            Ensure                  = "Present"
            EndpointName            = "PSDSCComplianceServer"
            Port                    = 9080
            PhysicalPath            = "$env:SystemDrive\inetpub\wwwroot\PSDSCComplianceServer"
            CertificateThumbPrint   = "AllowUnencryptedTraffic"
            State                   = "Started"
            IsComplianceServer      = $true
            DependsOn               =
("[WindowsFeature]DSCServiceFeature","[xDSCWebService]PSDSCPullServer")
        }
    }
}

CreatePullServer -ComputerName pull1.lab.pri
```

> Our copy of the ISE was acting a little wacky and not fully recognizing the script syntax at first. A reboot solved the problem. However, if the ISE is complaining about the **Import-DSCResource** command, make sure KB2883200 is installed. Remember too that you can only run these scripts if PowerShell 4 is installed.

Notice that this file creates a pull server as well as a compliance server (more on that toward the end of this guide). Each is basically just a website under IIS, running on different ports. Both are configured to use unencrypted HTTP, rather than an SSL certificate. We think that's probably going to be a common configuration on private networks, although HTTPS does provide authentication of the server, which is nice to have.

The configuration includes three items, with each successive one depending on the previous one. The first installs the DSC Pull Server feature, the second sets up the pull server website, and the third sets up the compliance server website. We've used paths that are more or less the defaults.

We saved the script as C:\dsc\PullServerConfig.ps1. We then ran it, resulting in the creation of C:\dsc\CreatePullServer. Notice that the new subfolder name matches the name we gave our configuration in the script. In that folder, we found Pull1.lab.pri.mof, the MOF file for server PULL1. That's the name of the Windows Server 2012 R2 computer that we plan to turn into a pull server.

Next, we ran **Start-DscConfiguration .\CreatePullServer -Wait**. Notice that we gave it *the path where the MOF files live*; it then enumerates the files in that path and starts pushing the MOF files to their target nodes. Because we used –Wait, the command runs right away and not in a job. That's often the best way to do it when you're first starting, since any errors will be clearly displayed. If you don't use –Wait, you'll get back a job object and it can be a bit difficult to track down any errors.

You shouldn't get any errors. If you did, triple-check that KB2883200 is installed on the target node. You will likely get a warning if Windows automatic updating isn't enabled – that's just reminding you to run Windows Update to make sure the components you just installed are completely up-to-date.

Creating a Configuration to Be Pulled

Our next step is to create a sample configuration. We'll store this on the new pull server, and then configure another server to go grab it. On the same Windows 8.1 computer, we created the following and saved it as WindowsBackupConfig.ps1:

```
Configuration WindowsBackup {
    Node member2.lab.pri {
           WindowsFeature Backup {
                  Ensure = 'Present'
                  Name   = 'Windows-Server-Backup'
           }
    }
}

WindowsBackup
```

This is obviously not going anything incredibly magical. It's just telling a computer, MEMBER2, to make sure Windows Backup is installed. So now we need to run it and create the appropriate MOF. We do that by simply running the script. It'll create a folder named **WindowsBackup** (since that's the name of the configuration), and in there will be a MOF file for **Member2.lab.pri**, the node we're targeting.

Now we need to get that MOF to the pull server. Thing is, we also need to rename the file, because the node name isn't what the pull server wants in the filename. Instead, the pull server wants the MOF's filename to be a globally unique identifier (GUID). If you look back at the configuration we used to create the pull server, we said that it should store configurations in $env:PROGRAMFILES\WindowsPowerShell\DscService\Configuration. So that's where the renamed MOF needs to go.

First, create the GUID:

```
$guid = [guid]::NewGuid()
```

Now, copy the file:

```
$source = "WindowsBackup\member2.lab.pri.mof"
$dest = "\\pull1.lab.pri\c`$\program files\windowspowershell\dscservice\configuration\$guid.mof"
copy $source $dest
```

Notice the cute trick with double quotes to insert the GUID into the destination path, and the backtick to keep PowerShell from treating $\ as a variable name. Also notice that our pull server is named Pull1.lab.pri.

With the MOF in place, we now need to generate a checksum file for it on the pull server. Computers that attempt to pull the configuration use the checksum to ensure the integrity of the file.

```
New-DSCChecksum $dest
```

This of course relies on the $dest variable we created earlier, and should result in a .checksum file being created in the same folder as the MOF.

> If you modify your configuration, you must delete the old checksum file and create a new one. If you don't, the new configuration will never be pulled by any server that already pulled the old version.

Telling a Computer to Pull the Configuration

By default, Windows computers wait for you to push a configuration to them, and then they re-check it every 15 minutes. We're going to reconfigure a computer named Member2.lab.pri to instead pull its configuration from our pull server, using the default refresh interval of 30 minutes. To do that, we need to use our Windows 8.1 computer to quickly create another configuration file:

```
Configuration SetPullMode
{
    param([string]$guid)
    Node member2.lab.pri
    {
            LocalConfigurationManager
            {

                    ConfigurationMode = 'ApplyOnly'
                    ConfigurationID = $guid
                    RefreshMode = 'Pull'
                    DownloadManagerName = 'WebDownloadManager'
                    DownloadManagerCustomData = @{
                            ServerUrl = 'http://pull1.lab.pri:8080/PSDSCPullServer.svc';
                            AllowUnsecureConnection = 'true' }
            }
    }
}

