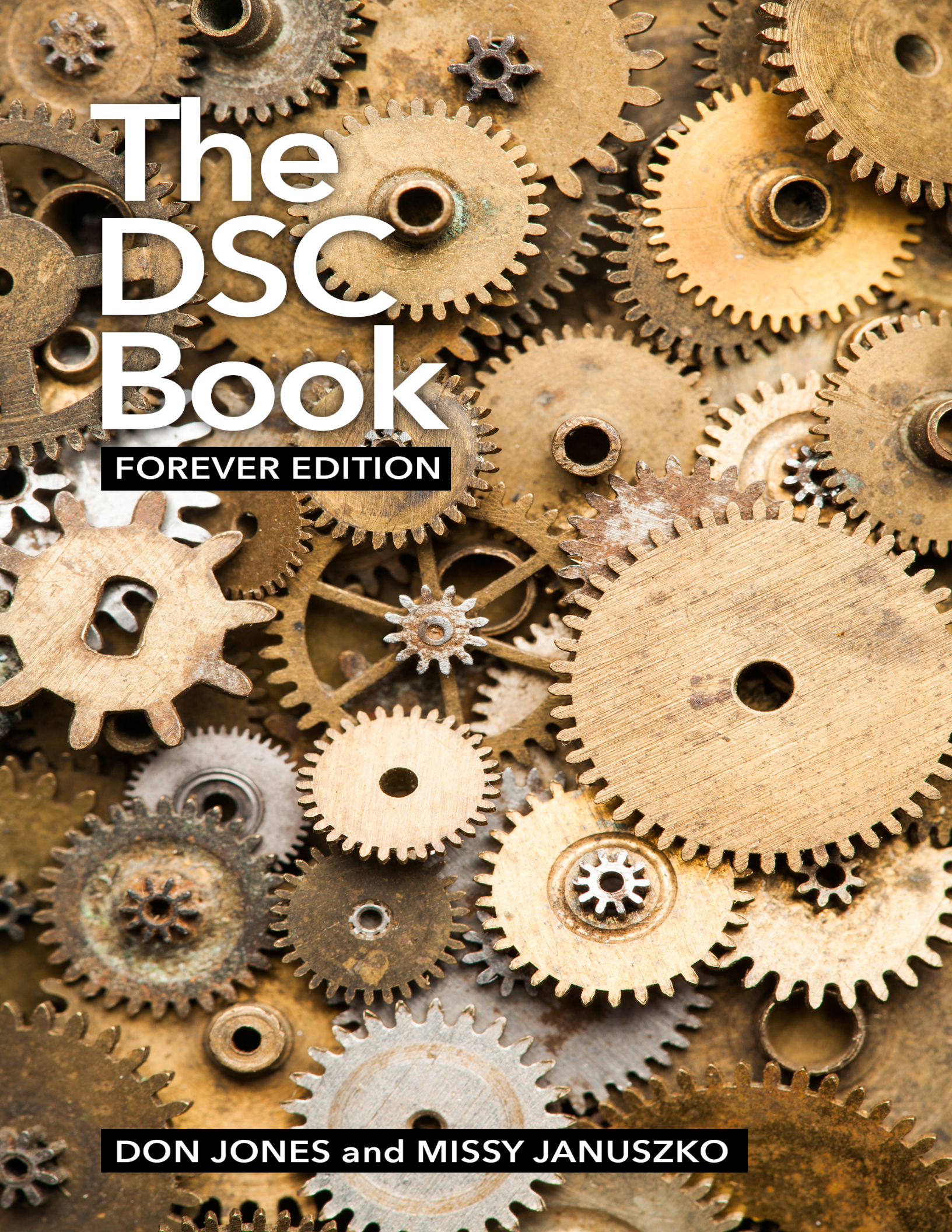
The DSC Book
FOREVER EDITION
DON JONES and MISSY JANUSZKO

The DSC Book

“Forever” Edition

Don Jones and Melissa Januszko

This book is for sale at http://leanpub.com/the-dsc-book

This version was published on 2018-12-18

Tweet This Book!

Please help Don Jones and Melissa Januszko by spreading the word about this book on Twitter!

The suggested tweet for this book is:

Just got the last #PowerShell #DSC book I'll need to buy http://leanpub.com/the-dsc-book

The suggested hashtag for this book is #dscbook.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#dscbook

Also By Don Jones

The PowerShell Scripting and Toolmaking Book

Become Hardcore Extreme Black Belt PowerShell Ninja Rockstar

Be the Master

Don Jones' PowerShell 4N00bs

Don Jones' The Cloud 4N00bs

Instructional Design for Mortals

How to Find a Wolf in Siberia

Tales of the Icelandic Troll

PowerShell by Mistake

The Culture of Learning

For Chris... for everything.

Contents

About This Book

The 'Forever Edition' of this book is published on LeanPub[1], an "Agile" online publishing platform. That means the book is published as we write it, and *that* means although you may not have a complete book right now, you eventually will - and we'll also be able to revise it as needed. We also appreciate your patience with any typographical errors, and we appreciate you pointing them out to us - in order to keep the book as "agile" as possible, we're forgoing a traditional copyedit. Our hope is that you'll appreciate getting the technical content quickly, and won't mind helping us catch any errors we may have made. You paid a bit more for the book than a traditional one, but that up-front price means you can come back whenever you like and download the latest version. We plan to expand and improve the book pretty much forever, so it's hopefully the last one you'll need to buy on this topic!

You'll also find this book offered on traditional booksellers like Amazon. In those cases, the book is sold as a specific edition, such as "Second Edition." These represent a point-in-time snapshot of the book, and are offered at a lower price than the Agile-published version. These traditional editions do not include future updates.

If you purchased this book, thank you. Know that writing a book like this takes hundreds of hours of effort, during which we're not making any other income. Your purchase price is important to keeping a roof over our families' heads and food on our tables. Please treat your copy of the book as your own personal copy - it isn't to be uploaded anywhere, and you aren't meant to give copies to other people. We've made sure to provide a DRM-free file (excepting any DRM added by a bookseller other than LeanPub) so that you can use your copy any way that's convenient for you. We appreciate your respecting our rights and not making unauthorized copies of this work.

If you got this book for free from someplace, know that you are making it difficult for us to write books. When we can't make even a small amount of money from our books, we're encouraged to stop writing them. If you find this book useful, we would greatly appreciate you purchasing a copy from LeanPub.com or another bookseller. When you do, you'll be letting us know that books like this are useful to you, and that you want people like us to continue creating them.

---

[1] http://leanpub.com

About the Authors

Don Jones has been a Microsoft MVP Award recipient since 2003 for his work with Windows PowerShell and administrative automation. He has written dozens of books on information technology, and today helps design the IT Ops curriculum for Pluralsight.com. Don is also President, CEO, and co-founder of The DevOps Collective (devopscollective.org), which offers IT education programs and scholarships, and which runs PowerShell.org and PowerShell + DevOps Global Summit (powershellsummit.org).