SetPullMode –guid $guid
Set-DSCLocalConfigurationManager -Computer member2.lab.pri -Path ./SetPullMode –Verbose
```

There are a few important things to notice, here.

- **LocalConfigurationManager** isn't a DSC resource, per se – and notice that the configuration item doesn't have a name. This is modifying the Local Configuration Manager on the target – the bit that runs DSC.

- When run, this will create a folder named **SetPullMode**, which will contain **Member2.lab.pri.meta.mof**. The "meta" portion of the filename indicates that this is configuring the target's Local Configuration Manager (LCM), rather than configuring something else on the target.

- At the end of the script, we're actually running the command *and* pushing the MOF file out to the target node.

- The configuration accepts a GUID as a parameter. When we run the configuration, at the end of the script file, we're passing in the $guid variable from earlier. Also notice that the $guid gets used as the **ConfigurationID** setting's value. This is how we tell the target node which configuration it should be pulling from the pull server.

- The **ServerUrl** is the pull server's FQDN. We specified port 8080, because that's the port where we set up the pull server – go back and look at the pull server configuration script to see where that is.

- We're using the WebDownloadManager because the pull server is providing service over HTTP. That's different than a pull server that provides service over file sharing – we'll tackle that next.

Running this script produces the MOF, and then pushes it to the target node. Very quickly, the target node will contact the pull server, grab its configuration, and then start evaluating the configuration. Before we run this script, then, let's quickly check the status of the Windows Backup feature on Member2:

```
PS C:\> Enter-PSSession member2.lab.pri
[member2.lab.pri]: PS C:\Users\Administrator.LAB\Documents> cd \
[member2.lab.pri]: PS C:\> Get-WindowsFeature -Name Windows-Server-Backup

Display Name                                            Name                       Install State
------------                                            ----                       -------------
[ ] Windows Server Backup                               Windows-Server-Backup          Available


[member2.lab.pri]: PS C:\>
```

Now we run the script that configures Member2 to pull:

```
    Directory: C:\dsc\SetPullMode


Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---         2/28/2014   1:11 PM       1858 member2.lab.pri.meta.mof
VERBOSE: Performing the operation "Start-DscConfiguration: SendMetaConfigurationApply" on target "MSFT_DSCLocalConfigurationManage
VERBOSE: Perform operation 'Invoke CimMethod' with following parameters, ''methodName' = SendMetaConfigurationApply,'className' =
rationManager,'namespaceName' = root/Microsoft/Windows/DesiredStateConfiguration'.
VERBOSE: An LCM method call arrived from computer WIN81 with user sid S-1-5-21-3762870595-1619700840-2352719374-500.
VERBOSE: [MEMBER2]: LCM:  [ Start  Set      ]
VERBOSE: [MEMBER2]: LCM:  [ Start  Resource ]  [MSFT_DSCMetaConfiguration]
VERBOSE: [MEMBER2]: LCM:  [ Start  Set      ]  [MSFT_DSCMetaConfiguration]
VERBOSE: [MEMBER2]: LCM:  [ End    Set      ]  [MSFT_DSCMetaConfiguration]  in 0.0780 seconds.
VERBOSE: [MEMBER2]: LCM:  [ End    Resource ]  [MSFT_DSCMetaConfiguration]
VERBOSE: [MEMBER2]: LCM:  [ End    Set      ]
VERBOSE: Operation 'Invoke CimMethod' complete.
VERBOSE: Set-DscLocalConfigurationManager finished in 0.399 seconds.


PS C:\dsc>
```

Now we may need to wait just a smidge. By default, this "pull" thing happens every half-hour. But eventually, it kicks in and works, installing Windows Backup for us.

One Config, Many Nodes

It's perfectly legal to configure multiple computers' LCMs to have the same ConfigurationId. When you do so, they'll all be pulling the exact same configuration MOF from the pull server. That's an appropriate scenario when you have computers that need an identical configuration – say, web servers in a web farm, for example.

Think about it! You deploy a new virtual machine, configure it's LCM, and you're done. The server grabs the MOF from the pull server and, provided you put the complete configuration into that MOF, the server completely sets itself up. It installs roles like IIS, copies content from a file server, whatever.

Or, let's say you have several web servers configured identically – and you need to make a change. No problem! Modify your configuration, run it to produce the MOF, and put the MOF on the pull server. Within half an hour (by default) every node will re-check the MOF, grab the new one, and start implementing the new config.

What About HTTPS?

Making an HTTPS pull server is really no different. In the pull server configuration, instead of this:

```
            CertificateThumbPrint    = "AllowUnencryptedTraffic"
```

You'd provide the thumbprint of a valid SSL certificate. Then, in the LCM configuration for the target nodes, instead of this:

```
DownloadManagerCustomData = @{
    ServerUrl = 'http://pull1.lab.pri:8080/PSDSCPullServer.svc';
    AllowUnsecureConnection = 'true' }
```

You'd omit the **AllowUnsecureConnection**, or set it to False. That will trigger the client to use HTTPS. It's important that the client already trust the Certificate Authority (CA) that issued your server's SSL certificate, and that the certificate already be installed in the local machine store of the pull server. That's an easy detail to miss: normally, if you just double-click a certificate file to install it, the certificate ends up in your *personal* store, which does no good. Use the Certificates console to install the certificate, so that you can install to the machine store instead. You can get a directory of the CERT: drive in PowerShell to list stores and certificates, and to obtain a certificate's thumbprint.

Deploying Modules via Pull Server

A machine that's pulling its configuration can also pull any DSC resources that the configuration refers to, but which aren't present on the local machine already. There are some tricks for setting this up – there's a whole section, later in this guide, about how to do it. It isn't as easy as just dumping the file in the right place!

Configuring an SMB Pull Server

Now that we've covered setting up a pull server that uses HTTP(S), let's talk about setting up a pull server that uses simple file copying – that is, Server Message Blocks (SMB).

Why SMB Instead of HTTP(S)?

SMB pull servers are a lot easier to set up. For a purely intranet environment, they might be all you need, and the "pull server" can simply be any old file server on the network. However, if you have computers on the other side of a firewall, SMB is a lot harder to get through firewalls than HTTP, because SMB uses a range of ports, along with endpoint mapping.

Also, only HTTP(S) supports the use of a *compliance server,* which is what pull clients report their current configuration status to. More than that toward the end of this guide.

Setting Things Up

We're going to assume you've read through the HTTP(S) pull server setup, and just point out the major differences involved in using SMB instead.

First, you need to make sure that target nodes are configured to use the pull server. In the LCM configuration, instead of this:

```
DownloadManagerName = "WebDownloadManager"
DownloadManagerCustomData = @{
    ServerUrl = 'http://pull1.lab.pri:8080/PSDSCPullServer.svc';
    AllowUnsecureConnection = 'true' }
```

You'd include something like this:

```
DownloadManagerName = "DscFileDownloadManager"
DownloadManagerCustomData = @{
    SourcePath = "\\PULL1\PullShare" }
```

This tells the target node to get its configuration from the \\PULL1\PullShare shared folder. Easy enough. You don't actually have to go through any complex setup on the pull server; you just create the specified shared folder.

Drop your MOF files (which must have a *GUID*.mof filename) and their checksum files (*GUID*.mof.checksum; use New-DscChecksum to generate checksum files) into the shared folder. Boom, you're done.

An SMB pull server can also include custom resources. As we discuss in the "Deploying Resources" section of this guide, you have to ZIP up the resource modules, name them according to convention, and provide checksum files. But then you just drop those ZIP and checksum files right into the pull server shared folder – nothing more to do.

Load-Balancing and High Availability for Pull Servers

Since pull servers are either just web servers or file servers, it's pretty easy to add load balancing and high availability.

For example, for an SMB pull server, you could use Distributed File System (DFS) to create a UNC path that points to multiple DFS replicas on different file servers, and then use DFS Replication (DFS-R) to ensure each replica contained the same configuration files and ZIPped modules.

For an HTTP(S) pull server, you would create a "farm" of web servers, each configured identically. You could load balance between them using a hardware or software load balancer, Windows Network Load Balancing (NLB), or even IIS' Application Request Routing (ARR). To ensure that each web server has the same content, you could replicate between them using DFS-R, or have them all draw their content from a common back-end shared folder (that would normally be a clustered file server, so that the shared folder wasn't a single point of failure).

Client Authentication to the Pull Server

Read back through the section on the Local Configuration Manager and you'll see that the LCM can be configured to have a credential. That credential can be used in one of two ways:

- For an SMB pull server, the LCM will present that credential to gain access to the SMB file share.

- For an HTTP(S) pull server, the LCM will present that credential to the web server *if prompted to do so.* Note that the credential is sent via the HTTP authentication protocol, which means you'll usually configure IIS to use Basic authentication, which means the password and username will be transmitted in the clear. You should only do that if you've configured the pull server and the LCM to use encrypted (HTTPS) connections, so that the password and username can't be intercepted.

Many environments probably won't use authentication. That is, they'll set up an HTTP(S) pull server and let IIS use its default Anonymous authentication. Doing so means *anyone* can potentially connect to the pull server and request configurations. On a purely intranet network, that might be acceptable – but keep in mind that if an intruder gains access to your LAN, they could query configuration information about your computers. That information would help them understand how your environment works, and might well give them clues for how to best attack your environment. So not using authentication isn't entirely without risks.

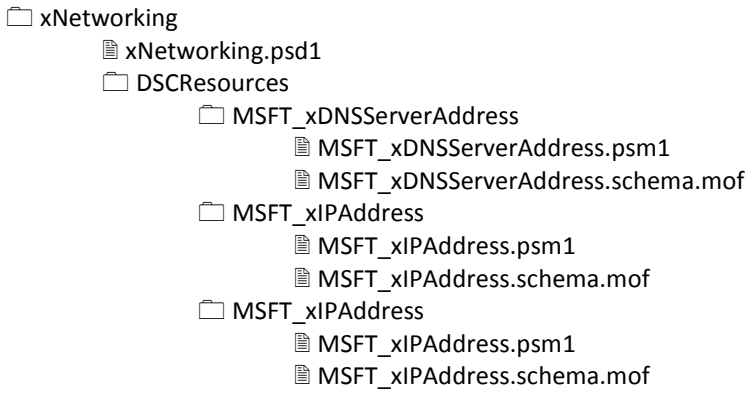
Writing a Custom DSC Resource

A custom DSC resource consists of at least two parts:

- A *root module*, which doesn't actually need to contain any code. This is just how PowerShell loads resources into memory. You can think of a root module as a "package" of DSC resources.

- One or more *resources*, which are kind of like sub-modules under the root module. These actually implement the code of the resource.

For example, the xNetworking module provided by Microsoft (in the DSC Resource Kit) looks a bit like this on disk:



So let's review what that all means:

- The *root module name* is xNetworking. It lives in a folder named xNetworking, and contains a module manifest named xNetworking.psd1. The manifest contains the following useful information (there's other textual information, but this is the important stuff):