Don's other recent works include: * Learn Windows PowerShell in a Month of Lunches (Manning.com) * The DSC Book (LeanPub.com) * The Pester Book (LeanPub.com) * The PowerShell Scripting & Toolmaking Book (LeanPub.com) * Learn PowerShell Toolmaking in a Month of Lunches (Manning.com) * Learn SQL Server Administration in a Month of Lunches (Manning.com)

Follow Don on Twitter @concentratedDon, on Facebook at facebook.com/concentrateddon, or on LinkedIn at LinkedIn.com/in/concentrateddon. He blogs at DonJones.com.

Missy Januszko is an independent IT consultant, with more than 20 years of experience as an enterprise hosting architect, large-scale infrastructure designer, and hosted application designer. She specializes in DevOps, automation and configuration management, PowerShell, and Active Directory, and has broad experience across the entire line of Microsoft business technologies. Missy is a co-author of "The DSC Book" with Microsoft MVP Don Jones, and she is also a conference speaker on DSC-related topics.

Feedback

We'd love your feedback. Found a typo? Discovered a code bug? Have a content suggestion? Wish we'd answered a particular question? Let us know.

First, please have a chapter name, heading reference, and a brief snippet of text for us to reference. We can't easily use page numbers, because our source documents don't have any.

Second, understand that due to time constraints like having full-time jobs, we can't personally answer technical questions and so forth. If you have a question, please hop on the forums at PowerShell.org[2], where we and a big community of enthusiasts will do our best to help.

Then, head to the LeanPub website and use their email link[3] to email us. We can't always reply personally to every email, but know that we're doing our best to incorporate feedback into the book.

Finally, accept our thanks!

[2] http://powershell.org
[3] https://leanpub.com/the-dsc-book/email_author/new

A Note on Code Listings

The code formatting in this book only allows for about 60-odd characters per line. We've tried our best to keep our code within that limit, although sometimes you may see some awkward formatting as a result.

For example:

```
        Invoke-CimMethod -ComputerName $computer `
                         -MethodName Change `
                         -Query "SELECT * FROM Win32_Service WHERE Name = '$ServiceNa\
me'" `
```

Here, you can see the default action for a too-long line - it gets word-wrapped, and a backslash inserted at the wrap point to let you know. We try to avoid those situations, but they may sometimes be unavoidable. When we *do* avoid them, it may be with awkward formatting, such as in the above where we used backticks (‘) or:

```
        Invoke-CimMethod -ComputerName $computer `
                         -MethodName Change `
          -Query "SELECT * FROM Win32_Service WHERE Name = '$ServiceName'" `
```

Here, we've given up on neatly aligning everything to prevent a wrap situation. Ugly, but oh well.

You may also see this crop up in `inline code` snippets, especially the backslash.

If you are reading this book on a Kindle, tablet or other e-reader, then we hope you'll understand that all code formatting bets are off the table. There's no telling what the formatting will look like due to how each reader might format the page. We trust you know enough about PowerShell not to get distracted by odd line breaks or whatever.

When *you* write PowerShell code, you should not be limited by these constraints. There is no reason for you to have to use a backtick to "break" a command. Simply type out your command. If you want to break a long line to make it easier to read without a lot of horizontal scrolling, you can hit `Enter` after any of these characters:

- Open parenthesis (
- Open curly brace {
- Pipe |

- Comma ,
- Semicolon ;
- Equal sign =

This is probably not a complete list, but breaking after any of these characters makes the most sense.

Anyway, we apologize for these artifacts.

Code Samples

Longer code listings are available at http://github.com/concentrateddon/TheDSCBookCode. Shorter listings - because they're often not useful as-is - are not necessarily replicated there. We really mean for most of the code in this book to be *illustrations,* not copy-and-paste usable.

Introduction

Desired State Configuration, or DSC, was introduced as part of Windows Management Framework (WMF) 4.0 - the same package that included Windows PowerShell 4.0. DSC is essentially the final step originally envisioned for Windows PowerShell in inventor Jeffrey Snover's *Monad Manifesto*[4]. DSC builds upon a decade-long investment in Windows PowerShell that included broad cmdlet coverage, a robust (yet simple) scripting language, Remoting, and more.

What is DSC?

Many organizations - maybe even the one you work for - have written configuration documents that describe how certain computers should be configured. "In our environment, a domain controller has *this* set of services installed, *these* pieces of software, and is configured *this* way." When someone sets up a new machine, they often consult these standards, which are designed to help configurations stay consistent, reliable, and secure.

At a high level, DSC simply asks you to write those configuration documents in a specific way that a computer, as well as a human, can understand. Rather than simply writing, "a domain controller should have Active Directory, DNS, and DHCP installed," you might write it like this:

```
WindowsFeature DnsServer {
        Name = "DNS"
        Ensure = "Present"
}
WindowsFeature DhcpServer {
        Name = "DHCP"
        Ensure = "Present"
}
WindowsFeature AD {
        Name = "ADDS"
        Ensure = "Present"
}
```

You would then deploy that configuration document directly to each domain controller, including any new ones you want to create. DSC "reads" that document, and does whatever is necessary to "make it happen." It checks to make sure the specific services are installed, and if they're not, it installs them. Every so often, it can re-check to make sure everything is still configured correctly, and it can even report back to you and let you know that everything is in order (or not).

[4] https://leanpub.com/themonadmanifestoannotated

The actual process is actually a little more complex (but only a little bit), and your configurations can obviously be a *lot* more detailed and comprehensive. But, at a high level, this is what DSC does. It takes a human-readable document that is also machine-readable, and it does the work of configuring systems to match your specification.

Not Just for Provisioning

It's very easy to fall into the trap of thinking of DSC as a way of setting up computers. "I'll push out this configuration to get the computer going, and then from there I'll manage it like I usually do." That thinking misses the entire point of DSC.

DSC is about putting a computer into a *desired state* of configuration, and then *keeping* the computer in that desired state. But what you "desire" of a computer can certainly change over time. The ultimate idea behind DSC is that you *never again configure a computer manually*. Instead, when you change your mind about what a computer should look like, you modify its DSC configuration. DSC reads that new configuration, realizes that the computer doesn't look like that, and "corrects" the configuration, bringing the computer in compliance with your *new* desired state.

How Does DSC Compare to Group Policy?

On the surface, DSC and Group Policy seem to serve the same high-level purpose. Both of them enable you to describe what you want a computer to look like, and both of them work to keep the computer looking like that. But once you dig a little deeper, the two technologies are grossly different.

Group Policy is part and parcel of Active Directory (AD), whereas DSC has no dependency on, or real connection to, AD.

Group Policy makes it easy to target a computer dynamically based on its domain, its location (organizational unit, or OU) within the domain, its physical site location, and more. Group Policy can further customize its application by using Windows Management Instrumentation (WMI) filters and other techniques. DSC, on the other hand, is very static. You decide ahead of time what you want a computer to look like, and there's very little application-time logic or decision-making.

Group Policy predominantly targets the Windows Registry, although Group Policy Preferences (GPP) enables additional configuration elements. Extending Group Policy to cover other things is fairly complex, requires native C++ programming, and involves significant deployment steps. DSC, on the other hand, can cover any configuration element that you can get to using .NET or Windows PowerShell. Extending DSC's capabilities is a simple matter of writing a PowerShell script, and deploying those extensions is taken care of by DSC's infrastructure.

For these reasons, DSC has primarily been focused on server workloads, since server configurations tend to be fairly unchanging. DSC is also suitable for statically configured client computers, such as computers in a call center, a student lab environment, a kiosk scenario, and so on. But because

DSC doesn't readily support a lot of application-time logic, it's less suitable for the dynamic, people-centric world of client computers.