```
@{
GUID = 'e6647cc3-ce9c-4c86-9eb8-2ee8919bf358'
PowerShellVersion = '4.0'
}
```

- A folder named DSCResources contains all of the resources.

- Each resource lives in its own folder, and then folder name is the resource's complete name (often longer than the resources' "friendly names").

- Within each resource folder is a script module (.psm1) and a schema (.schema.mof) file. Both files' filenames are the same as the resource's complete name.

The decision to bundle multiple resources into one root module, or to make each resource independent, is mostly a deployment decision. You have to deploy the entire root module, and the root module's version (from its manifest) defines the version number for each resource. So if you've got one guy maintaining 3 resources more or less in tandem, then maybe it makes sense to put them into a single root module.

Planning Your Resource

The first question you need to ask yourself is, "what information will my resource need to configure itself?" For example, if you were building a resource that could add or remove users from a line-of-business application, then the resource might need to know the user's name, perhaps their role or job title, and other pieces of information. Each piece of information becomes a *property,* or setting, of the resource.

Take the built-in **File** resource as another example. Its properties include the source path and destination path for a file. At a minimum, it needs those two pieces of information to see if a file exists on the target node, and to go get the file if necessary.

As a sort of running example, we're going to create a sample resource for a fictional line-of-business application. Our resource will be named **adatumOrdersApp**. Naming is important: we're pretending that we work for Adatum, one of Microsoft's fictional company names. Prefixing our resource name with the company name helps ensure that our resource doesn't conflict with any other resources that relate to order applications.

We've decided that our resource will be responsible for maintaining the user list of the application. Users consist of a login name, a full name, and a role. The role must be "Agent," "Manager" or "Auditor." So our resource will need to expose at least three properties. In addition, it will expose an "Ensure" property that can be "Present" or "Absent" (e.g., ensuring a user is, or isn't, in the system).

Starting Your Resource

We recommend starting with the DSC Resource Designer, which you can get from http://gallery.technet.microsoft.com/scriptcenter/xDscResourceDesigne-Module-22eddb29 (be sure to search for newer versions; as of this writing, it's v1.0). This gets unzipped until your Modules folder (instructions are on the download page), and adds PowerShell commands that help create a skeleton for your resource.

We start by defining our properties:

```
$logon    = New-DscResourceProperty –Name LogonName –Type String –Attribute Key
$ensure   = New-DscResourceProperty –Name Ensure –Type String –Attribute Write
                                    –ValidateSet "Present","Absent"
$fullname = New-DscResourceProperty –Name FullName –Type String –Attribute Write
$role     = New-DscResourceProperty –Name Role –Type String –Attribute Write
                                   –ValidateSet "Manager","Agent","Auditor"
```

> Note that we're formatting the code samples here for easier reading in the book. These won't run if you type them exactly as shown, but they're not meant to run, because they're controlling a fictional application.

That gives us four variables that each define one of our resource's properties, or settings. Now we can create the skeleton for the resource:

```
New-DscResource –Name adatumOrdersApp
                -Properties $logon,$ensure,$fullname,$role
                -Path 'C:\program files\windowspowershell\modules\adatumOrderApp'
                -FriendlyName adatumOrderApp
```

That command should create a folder for our root module, which will be named adatumOrderApp. It will also create the subfolder for the actual resource, which is also named adatumOrderApp. It'll also

create an empty adatumOrderApp.psm1 script module, and the adatumOrderApp.schema.mof file that defines the resource's structure.

Programming the Resource

The rest of our work will happen in adatumOrderApp.psm1. The file will contain the three functions that DSC requires – we just need to fill them in:

- **Get-TargetResource**. This function will be set up to accept, as parameters, only the property we defined as the key. If you look back, we did that to the LogonName property, meaning that property uniquely represents a user in our system. Get-TargetResource needs to check and see if the specified LogonName exists. The function is expected to return a hashtable that includes keys named LogonName, Ensure, Role, and FullName. If the specified user exists, then obviously those are all populated with the correct information and Ensure is set to "Present." If the user doesn't exist, then Ensure will be "Absent," LogonName will be whatever user we were checking for, and FullName and Role will be empty.

- **Set-TargetResource**. This function accepts parameters for LogonName, Ensure, Role, and FullName. It's expected to *check and see if the specified user exists.* It must take one of the following actions:

    - If Ensure is set to "Absent" and the user exists, this function should remove the user.
    - If Ensure is set to "Present" and the user exists, this function does nothing.
    - If Ensure is set to "Absent" and the user does not exist, this function does nothing.
    - If Ensure is set to "Present" and the user does not exist, this function must create the user.

  This function can set $global:DSCMachineStatus=1 if whatever this function did will require a system restart of the target node. Write-Verbose and Write-Debug should be used to return status information or debug information.

- **Test-TargetResource**. This function will accept parameters for LogonName, Ensure, Role, and FullName. It's expected to check and see if the specified information exists, and return either $True or $False. For example, if it's given a user name and Ensure is set to "Absent," it should return $False if the user does exist, and $True if the user doesn't. In other words, this function is begin given a desired configuration, and then indicating if the current system state meets that desire or not. This function doesn't *do* anything, it's just looking.

Within the function, you code whatever is necessary. Obviously, the code is going to differ massively based on whatever it is you're doing. But the general gist is:

- Your **Get** function returns the current state.
- Your **Set** function sets the system state.
- Your **Test** function compares the current state to the desired state.

We'll include a more detailed walkthrough of a more concrete example in a future revision of this guide. In the meantime, you'll find numerous production examples at http://github.com/powershellorg/DSC.

Think Modularly

As you plan a DSC resource, try to think about modular programming practices. For example, you will probably find a lot of duplicated code in your Test-TargetResource and Set-TargetResource functions, because they both have to check and see if things are configured as desired. Rather than actually

duplicating code, it might make sense to move the "check it" code into separate "utility" functions within the script module. You would then only export the three –TargetResource functions, keeping your "utility" functions private.

Here's a semi-pseudo-code mockup to help illustrate the idea:

```
Function Get-TargetResource {
  # ...
}

Function Test-TargetResource {
  (_CheckConfigA –and _CheckConfigB)
}

Function Set-TargetResource {
  If (-not _CheckConfigA) {
    # set item A
  }
  If (-not _CheckConfigB) {
    # set item B
}