It's entirely possible that, in the future, Microsoft could rip out the guts of Group Policy and replace it with DSC. Client-side logic could still pull DSC configurations dynamically from an AD domain controller, just as Group Policy does today. It'd be a way of massively extending what Group Policy can do, while keeping all the dynamic-targeting advantages of Group Policy. But today, you'll mostly see DSC discussed in terms of server workloads and more-or-less static configurations.

Cattle, Not Pets

DSC is one of those technologies that can be hard to grasp until you truly embrace a couple of its core design principles. It's not one of those things where you can kind of pick and choose the things you like about it; if you're going to be successful with DSC, you kind of have to "drink the Kool-Aid" and go all-in with its approach.

The first principle that you have to embrace is that *DSC is meant for scale*. If you're managing five servers, DSC is going to be irritating. DSC is useful when you need to manage a hundred servers… or a thousand… or tens of thousands. And one thing you have to accept about managing that many servers is that you have to avoid servers that are unique, special unicorns - or as Jeffrey Snover and I like to call them, *pets*.

A pet has a name. It has a unique personality. It's special. You cuddle it, and you have a lot of fun stories about it and its history. DSC isn't so hot at managing those kinds of machines. Oh, you can do it, for sure - but you're going to find yourself running up against seeming shortcomings in DSC. Thing is, they're not shortcomings - DSC just wasn't designed with special snowflakes in mind.

DSC was designed to manage *at scale*. You can't have a thousand pets in your home, but you can have a thousand cattle on a farm, right? DSC is designed for cattle management. Cattle don't have names (well, they might have names for themselves, but nobody cares). Cattle aren't unique. They're not individually special. If one does become special - perhaps by needing customized living arrangements and a 401(k) plan - you're likely just to shoot it, eat it, and go back to managing the herd.

Adopting this approach means you will probably have to make some significant changes to how you think about server management. For example, today, you probably have unique names for all of your servers, and you probably know those names. Those are pets. If you decide to use DSC to manage a server's complete configuration, including its unique, special name, then DSC isn't going to be as efficient as it could be. Instead, you could switch to a "cattle" approach. Sure, your servers will still have unique names - but Windows makes a name up for itself when you install it. Just because the server thinks it has a name doesn't mean you need to know it, or that you need to care about it. So DSC simply ignores the server's name. It's just a cow, right? Instead, DSC assigns the server to a load balancer, or to a CNAME record in DNS, or some other abstraction. So you'll be able to refer to the server - and you might simply refer to a *group* of them, as with a group of web

servers that are all serving the same function. You just won't use a special, unique name for that server.

This is a *huge* shift in thinking for most Windows administrators. Fortunately, you don't need to go all-in with that concept right off the bat. DSC will still let you set things up more or less the way you do today, although you may find some parts of it seem inefficient or cumbersome. As you start to shift your thinking, though, DSC will become an easier and more efficient way to manage your machines.

Technology, not Tool

The other thing to know about DSC is that it's a technology - not a tool. In the same way, Active Directory is a technology, and something like AD Users & Computers is a tool that you use to manage the technology. Today, Microsoft hasn't built a lot of tools to make managing DSC easier - it's a pretty manual, low-level process. That's changing a bit - I have a chapter coming up on DSC in Azure, for example, that shows some of what Microsoft has been building in the way of tools. Just understand that, right now, there are some rough edges around DSC because all we have is the technology, not any tooling to help us use the technology more effectively. But don't let that stop you.

A 2018 Update: What is DSC Good For?

This is a question we get asked a lot. As more and more workloads start running on things like containers, where DSC doesn't have a strong role, you might wonder if DSC has a future. We think so, but you first have to understand one thing: there is no one-size-fits-all tool. Meaning, DSC is good for particular scenarios; it's less good for others. So let's think about the kinds of workloads we have. In no particular order:

Ephemeral. These are workloads that exist for only a very short period of time, such as a container that's been spun-up to service a particular web request. These workloads require *declarative configuration*, meaning they spin up in their "desired state," perform their job, and then go away. DSC isn't a good fit for these.

Endpoint. Desktops, laptops, and other client computers. While DSC *can* be used for these, we don't feel it's *awesome* in all client configuration scenarios. DSC isn't a *configuration management* system; it presently lacks the tooling that could make it a robust client solution. GPO, on the other hand, has that tooling, as do platforms like System Center Configuration Manager. We see people trying to use DSC as a software deployment platform, and it just isn't great at it. Deploying software (let alone patches) to thousands of clients is *hard*, and it's why SCCM is so complex. DSC doesn't match up, because it wasn't designed to. This is triply true if you're deploying MSIs, because so many of them are badly built and DSC has no provisions for dealing with many of them (like ones that require a user profile to be active, which DSC doesn't do).

Static and long-lived. We don't think much about bare-metal servers these days, but by definition they're not something you just deploy in a moment and tear down a moment later. Bare-metal is time-consuming and can be delicate; DSC is a great way at keeping these workloads configured the same way, all the time. Proof? Azure Stack, a bare-metal solution that relies on thousands of lines of DSC code (and more than a few custom resources) to keep its delicate balancing act in order.

Legacy. Ephemeral workloads might be the Now Hotness, but we all have (and likely always will have) workloads that require a lot of delicate configuration, need to run in their own VM, and are special snowflakes. DSC can be great at helping to keep these properly configured. You might not ever use DSC to initially build one of these serves, but you might well use it to help control configuration drift on them.

State-Coupled. We all have plenty of workloads where the configuration of the workload itself–it's "state," if you will–is part of its install. That is, perhaps the data used by the workload is elsewhere, but the workload's identity and "bootstrap" code is part of the "system drive." Think Exchange Server, SharePoint Server, and SQL Server. These are "kinda special snowflakes" in that they're difficult to rebuild, and by their very nature you wouldn't just tear them down and rebuild them all the domain. Domain controllers fall into this category, because can you imagine the replication workload if you just tore down half your Dns and built them from scratch every night? DSC can also be good at maintaining configuration on these workloads.

We do *not* see a world where DSC is running on *every single computing workload you own*. The trick is in finding the right place and time, and using the tool how it was intended to be used. Nobody's taking a Camry to an F-1 race, but nobody's saying a Camry is a bad car. You just need the right tool for the workload at hand.

Part 1: Design Decisions

DSC isn't something you can just start installing. It's not a *solution* like Exchange Server or SQL Server; you don't "install it, and then figure it out." DSC is a set of technology components, and they're designed to fit together in a lot of different ways, to accommodate a lot of different business needs. It's *critical* that you understand these components *up front*, or you're likely to deploy something that's painful at best, and a security nightmare at worse.

Designing DSC

Deciding *how* to "do" DSC in your environment is a huge deal. Decide wrong, and you could end up not getting the flexibility you need, or you could end up making things way more complicated than you need.

There are two important things to remember when it comes to designing your DSC approach:

- There is no one approach that is right for everyone - you need to choose an approach that meets *your* needs.
- You need to understand the options thoroughly before deciding.

This chapter will help you understand what DSC is doing, how it does it, and what your options are for designing your DSC approach.

The Players

The main actor in the world of DSC is the Local Configuration Manager, or LCM. The LCM is a little piece of software that lives on each computer ("target node") that you manage with DSC. The LCM's job is to read your configuration documents, see if the local computer is in compliance with its configuration document, and potentially fix the computer so that it is compliance. The LCM is really the main "moving part" in DSC.

A secondary player in the DSC game is your authoring computer. This is simply the machine where you sit, and whose keyboard you press with your fingers.

A third and optional player is a DSC Pull Server. This is a normal Windows Server (usually Windows Server 2012 or later) running Internet Information Service (IIS) and a special piece of additional software that is the Pull Server itself. Technically, you can also turn a file server into a Pull Server, but that lacks some of the functionality and security options of an IIS-based Pull Server. In this book, I'll assume you're using the IIS-based Pull Server, rather than a file server. The Pull Server is where you put all your configuration documents, as well as supporting code. You then configure target nodes' LCMs with the location of the Pull Server, and they go to the Pull Server to find their configuration documents. It's a little bit like how client computers "pull" their Group Policy objects (GPOs) from a domain controller.

The Pull Server can also host a component called the Reporting Server. This is a separate web service, also running under IIS, where target nodes' LCMs can check in with a status report. This lets nodes indicate that they're in compliance with their configuration, or that they're awaiting a restart, or that a problem has occurred. It's important, though, to understand the scope of the Reporting Server.

First, that LCM status is just a single-digit code. It isn't a whole problem report, or configuration description, or anything like that. Also, there's no actual… um… "reporting." The Reporting Server tracks node status in a simple local database, and it doesn't provide any native reporting, dashboards, or anything like that. This is the "Technology vs. Tool" thing I pointed out in the Introduction - we have the technology, but no native tools for it.

So those are the players in the game. Now, you need to know the pieces that they play with.

The Pieces

The first and main "piece" in DSC is the Managed Object Format (MOF) file. This is a plain-text file that contains your configuration specifications. It's the "configuration document" that I've loosely referred to. MOF is actually an industry-standard format developed by the Distributed Management Task Force (DMTF), an industry group that includes Microsoft and other vendors. A MOF is not a script - it's simply a bunch of text instructions, written in a specific format that a computer can easily interpret. MOF files aren't terribly human-readable, but you can work through one if you have to. Think of them as a kind of fancy INI file. The important thing to understand is that a MOF is a static file - it doesn't (except for one hair-splitting exception) contain any executable code, any logic, or any decision-making capability. In that way, a MOF is like a Word document - it's just a static description of what a computer should look like.

So how do you create a MOF file? Well, you could use Notepad, if you're the nitty-gritty, I-eat-nails-for-breakfast kind of person. But nobody does that. Another, easier way to create a MOF is a *PowerShell configuration script*. A configuration script is a special kind of script that, when you run it, produces one or more MOF files. Configuration scripts can have incredibly complex logic that governs the final MOF contents - or a configuration script can be essentially static, contain no logic, and produce the same MOF file every single time you run it.

There's a special flavor of MOF called a *meta-configuration MOF*. Again, these can be created in Notepad or, more likely, by writing and running a PowerShell configuration script. These meta-MOFs are used to change the configuration of the LCM itself. They're how you tell the LCM how often to run, where its Pull Server lives, and so on.

The second "piece" in DSC is the collection of *DSC resources* used by the LCM. A resource is usually just a PowerShell script, although they can also be written in a .NET language like C# and compiled into a DLL. One or more resources can be collected into a *DSC resource module*, which is what you actually distribute. Your authoring computer will need a copy of any DSC resource modules that you plan to use, and each target node will need a copy of any DSC resource modules that are used by its configuration.

DSC resources provide the instructions for how the LCM checks the node's current configuration, and fixes the configuration if needed. Each DSC resource consists of three basic components: Get, Set, and Test. So let's say you have a resource designed to handle the installation and removal of Windows features (this exists - it's the WindowsFeature resource). Your MOF might instruct the LCM to ensure that the Web-Server Windows feature is present. When the LCM reads that MOF,

it will load the WindowsFeature resource and run its Test function, passing the Web-Server feature name. The Test function will see if Web-Server is installed or not, and return either True or False. If Test returns False, then the LCM knows that the node is not in compliance with the configuration MOF. If it's been configured to fix that, then the LCM will call the Set function of WindowsFeature, again passing the Web-Server feature name. The Set function will install the feature, bringing the node into compliance.

To summarize that: the MOF contains a description of what you want the node to look like, and the DSC resources know how to check the configuration and, if necessary, fix it. So your abilities with DSC are limited only by the DSC resources that you have. Anything you have a resource for, you can manage via DSC. And, because DSC resources are just PowerShell scripts, you can always create a DSC resource for whatever you might need.

The CIM Connection

It's interesting to know that, under the hood, DSC is based almost entirely in CIM. For example, when you run a command to send a configuration MOF to a node, that connectivity happens over a CIM session, just like the ones used by commands like Get-CimInstance.

Uniqueness in MOFs

There's an important technical detail about MOFs that you need to know before we go any further. Take a look at this chunk of a PowerShell configuration script:

```
1 WindowsFeature DnsServer {
2         Name = 'DNS'
3         Ensure = 'Present'
4 }
```

This is what I call a **configuration setting**. It specifies the configuration for one specific item. The **WindowsFeature** part of that is the name of a resource, which would be contained in a resource module, and which would be locally installed on the node. Following that is the **name** for this configuration setting. I made this up - "DnsServer" is meaningful to me, so that's what I used. This name does need to be unique - in the final MOF, no other configuration setting can use the same name. But I could also have used "Fred" or "Whatever" as legal names - DSC doesn't care what the name is, so long as it's unique.

Inside the configuration setting are its **properties**, including Name and Ensure. These properties and their values will be passed to the DSC resource by the LCM, and tell the resource what, specifically, it's supposed to be checking and doing. Every DSC resource is designed to have one or more *keys*, which are the properties they use to identify some manageable component of the computer. In the

case of the WindowsFeature resource, the key happens to be the **Name** property. Here, I've specified "DNS" for the name, because "DNS" is the name of the Windows Feature I want installed.

Because **Name** was designed to be the key for the WindowsFeature resource, *it must be unique for all instances of the WindowsFeature resource within the final MOF*. This restriction is designed to stop you from doing stuff like this:

```
WindowsFeature DnsServer1 {
        Name = 'DNS'
        Ensure = 'Present'
}
WindowsFeature DnsServer2 {
        Name = 'DNS'
        Ensure = 'Absent'
}
```

This doesn't make any sense! You want DNS installed - but not installed? Make up your mind! Because this kind of configuration can throw the LCM into an infinite loop of installing and uninstalling, DSC won't allow it. In fact, DSC will detect the conflict upon trying to create a MOF from a configuration with these settings, an error will be returned, and a MOF will not be output.

This is really, really important - and it can be tricky, because as you physically write a configuration script, there's no easy or obvious way to see exactly what the key properties are. You *can* look it up, and I'll show you how, and you have to be very careful not to create duplicates.