Function _CheckConfigA {
  # ...
}

Function _CheckConfigB {
  # ...
}

Export-ModuleMember –Function *-TargetResource
```

In this mockup, the "utility" functions _CheckConfigA and _CheckConfigB would return True or False. Each one presumably checks some configuration thing on the computer. Test-TargetResource calls both of them, using a logical –and so that Test-TargetResource will only output True if both _CheckConfigA and _CheckConfigB output True. The Set-TargetResource function uses both "utility" functions, implementing some configuration change is either setting isn't correct. There's no technical requirement that your "utility" function names begin with an underscore; that's just a convention some programmers use to indicate an element that's meant "for internal use only."

Writing Composite DSC Resources

One limitation in DSC is that a given node can only be targeted with a single configuration MOF. Whether you're pushing MOFs, or have configured a node to pull a MOF from a pull server, you get one MOF per computer. That means your configuration scripts can start to get out of hand pretty quickly. For example, suppose you have some common configuration settings that are shared by your domain controllers, your web servers, and your SQL Server computers. You'd end up copying-and-pasting that common configuration stuff a lot, right? Problem being, if you ever need to change that common configuration, you now have to do it in multiple places. No fun at all.

Composite resources are designed to solve that need. Basically, you create a configuration script for your common configuration items. Those then get saved in a way that makes them *look like* a custom DSC resource. You can then consume that resource in other configurations. So you might end up with a main configuration file for a domain controller, one for a SQL Server, and one for a web server. Each will obviously contain some unique things, but they might all reference your "common configuration" for the things it contains. Understand that the "common configuration" isn't truly a DSC resource, but it's made to *look* like one. It's just a ton easier to make!

For example, suppose you created this configuration:

```
Configuration CommonConfig {
    Param([string]$IPAddress)

    Import-DscResource –ModuleName xNetworking

    WindowsFeature Backup {
           Ensure = 'Present'
           Name   = 'Windows-Server-Backup'
    }

    xIPAddress IPAddress {
           IPAddress = $IPAddress
           InterfaceAlias = "Ethernet"
           DefaultGateway = "192.168.10.2"
           SubnetMask = 24
           AddressFamily = "IPv4"
    }

    xDNSServerAddress DNSServer {
           Address = "192.168.12.8"
           InterfaceAlias = "Ethernet"
           AddressFamily = "IPv4"
    }

}
```

Notice that there's no **Node** section in this configuration! That's because this isn't going to target a specific node. We did use the xNetworking module (from http://gallery.technet.microsoft.com/scriptcenter/xNetworking-Module-818b3583), and we provided an input parameter so that the target node's desired IP address can be provided. We've hardcoded the remaining IP address and DNS server settings, presumably based on our knowledge of how our environment is set up. You could obviously parameterize any of those things – we just wanted to illustrate how you can mix-and-match parameterized information with hardcoded values.

The trick is in how you save this file. We're going to name it **CommonConfig.schema.psm1** – notice the "schema.psm1" filename extension, because that's what makes this magical. You then need to save it the same way you'd save a resource. For example, we'll save it in \Program Files\WindowsPowerShell\Modules\CommonConfigModule\DSCResources\CommonConfig\CommonConfig.schema.psm1.

Now you need to create a module manifest for it, in the same folder. The manifest filename must be CommonConfig.psd1:

```
New-ModuleManifest –Path "\Program
Files\WindowsPowerShell\Modules\CommonConfigModule\DSCResources\CommonConfig\CommonConfig.psd1" –
RootModule "CommonConfig.schema.psm1"
```

If you don't already have one, you'll also need a manifest at the root of the module folder:

```
New-ModuleManifest –Path "\Program
Files\WindowsPowerShell\Modules\CommonConfigModule\CommonConfigModule.psd1"
```

You're done. Now you can use that in a configuration:

```
Configuration DomainController {

    Import-DscResource –ModuleName CommonConfigModule

    Node DC1 {
           CommonConfig Common {
                  IPAddress = "192.168.10.202"
           }
    }
}
```

See how we took the $IPAddress parameter from the common configuration, and used that as a setting in the configuration item? Also notice that we imported the DSC resource at the top of the configuration, and then referred to the configuration name, **CommonConfig**, as a configuration setting.

We didn't really have to do any actual coding, here: the underlying **WindowsFeature**, **xIPAddress**, and **xDNSServerAddress** resources are doing all the work. We just bundled up those settings into a reusable configuration that looks like a resource.

You'll find a longer and more detailed discussion on composite resources at http://blogs.msdn.com/b/powershell/archive/2014/02/25/reusing-existing-configuration-scripts-in-powershell-desired-state-configuration.aspx.

Deploying Resources via Pull Servers

There are two ways to deploy new resources to your nodes: manually, or via a pull server. Manually is obviously time-consuming across a large number of nodes, although you could potentially use software like System Center Configuration Manager to automate that process. You can't really use DSC itself to deploy resources. While the **File** resource could handle it, you couldn't actually *use* the new resources in a configuration until you were sure they existed on the target nodes. So you couldn't write a configuration that copied a resource *and* tried to use it – even if you set dependencies.

Microsoft basically expects you to use pull mode, since nodes can figure out what resources they're missing and copy them from the pull server.

When you create a pull server, you specify the path where resources (modules) will be located. In our example of creating an HTTP(S) pull server, for example, we used this path:

```
            ModulePath              = "$env:PROGRAMFILES\WindowsPowerShell\DscService\Modules"
```

There's a specific naming convention required for modules deployed to a pull server. First, the entire module must be zipped. The filename must be in the form *ModuleName_version*.zip. You must use **New-DscChecksum** to create a corresponding checksum file named *ModuleName_version*.zip.checksum. Both files must be located in the pull server's **ModulePath**.

When you create a configuration that uses a module from the pull server, you have to ensure that the MOF contains the module's name and version number. For example, this MOF excerpt clearly indicates that the **WindowsFeature** resource lives in version 1.0 of the PSDesiredStateConfiguration module:

```
Instance of MSFT_RoleResource as $MSFT_RoleResource1ref
{
  ResourceID = "[WindowsFeature]Backup";
  ModuleName = "PSDesiredStateConfiguration";
  ModuleVersion = "1.0";
};
```

> Note that there are reports (see http://connect.microsoft.com/PowerShell/feedback/details/804230/lcm-fails-to-extract-module-zipped-with-system-io-compression-zipfile) of ZIP files not working with DSC if they were created by using the .NET Framework's System.IO.Compression.ZipFile.CreateFromDirectory() method. Be aware.

Note that DSC will always try to use the newest version of a resource (technically, only the module in which a resource lives has a version; every resource in that module is considered to be the same version number as the module itself). So, when DSC pulls a configuration, it'll check the pull server for new versions of any resources used by that configuration. You can configure the LCM to *not* download newer modules, if desired. In a configuration, you can also specify a version number when using **Import-DscResource**; if you do so, then DSC will look for, and only use, the specified version.

Using the Log Resource

Using the **Log** resource in a configuration script doesn't actually reconfigure the target node. Instead, it gives you a way to write to the target node's event log. Specifically, the **Applications and Services Logs / Microsoft / Windows / Desired State Configuration** event log. Here's an example, which would appear within the Node block of a configuration script:

```
Log InterestingInfo
{
  Message = "Whatever you'd like to say"
  DependsOn = "[File]MassiveFileCopy"
}
```

This configuration item is named InterestingInfo, and it will only run if the **File** setting named MassiveFileCopy has already completed.

Configuring the Local Configuration Manager

In our walkthrough on building a pull server, we included a short example of modifying a target node's Local Configuration Manager (LCM). On any machine, you can run **Get-DscLocalConfigurationManager** to check that node's LCM settings. These include:

- **AllowModuleOverwrite**. When set to $True, this allows new modules (downloaded from a pull server) to overwrite older ones.

- **CertificateID**. The thumbprint of the node's certificate. This certificate has to be stored locally on the node, and it will be used to decrypt any encrypted data that's included in configuration files. For example, DSC lets you include PSCredential objects in a MOF file, but requires that they be encrypted. We'll cover this in the next section of the guide.

- **ConfigurationID**. The GUID that the node will look for on the pull server when it pulls its configuration.

- **ConfigurationMode**. How the LCM works. **ApplyOnce** only applies a configuration one time. **ApplyAndMonitor** applies the configuration, and then re-examines it every so often. Changes are reported in logs. **ApplyAndAutoCorrect** reapplies the configuration automatically.

- **ConfigurationModeFrequencyMins**. The number of minutes the LCM re-examines the configuration. This must be a multiple of **RefreshFrequencyMinutes**. If you specify a non-multiple, your value will be rounded up to the nearest multiple. The minimum value is 30 minutes.

- **Credential**. The credential used to access remote resources. Normally, the LCM runs as SYSTEM, and has very limited ability to access non-local resources. If you're going to have the LCM pull from an SMB pull server, for example, then you'll almost definitely want to provide a valid credential so that it can do so.

- **DownloadManagerCustomData**. Settings that are passed to the selected download manager. For example, the WebDownloadManager requires a URI, and can be configured to allow HTTP.

- **DownloadManagerName**. Either **WebDownloadManager** or **DscFileDownloadManager.**

- **RebootNodeIfNeeded.** If set to $True, the target node is automatically restarted when a configuration requires it. This is $False by default, which means configuration operations that require a reboot won't complete until someone manually reboots the node. Microsoft chose $False because it's safer – when set to $True, the reboot happens *right away* once DSC decides it's necessary. On a server in a production environment, you can see where that'd be problematic.

- **RefreshFrequencyMins.** In Pull mode, specifies how often the node checks for a new configuration on the pull server. The minimum value is 15 minutes.

- **RefreshMode.** Either **Push** or **Pull**. You must specify a **DownloadManagerName** setting.

Note that the LCM will only deal with a single configuration. If you push a new one, the LCM will start using that one instead of any previous one. In pull mode, it will pull only one configuration. You can use composite configurations (we described them earlier) to combine multiple configurations into one.

When you create a meta configuration MOF, you apply it by running **Set-DscLocalConfigurationManager**, not **Start-DscConfiguration**. Configurations containing a **LocalConfigurationManager** setting should not contain any other configuration items.

For more information on LCM configuration, read http://blogs.msdn.com/b/powershell/archive/2013/12/09/understanding-meta-configuration-in-windows-powershell-desired-state-configuration.aspx. As of this writing, the information at http://technet.microsoft.com/en-us/library/dn249922.aspx is inaccurate.

Including Credentials in a Configuration

There may come times when you need a configuration to include a credential. In the configuration script, you can simply pass in a PSCredential object as a parameter, and use Get-Credential to create the PSCredential object. When the configuration script is run, the PSCredential object is serialized into the MOF, which is then transferred (via push or pull) to target nodes. Of course, you don't want credentials passed in clear text, right?

The Right Way

In order for credential-passing to work, a certificate must be present on all target nodes (specifically, in the Local Machine store). The thumbprint of that certificate must be specified in the nodes' LCM configuration (see the previous section of this guide). The same certificate must be installed on the computer where you're running the configuration to create MOF files. When you create the configuration, you pass in the certificate's thumbprint as part of the configuration data. We'll walk through how to do that.

Note that this walkthrough is adapted from the PowerShell team blog entry at http://blogs.msdn.com/b/powershell/archive/2014/01/31/want-to-secure-credentials-in-windows-powershell-desired-state-configuration.aspx, which you may want to review for further details.

Here's a quick example of a configuration that includes a PSCredential:

```
configuration CredentialEncryptionExample
   {
       param(
           [Parameter(Mandatory=$true)]
           [ValidateNotNullorEmpty()]
           [PsCredential] $credential
           )

        Node $AllNodes.NodeName
       {
           File exampleFile
           {
               SourcePath = "\\Server\share\path\file.ext"
               DestinationPath = "C:\destinationPath"
               Credential = $credential
           }
       }
   }
```

This configuration, when run, will prompt you for a PSCredential – a username and password. Notice that the **File** resource is being used to copy a file from a UNC path. Assuming that UNC path requires a non-anonymous connection, we'll need to provide a credential – and we pass the PSCredential that you were prompted for.

The trick in this configuration comes from the $AllNodes variable. Here's how we set that up:

```
$ConfigData = @{
  AllNodes = @(
    @{
       # The name of the node we are describing
```

```
        NodeName = "SERVER2"
        # The path to the .cer file containing the
         # public key of the Encryption Certificate
        # used to encrypt credentials for this node
        CertificateFile = "C:\publicKeys\targetNode.cer"

        # The thumbprint of the Encryption Certificate
        # used to decrypt the credentials on target node
        Thumbprint = "AC23EA3A9E291A75757A556D0B71CBBF8C4F6FD8"
      };
    );
}
```

So we've created a hashtable in $ConfigData. The hashtable has a single key, named AllNodes, and the value of that key is an array of one item. That one item is itself a hashtable with three keys: NodeName, CertificateFile, and Thumbprint. The certificate isn't installed on this computer, but has instead been exported to a .CER file. However, the certificate must be *installed* on any nodes that we'll target with this configuration. We've also configured the LCM on the target node to have that thumbprint as its CertificateId setting.

So, we've created the configuration. We've created a block of *configuration data* to pass to it, and that block includes the certificate details. Now we need to run the configuration, passing in that configuration data:

```
CredentialEncryptionExample –ConfigurationData $ConfigData
```

Notice that we never defined –ConfigurationData as a parameter of the configuration; it's a built-in parameter supported by all configurations, automatically. PowerShell will automatically encrypt the credential in the MOF file, because PowerShell was designed to recognize PSCredential objects and encrypt them. The target node will recognize the encrypted data, and use its copy of the certificate to decrypt it.

As a note, the node where we created the MOF file uses the public key from the certficiate; the target node uses the private key for decryption.

The Easier, Less-Right Way

Of course, certificates can sometimes be less than convenient, and sometimes you don't need the security. For example, suppose you're setting up a lab environment, and all of your passwords are going to be Pa$$w0rd anyway. Who cares about encryption?

In that case, you can force PowerShell to include plain-text, unencrypted passwords in the MOFs. Here's an example that we used to set up a 3-machine lab environment:

```
Configuration LabSetup {
    Param(
        [Parameter(Mandatory=$True)]
        [PSCredential]$DomainCredential,

        [Parameter(Mandatory=$True)]
        [PSCredential]$ SafeModeCredential,

        [Parameter(Mandatory=$True)]
        [string]$EthernetInterfaceAlias
    )

    # Ensure KB2883200 is installed on all computers
    # Expecting Windows 8.1 and Windows Server 2012 R2
```

```
    # Run this script on each computer to produce
    # MOF files and start configuration

    # You should do DC1 first, then the other two

    # You will be prompted for two credentials
    # For the first, provide DOMAIN\Administrator and Pa$$w0rd
    # For the second, provide Administrator and Pa$$w0rd

    # This assumes that the Student Materials files are at
    # E:\AllFiles

    # Computers must have the following downloaded and installed
    # into C:\Program Files\WindowsPowerShell\Modules:
    # - http://gallery.technet.microsoft.com/xActiveDirectory-f2d573f3
    # - http://gallery.technet.microsoft.com/scriptcenter/xComputerManagement-Module-
3ad911cc
    # - http://gallery.technet.microsoft.com/scriptcenter/xNetworking-Module-818b3583

    Import-DscResource -ModuleName  ActiveDirectory,xNetworking,xComputerManagement

    Node 'DC1' {
        WindowsFeature ADDSInstall {
            Ensure = 'Present'
            Name = 'AD-Domain-Services'
        }

        xIPAddress IP {
            IPAddress = '10.0.0.10'
            InterfaceAlias = $EthernetInterfaceAlias
            DefaultGateway = '10.0.0.1'
            SubnetMask = 24
            AddressFamily = 'IPv4'
        }

        xDNSServerAddress DNS {
            Address = '127.0.0.1'
            InterfaceAlias = $EthernetInterfaceAlias
            AddressFamily = 'IPv4'
        }

        xComputer Computer {
            Name = 'DC1'
        }

        xADDomain Adatum {
            DomainName = domain.pri'
            DomainAdministratorCredential = $domaincredential
            SafemodeAdministratorPassword =  $SafeModeCredential
            DependsOn =  '[WindowsFeature]ADDSInstall',
                         '[xIPAddress]IP',
                         '[xDNSServerAddress]DNS',
                         '[xComputer]Computer'
        }