Getting the MOF to the LCM

There are three ways of getting a MOF to an LCM:

- Manual file copy. You simply have to drop the MOF in the correct location (and run a command to encrypt it, but we'll get into that later). The LCM will see the new MOF, and will process it on the next run.
- Push mode. Usually from your authoring computer, you'll run Start-DscConfiguration to *push* one or more MOF files to target nodes, with a maximum of one MOF per node. This is a one-time operation that relies on Remoting to provide the connection to the nodes.
- Pull mode. If you configure the LCM to do so, it can look to one (or more, which I'll explain in a bit) Pull Server(s) for configuration MOFs. By default, the LCM will check for a new MOF every 30 minutes. If it finds one, it'll download and start processing the new MOF.

Pull Servers can also contain ZIPped copies of DSC resource modules. When a node gets a MOF, it does a quick pre-scan of the MOF to make sure all needed DSC resources are already locally installed. If they're not, then the LCM can be configured to download needed DSC resource modules from

a Pull Server. The LCM then unzips and installs those locally, and then runs its MOF. The LCM can be configured to "pull" resource modules in this way, even if it's not "pulling" MOFs. In other words, you can configure the LCM to wait for you to *push* it a MOF, but still download needed DSC resource modules from a Pull Server.

Configuration Variations

So far, I've described the DSC process with the assumption that you're creating one MOF for each target node. That's a good way to understand how DSC works, but it obviously might not be ideal for every environment. After all, you probably have many different kinds of servers, but a lot of them probably have many similarities. So some of the first design questions people often ask about DSC tend to revolve around modularizing the MOFs.

By the way, in many of the following examples, I'm going to be referring to DSC resources that are fictional, and even using real ones with values that might not work in the real world. The goal of this chapter is to help you understand the design decisions you need to make, not to provide a reference of resources or syntax. So I've tried to construct these examples in a way that best illustrates the *concepts*. I just wanted you to know that, so you don't try copying these into a real configuration. I'll provide plenty of *functional* examples in later chapters.

One MOF, Many Nodes

One thing you can do is simply assign the same MOF to multiple nodes. Provided the MOF doesn't try to configure any must-be-unique information, like computer name or IP address, this is totally legal. You can either push the same MOF to more than one node, or you can configure multiple nodes to pull the same MOF from a Pull Server. This works well in situations where you *want* multiple nodes configured identically, such as a bunch of web servers that are all part of the same load-balanced set.

Configuration Data Blocks

When you run a configuration script, you can pass in a *configuration data* block. This is a data structure that lists the nodes you want to produce MOFs for, and can provide unique, per-node information. In your configuration script, you can then write logic - such as If constructs - that change the output of the MOF file for each node, based upon that configuration data.

For example, you might create a configuration data block that, in part, looks like this:

```
$ConfData =
@{
    AllNodes =
    @(
        @{
            NodeName = "SERVER1";
            Role     = "WebServer"
        },
        @{
            NodeName = "SERVER2";
            Role     = "SQLServer"
        }
                )
}
```

Then, in the configuration script, you might do something like this:

```
if ($node.role -eq "WebServer") {
        WindowsFeature InstallIIS {
                Ensure = 'Present'
                Name = 'Web-Server'
        }
}
```

The MOF files for each of those two nodes would therefore be different. Configuration data blocks can be defined in the same physical file as a configuration script, but more often the data is stored in a separate file, and then loaded and passed to the configuration script when that script is run.

While this technique does enable a single configuration script to handle a wide variety of situations, it also becomes possible for that configuration script to become incredibly complex - and potentially harder to maintain.

Simple Includes

One way to make a complex configuration script easier to maintain is to break it into multiple files. Potentially, some of those chunks could then be reused across multiple configuration scripts. This isn't actually a complex technique; you just use standard PowerShell dot-sourcing. For example, consider the following as a sort of "master" configuration script:

```
configuration DatabaseServer {
        param(
                [string[]]$ComputerName
        )
        node $ComputerName {
                . ./standard-config.ps1
                . ./security-config.ps1
                . ./sql-server-2012-config.ps1
        }
}
```

When run, this configuration script would "include" (dot-source) three different PowerShell scripts. Those scripts would not contain the outer **configuration** block, nor would they contain a **node** block. Instead, they would simply contain configuration settings. As you can see, some of those three - like standard-config.ps1 or security-config.ps1 - might also be used in "master" configuration scripts that applied to different server roles.

This technique can also make use of configuration data blocks, enabling you to add per-node logic to each of the "included" scripts.

Composite Configurations

In a composite configuration, you actually create a complete configuration script. It can even be set up to accept input parameters. For example, consider this extremely simple configuration script:

```
configuration customsyncfile {
        param(
                [string]$FilePath
        )
        node {
                SyncFile SyncFile1 {
                        SourceFilePath = $FilePath
                        DestFilePath = c:\inetpub\wwwroot
                        SyncMode = 'Continuous'
                        SyncMethod = 'FileHash'
                        SyncTimeout = 5000
                        SyncWait = 100
                }
        }
}
```

In this example, the configuration has been set up to accept a single input parameter, **FilePath**. Inside the configuration, I've referred to a (fictional) DSC resource named **SyncFile**. It obviously accepts a

number of properties; I've provided values for each of these, but the **SourceFilePath** property will use whatever was given to the configuration's **FilePath** input parameter.

This configuration would then be saved in a way that *made it look like a DSC resource* named "customsyncfile." This then becomes a *component configuration*. I would then write a second, *composite configuration*, perhaps like this:

```
configuration WebServer {
        param(
                [string[]]$ComputerName
        )
        node $ComputerName {

                customsyncfile WebServerFiles {
                        FilePath = '\\dfsroot\webserverfiles\sources\source1'
                }

                WindowsFeature IIS {
                        Name = 'Web-Server'
                        Ensure = 'Present'
                }

        }
}
```

My composite configuration has two configuration settings, one of which is using the WindowsFeature DSC resource, and the other of which is using my **customfilesync** component resource. As you can see, I'm able to provide a **FilePath** to the component resource, which will be used as its **SourceFilePath** internally. I don't need to specify any other properties for **customfilesync**, because it wasn't set up to accept any additional input. Everything else that the underlying **SyncFile** DSC resource needs has already been hardcoded into the component configuration.

Composite configurations are a good way to modularize configuration elements. A component configuration can contain as much or as little as you need. You could, for example, have an entire component configuration that took care of every configuration element that your organization's IT Security team needed, while another one handled installation of standardized elements like management agents, encryption certificates, and so on. Each component configuration could be "owned" by a different team within your organization, helping to provide some separation of duties.

Keep in mind that the component configurations do need to be present on the target nodes that use them - any DSC resources used *in* the component configurations also need to be installed. Component configurations can, however, be ZIPped up and distributed by means of a Pull Server, just like any DSC resource module.

I'll cover the technical details of composite configurations in a later chapter.

Partial Configurations

Partial configurations are a new feature in PowerShell v5, and they're another way of modularizing configurations. They work best with a Pull Server setup, although it's possible to push partial configurations as well, and you can also have a mixed mode where you push some partials, and let the node pull other partials.

So what's a partial configuration? Basically, you just create a bunch of different configuration scripts, run them to produce MOFs, and drop all the MOFs onto a Pull Server. You then configure a node's LCM to pull one or more of those MOFs. That configuration gets into some technical details that I'll cover later; for now, just understand that a single node can pull multiple MOFs. Some of those MOFs might be unique to that node, while others might be shared across many nodes.

When the LCM grabs all these MOFs, it first merges them into a single MOF. That's a bit more complex than just concatenating the files, but the LCM knows what to do.

But a lot of us in the community have mixed feelings about partial configurations, and our trepidation comes from the uniqueness constraints I described earlier in this chapter. When you're authoring partial MOFs, it becomes really easy to *accidentally* reuse the same configuration setting name. It also becomes easy for two separate MOFs to contain duplicate resource key properties. For example, MOF A might set the IIS WindowsFeature to install, while MOF B sets the IIS WindowsFeature to uninstall. You won't necessarily be aware of that as you sit on your authoring computer, but once the LCM merges the MOFs, it'll detect the conflicts and completely abort the configuration.

The problem we have with this is that the problem is occurring as far from you, the administrator, as possible. It's happening in the context of a background scheduled task, running on a remote computer. Debugging that situation - or even realizing that the LCM is exploding - can be hard. Right now, we don't have any way of *locally* saying, "hey, suppose I send these 10 MOFs to a node - will any uniqueness problems pop up?" So there's no local "combine and test" or "what if" that you can run.

Our feeling is that a smarter Pull Server is what's needed. A node could check in, and say, "Hey, I need this list of 8 MOFs, please." The Pull Server could then load those off disk, *combine them on the server*, and send the merged MOF to the LCM. But the Pull Server could also detect any conflicts (and potentially even run some dynamic logic to alter the merged MOF based on criteria you provide). Any problems would be happening in one central place - the Pull Server - and could be logged, easier to get to, and easier to troubleshoot and solve. Sadly, right now, we don't have this kind of Pull Server.

In case you're still interested in partial configurations, I'll cover their technical details in an upcoming chapter.

The Choice of Not Choosing

None of the above is meant to give you the impression that composites, includes, partials, or anything else are somehow mutually exclusive. They're not. You can have a partial configuration that uses

a dot-sourced inclusion, and that is a composite configuration that relies on underlying component configurations. Any of those can have logic based on configuration data blocks. You can do it all.

What's the Best Practice?

Well… other than the aforementioned concerns around partial configurations, there really *aren't* best practices. None of these things is inherently good or bad. Different organizations work differently and have different requirements and different preferences. DSC gives you different approaches that you can mix and match to help meet your unique situation.

Understanding Dependencies

There's one more design thing you need to understand about configurations, and that's the order in which the LCM executes configuration settings from a MOF. Although today's LCM is single-threaded, Microsoft envisions a potential multi-threaded LCM someday. For that reason, you can't rely on the LCM simply executing configuration settings from the top of the MOF through to the bottom, as you might expect. But you will definitely have situations where you need such-and-such a configuration to finish before some other configuration. For example, you might want to copy an installer package to the node, and *then* run the installer. Doing those in reverse won't work, and so you need some way of telling the LCM which one to do first. That way is called DependsOn.

```
WindowsFeature ActiveDirectoryFeature {
        Name = 'ADDS'
        Ensure = 'Present'
}

ADDomainController MyDC {
        Ensure = 'Present'
        DomainName = 'MyCompany'
        DCType = 'Regular'
        DependsOn = [WindowsFeature]ActiveDirectoryFeature
}
```

Here, I've specified a (fictional) DSC resource that sets up an AD domain controller. However, that configuration setting can't run until the WindowsFeature resource named "ActiveDirectoryFeature" has run.

You can also specify that a configuration setting not run until *multiple* other settings have run:

```
DependsOn = [WindowsFeature]IIS,[WindowsFeature]DNS
```

When the LCM executes its MOF, it will scan for these dependencies and sort the configuration settings accordingly. Note that there are some caveats when it comes to using DependsOn in a composite resource; I'll cover those in more detail in the chapter on composite configurations.

Part 2: Configuring the Infrastructure

With some of your design decisions made, you can start getting things set up in your environment. In this Part, we'll discuss setting up a Pull Server and configuring the LCM. Now, *we're assuming you've already started making some design decisions*, and as you read through these chapters, you should be able to start finalizing them. But one problem with DSC is that design can be a little of a chicken-and-the-egg situation. You probably won't know how you want your LCMs configured until you know how you're going to create and modularize your configurations. So while we're going to show you all the possibilities, we can't *recommend* one to you - you may need to work through a bit more of the book so that you can start making some concrete decisions for your environment.

Infrastructure Prerequisites

There are a few things that we assume you have in place in order for DSC to work. It's not a huge list, but it's an important one.

- All nodes that will be used to author, push, or receive configurations are running PowerShell v5 or later. Keep in mind, this book doesn't cover v4.
- WS-Management (WS-MAN) traffic is permitted on the network. This isn't PowerShell Remoting; DSC uses CIM, which also runs over the WS-MAN protocol. It will be enabled by default on nodes that have PowerShell v5 installed, but you must ensure it's not being blocked by firewalls or other network elements.
- For authoring nodes, we assume that they have Internet connectivity to download new DSC-related resources, and that they have the PowerShell ISE installed (which would be the case by default on Windows client computers).

Configuring the LCM

The Local Configuration Manager, or LCM, is the "agent" for DSC. It lives and runs on each target node, accepts configurations (via either pull or push), and executes those configurations. In WMF v4, the LCM ran as a Scheduled Task; as of v5, that's not the case. You can't schedule *when* the LCM will run, but you can schedule some parameters related to *how often* it runs. That's what this chapter is all about, along with the LCM's other configuration options.

Checking the Configuration

Run Get-DscLocalConfigurationManager to see the current configuration. Here's a default Windows 10 computer (in early July 2016; defaults may change in subsequent updates, so it's important not to treat this example as a reference):

```
ActionAfterReboot              : ContinueConfiguration
AllowModuleOverWrite           : False
CertificateID                  :
ConfigurationDownloadManagers  : {}
ConfigurationID                :
ConfigurationMode              : ApplyAndMonitor
ConfigurationModeFrequencyMins : 15
Credential                     :
DebugMode                      : {NONE}
DownloadManagerCustomData      :
DownloadManagerName            :
LCMCompatibleVersions          : {1.0, 2.0}
LCMState                       : Idle
LCMStateDetail                 :
LCMVersion                     : 2.0
StatusRetentionTimeInDays      : 10
PartialConfigurations          :
RebootNodeIfNeeded             : False
RefreshFrequencyMins           : 30
RefreshMode                    : PUSH
ReportManagers                 : {}
ResourceModuleManagers         : {}
PSComputerName                 :
```

Let's walk through each of those. But, before we do that, I have to point out that this set of configuration properties is from a *Windows* computer. Microsoft produces an LCM for Linux, too, and that LCM has a smaller set of properties (at least, as of this writing; it's unclear at this point how hard Microsoft will try to keep the Linux LCM in feature-lockstep with the Windows LCM).

Also, notice the lack of an AgentId - that's because this is a fresh machine, and it doesn't yet have one.

ActionAfterReboot

This determines what the LCM will do after the computer is rebooted. The default, ContinueConfiguration, will let the LCM pick up where it left off and continue processing the current configuration. You can also choose StopConfiguration, which, as the name implies, stops the configuration from running further. I imagine you'd really only use StopConfiguration in certain troubleshooting or lab situations.