    }

    Node 'CL1' {
        xIPAddress IP {
            IPAddress = '10.0.0.30'
            InterfaceAlias = $EthernetInterfaceAlias
            DefaultGateway = '10.0.0.1'
            SubnetMask = 24
            AddressFamily = 'IPv4'
        }

        xDNSServerAddress DNS {
            Address = '10.0.0.10'
```

```
            InterfaceAlias = $EthernetInterfaceAlias
            AddressFamily = 'IPv4'
        }

        xComputer Computer {
            Name = 'CL1'
            DomainName = 'DOMAIN'
            Credential = $DomainCredential
            DependsOn = '[xIPAddress]IP','[xDNSServerAddress]DNS'
        }

        Environment Env {
            Name = 'PSModulePath'
            Ensure = 'Present'
            Path = $true
            Value = 'E:\AllFiles\Modules'
        }
    }

    Node 'SRV1' {
        xIPAddress IP {
            IPAddress = '10.0.0.20'
            InterfaceAlias = $EthernetInterfaceAlias
            DefaultGateway = '10.0.0.1'
            SubnetMask = 24
            AddressFamily = 'IPv4'
        }

       xDNSServerAddress DNS {
            Address = '10.0.0.10'
            InterfaceAlias = $EthernetInterfaceAlias
            AddressFamily = 'IPv4'
        }

        xComputer Computer {
            Name = 'SRV1'
            DomainName = 'DOMAIN'
            Credential = $DomainCredential
            DependsOn = '[xIPAddress]IP','[xDNSServerAddress]DNS'
        }
    }

}

$ConfigurationData = @{
    AllNodes = @(
        @{
            NodeName='LON-DC1'
            PSDscAllowPlainTextPassword=$true
        }
        @{
            NodeName='LON-SRV1'
            PSDscAllowPlainTextPassword=$true
        }
        @{
            NodeName='LON-CL1'
            PSDscAllowPlainTextPassword=$true
        }
    )
}



LabSetup -OutputPath C:\LabSetup -ConfigurationData $ConfigurationData `
         -EthernetInterfaceAlias 'Ethernet0'

Start-DscConfiguration -Path C:\LabSetup -ComputerName $env:COMPUTERNAME
```

This is actually a pretty cool example – there are several things to notice:

- We've included more than one NODE section. So, we're using this single configuration to set up an entire *environment.* When we run this, we'll get three MOF files.

- Notice the PSDscAllowPlainTextPassword in the configuration data? That's what enables the MOFs to legally contain unencrypted passwords. Notice how we prompt for two credentials in the Param() block, and then use those credentials throughout the configuration.

- We parameterized the Ethernet interface alias, because sometimes it isn't "Ethernet0." Parameterizing it makes it easier to change that for future setups.

- Running this script on a lab computer produces all three MOFs, but then only the local computer's MOF is actually run. We presume that the "bare" computers will have the correct computer names already, something that's taken care of by our base images. Obviously, not every lab setup will be that way, but ours is.

Troubleshooting DSC and Configurations

Configurations can be difficult to troubleshoot and debug, because they're always running "behind the scenes" on a remote computer. Even worse, the remote computer isn't running your configuration script per se – it's going through a MOF file, and then running DSC resources to implement the instructions in the MOF. It's a lot of moving parts.

Fortunately, the DSC engine writes a lot of info into the Windows event logs. Specifically, you'll find the logs under **Applications and Services Logs / Microsoft / Windows / Desired State Configuration**. You'll need to right-click a folder in the event viewer to enable the Analytic and Debug logs; otherwise, you'll just see the Operational log. From PowerShell, you can run:

```
Get-WinEvent –LogName "Microsoft-Windows-Dsc/Operational"
```

To view the contents of the Operational log on the local computer. To enable the Analytic and Debug logs from the command-line:

```
Wevtutil.exe set-log "Microsoft-Windows-Dsc/Analytic" /q:true /e:true
Wevtutil.exe set-log "Microsoft-Windows-Dsc/Debug" /q:true /e:true
```

The Operational log contains error messages, and is a good place to start troubleshooting. The Analytic log has more detailed messages, and any verbose messages produced by the DSC engine. The Debug log contains developer-level messages that might not be useful unless you're working with Microsoft Product Support to troubleshoot a problem. For more information on using these logs, visit http://blogs.msdn.com/b/powershell/archive/2014/01/03/using-event-logs-to-diagnose-errors-in-desired-state-configuration.aspx.

Wave 2 of the DSC Resource Kit includes an xDscDiagnostics module, which you can use to help analyze DSC failures. There are two commands in the module: **Get-xDscOperation** and **Trace-xDscOperation**. For a quick walkthrough of using these commands, visit http://blogs.msdn.com/b/powershell/archive/2014/02/11/dsc-diagnostics-module-analyze-dsc-logs-instantly-now.aspx.

Compliance Servers

A DSC pull server can actually contain two components: the pull server bit, plus what Microsoft is presently calling a "compliance server." That name will likely change, but it's a good placeholder for now.

In short, the compliance server maintains a database (which it shares with the pull server piece) that lists all of the computers that have checked in with the pull server, and shows their current state of configuration compliance – that is, whether or not their actual configuration matches their desired configuration. So, part of the LCM's job is to not only grab configuration MOFs from the pull server, but also to report back on how things are looking.

You can then generate reports from that database, showing you what nodes are compliant and which ones aren't. Remember that the LCM can be configured to *not continually apply configurations* – we covered that earlier with its **ConfigurationMode** setting, which can be set to **ApplyAndMonitor.** In that mode, the LCM can let you know which machines are out of compliance, but not actually do anything about it.

Note that the LCM will only report back if you're using an HTTP(S) pull server; this trick won't work with an SMP pull server.

It's still early days for this particular aspect of DSC, so we'll expand this section of the guide once there's more information from Microsoft on this feature.