AllowModuleOverWrite

When set to $False - which is the default - the LCM *will not download any DSC resource modules that already exist locally*. You might wonder why the heck that's the default. Well, it's for safety. The presumption is that, once you put a resource onto a node, the node depends on that exact resource in order to run configurations properly. If you change or update a resource, you're meant to apply a new version to it. Your configurations can then reference that new version, and if properly configured, the LCM will download that new version. This AllowModuleOverwrite setting only applies to downloading new copies of the exact same module. In a test environment, where you're frequently iterating modules, you'll probably set this to $True.

Why would you not set this to $True in production? Well... predictability and stability. If an LCM is allowed to overwrite local modules, that can be convenient for you when making ad-hoc changes. But that subverts the entire best practice of code testing, versioning, and so on. Ideally, if Configuration A relies on Module B version 1.2, you don't want Module B version 1.3 showing up out of the blue. It might not work the way Configuration A expects - for example, Module B version 1.3 might require different properties or property values, which Configuration A doesn't provide. The correct practice would be to deploy Module B v1.3 as a separate thing, living independently of, and side-by-side with, v1.2. You would then author Configuration C, which specifically references Module B v1.3, and deploy Configuration C to your nodes. That way you always know what each node is using, and troubleshooting becomes a lot easier.

CertificateID

This is the certificate thumbprint (which you can obtain by exploring the CERT: drive) of the encryption certificate that the LCM will use to decrypt any encrypted information from its MOF. Most commonly, this is used to decrypt encrypted logon credentials. We'll cover certificates pretty thoroughly in an upcoming chapter.

ConfigurationDownloadManagers

This is a collection of pull servers - that is, servers from which MOF files will be pulled.

ConfigurationID

This is a Globally Unique Identifier (GUID) that you assign to the LCM. In WMF v4, this was the only way to tell the LCM which MOF to pull from a pull server. In WMF v5, you can also use configuration names. However, when using ConfigurationID, the node will essentially look on the pull server for a MOF file whose filename is the same GUID. Note that a GUID isn't just a random string of hexadecimal characters; it's a specific thing that must be generated. In PowerShell, you can use [guid]::newGuid() to generate one. **This is really important**: If you specify a ConfigurationID, you can run against a v4 or v5 pull server, and the v5 pull server will follow v4 behaviors. If you omit the ConfigurationID, whether you include Configuration Names or not, the pull server must be v5 or later, will behave as v5 or later, and requires a registration key to be specified in a Repository section.

ConfigurationMode

This controls the LCM's actions. The following values are allowed:

- Disabled. The LCM does not run. This is perhaps most often used in cases where a third-party management technology, like Chef, is actually running the show. Chef can use DSC resource modules under the hood, but it doesn't want the actual LCM stepping in and interfering.
- ApplyOnly. The LCM applies the current configuration, and then stops running until manually run.
- ApplyAndMonitor. The LCM applies the current configuration, and re-checks it on the ConfigurationModeFrequencyMinutes value. It does not attempt to fix the configuration, but will report a status to a Reporting Server if so configured.
- ApplyAndAutoCorrect. The LCM applies the current configuration, and re-checks it on the ConfigurationModeFrequencyMinutes value. It will attempt to fix out-of-compliance configuration settings and, if configured, will report status to a Reporting Server.

ConfigurationModeFrequencyMins

This controls how often the LCM will re-evaluate its current configuration. It defaults to, and has a minimum value of, 15 minutes. This is important, conceptually, because it means the LCM cannot guarantee *continuous* compliance with your desired configuration. If someone changes something, that change can "last" for 15+ minutes until the next consistency check. So in cases where the "desired configuration" could mean the difference between literal life or death, or someone going to jail or not, DSC might not be the right technology. In *many* cases, you could extend this to 24 hours or more (calculated in minutes) for cases when you simply need a once-a-day check on configuration compliance.

Credential

This appears to be obsolete, since the individual resource blocks (which we'll get into in a bit) can define this on a per-instance basis.

DebugMode

Whether or not the LCM's debug mode is on or off ($True or $False).

DownloadManagerCustomData

This appears to be obsolete in v5.

DownloadManagerName

This appears to be obsolete in v5.

LCMCompatibleVersions

The LCM versions that this LCM is compatible with. For example, LCM 2.0, which ships in WMF v5, is compatible with LCM 1.0, meaning it can accept the same configuration documents that the older LCM accepted, in addition to newer, 2.0-only ones.

LCMState

This tells you what the LCM is currently up to:

- Idle
- Busy
- PendingReboot
- PendingConfiguration

LCMStateDetail

This is usually a more descriptive version of what the LCM is doing right then, like "LCM is applying a new configuration."

LCMVersion

The version of the LCM.

StatusRetentionTimeInDays

This controls how long the LCM will retain current status information. The default is 10 days.

PartialConfigurations

This doesn't presently do anything.

RebootNodeIfNeeded

When set to $True, the LCM will automatically restart the target node if a DSC resource indicates that a restart is required to finalize the configuration. This is set to $False by default, on the very reasonable grounds that most people want a little more control over when their servers restart.

RefreshFrequencyMins

This controls how often the LCM will check the pull server for new MOFs. It defaults to 30 minutes, and in WMF v4 must be set to an even multiple of the ConfigurationModeFrequencyMinutes value (15 * 2 = 30, hence the default of 30). This value should always be larger than the Configuration-ModeFrequencyMinutes.

RefreshMode

Push or Pull. As easy as that. This defaults to Push, meaning that the LCM just sits and waits for you to hit it up with a configuration.

ReportManagers

This is a collection of configured reporting servers.

ResourceModuleManagers

This is a collection of pull servers from which modules (DSC resources) can be downloaded.

Changing the Configuration

To change the configuration of an LCM, even the local one, you need a *meta-MOF*. This is a special MOF that you can create by writing a special PowerShell configuration script. Meta-MOFs are pushed to their nodes, using the Set-DscLocalConfigurationManager cmdlet (which uses CIM sessions for connectivity). A meta-MOF configuration script might look something like the following:

```
[DSCLocalConfigurationManager()]
configuration LCMConfig
{
    Node Node1
    {
        Settings
        {
            RefreshMode = 'Push'
        }
    }
}
```

In this case, the [DSCLocalConfigurationManager()] bit tells PowerShell that this is to be a meta-MOF, which is generated somewhat differently than a normal configuration MOF. The name of the configuration, LCMConfig, is arbitrary. You simply need to specify that name - usually at the very bottom of the script file - to run the config and generate the MOF (it's basically like running a function). Within the Settings{} block is where you can put whatever settings you like from the list above.

Deploying the LCM Configuration

The file Node1.meta.mof will be created in a subfolder named LCMConfig. Once you've run that and generated a Node1.meta.mof, you can use Set-DscLocalConfigurationManager to "push" the meta-MOF to Node1.

```
Set-DscLocalConfigurationManager -Path ./LCMConfig -ComputerName Node1 -verbose
```

It's also possible to "inject" the meta-MOF into a node by copying it into the place where the LCM looks for it (C:/Windows/System32/Configuration/metaconfig.mof). For example, this is a great way to help configure the LCM as part of an automated VM build, since you can use things like the Hyper-V integration cmdlets to copy files into the VM.

NOTE: I should point out that there can be a bunch of caveats around "injecting" a MOF by simply copying a file. I've found it's nearly always better to push the meta-MOF to the LCM as part of an automated build process, or just to copy the meta-MOF to an arbitrary local file on the node and have the VM "push" that to itself. And keep in mind that you can also "sort of push" a meta-MOF into a VM node, even if WS-MAN isn't enabled on the node, by using integration components like those provided by Hyper-V. If you want to read more on injection, though, have a look at http://dille.name/blog/2014/12/07/injecting-powershell-dsc-meta-and-node-configurations/ and https://blogs.msdn.microsoft.com/powershell/2014/02/28/want-to-automatically-configure-your-machines-using-dsc-at-initial-boot-up/, although those articles refer to WMF v4, so the actual examples are slightly outdated.

Specifying Configuration Pull Servers

The basic settings I covered above are only the beginning of what you'll need to configure. One thing you're probably going to want to do is configure one or more pull servers, from which the LCM will pull its configuration MOF (or MOFs, if you're using partial configurations). So in addition to the Settings{} block in the meta-MOF, you can have one or more pull server sections.

```
[DSCLocalConfigurationManager()]
configuration LCMConfig
{
    Node localhost
    {
                Settings {
                        RefreshMode = 'Pull'
                }

                ConfigurationRepositoryWeb MainPull {
                        AllowUnsecureConnection = $false
                        ConfigurationNames = @('Server2Configuration')
                        RegistrationKey = '140a952b-b9d6-406b-b416-e0f759c9c0e4'
                        ServerURL = 'https://mypullserver.company.pri/PSDSCPullServer.svc'
                }

                ConfigurationRepositoryShare FilePull {
                        SourcePath = '\\organization\path\folder\'
                }
        }
}
```

The above example shows both a Web and a file share-based pull server configuration. You can mix and match; note that the file share-based version also allows a Credential property, where you can specify a credential to use to access the share (if you don't, then the share must allow anonymous, un-authenticated access). Frankly, file share-based pull servers are rare in my experience, and they offer less functionality. I'll assume you're using a Web-based pull server, which means the pull server is running IIS, and you've installed the DSC Pull Server feature on the server (we'll cover pull server setup in an upcoming chapter).

Also notice that each block has a name. This is arbitrary, and meaningful to me; DSC doesn't care what you name them. I've used "MainPull" and "FilePull."

Web pull servers support these properties:

- AllowUnsecureConnection. This should always be $False, and is by default. This is a lot less about ensuring the connection is encrypted (although it also means that), and more about

ensuring that the node is connecting to a known, trusted pull server. With this set to $True, it becomes very easy to use man-in-the-middle attacks to inject malicious configurations into your environment. When you wind up on CNN because you got p0wned by means of DSC, just remember that I Told You So, and stop using unsecure connections.

- CertificateID. This is the GUID (thumbprint) of a local certificate, which the node will send to the pull server to authenticate the node to the server. This is a great way to prevent unknown nodes from even trying to contact your pull server. This is nothing fancier than good old IIS certificate-based authentication.
- ConfigurationNames. When set, this means you should remove the ConfigurationID GUID from the Settings{} block, switching the node over to requesting one or more MOFs by filename. Notice this is specified as an array. When you use this, you must also specify the RegistrationKey. You must also use a pull server that has WMF v5. Finally, note that while ConfigurationNames accepts an array, it is illegal to specify more than one, unless you are specifying partial configurations (covered in a bit).
- RegistrationKey. This is a GUID (use [guid]::newGuid() to generate it) that is configured in advance on the pull server. It acts as a kind of password for the node to "log in" and retrieve MOFs. The theory is that, when you just use a ConfigurationID (GUID), it's really hard for a bad actor to guess at your MOF filenames. But when you start using ConfigurationNames, which are likely to be meaningful words and phrases, guessing filenames becomes easier. The RegistrationKey makes it harder for a bad guy to download your MOFs and inspect them (which would teach them a lot about your environment and facilitate an attack). Note that I've seen some documentation which suggests that you can remove RegistrationKey once the node is registered; in analyzing the code for the Linux LCM, and in speaking with some of the team, that's not true. The RegistrationKey is used every time the node runs a consistency check and hits the Pull Server.
- ServerURL. The URL of the pull server, which should start with https:// unless you're okay being hacked. And yes, that means the IIS web site that the pull server is running under must have an SSL certificate that is issued by a root Certificate Authority that the node trusts. The URL must end in 'PSDSCPullServer.svc' if you're using the Microsoft-supplied pull server service.
- SignedItemType. New in WMF 5.1, this lets you tell the LCM to expect digital signatures from files delivered from the pull server or pushed to the LCM. This can be "Configuration", "Module", or both (e.g., "Configuration","Module"). See the chapter, "Security and DSC," for information on digital signing for these items.
- TrustedStorePath. New in WMF 5.1, this goes along with SignedItemType. This is optional; if you don't include it, the LCM will use its normal certificate trust procedures to verify signed items. However, if you provide this (e.g., CERT:/LocalMachine/DSCStore), then the LCM will use that path to retrieve trusted publishers. *This is an excellent idea*, because you can restrict the CAs that are trusted to only those you control.

Notice that only SMB (file share) pull servers support a credential; Web pull servers do not. But Web pull servers support configuration names, certificate authentication, encryption, and lots more. They are Better In Every Way.

Specifying DSC Resource Pull Servers

Even when configured in Push mode, a node can be configured to download DSC resource modules from a pull server. This relieves you of the burden of manually deploying resource modules. You do not need to set up a unique pull server for this purpose; whatever pull server you've already got will do fine.

```
[DSCLocalConfigurationManager()]
configuration LCMConfig
{
    Node localhost
    {
                Settings {
                        RefreshMode = 'Pull'
                }

                ConfigurationRepositoryWeb MainPull {
                        AllowUnsecureConnection = $false
                        ConfigurationNames = @('Server2Configuration')
                        RegistrationKey = '140a952b-b9d6-406b-b416-e0f759c9c0e4'
                        ServerURL = 'https://mypullserver.company.pri/PSDSCPullServer.svc'
                }

                ResourceRepositoryWeb ModuleSource {
                        AllowUnsecureConnection        = $false
                        RegistrationKey = '140a952b-b9d6-406b-b416-e0f759c9c0e4'
                        ServerURL = 'https://mypullserver.company.pri/PSDSCPullServer.svc'
                }

        }
}
```

Notice that the RegistrationKey and ServerURL are the same in both the ConfigurationRepository-Web and ResourceRepositoryWeb blocks? I'm using the exact same pull server. I can also specify a CertificateID setting, if the pull server requires client certificate authentication. You can also define a ResourceRepositoryShare{} block, which would be for a file share-based pull server, and it accepts Credential and SourcePath properties.

Again: it's totally legal in WMF v5 to specify a ResourceRepositoryWeb{} block, completely omit any other configurations, and leave the LCM in Push mode. When you Push a configuration to the node, it'll scan to see what resources are needed. If it's missing any of those resources, the node will connect to the defined server and attempt to download the ZIPped resource module. It'll then unzip it and proceed with the configuration.

Specifying Reporting Servers

A reporting server provides a check-in point for the node. This is always Web-based, meaning it runs under IIS. It can be the exact same server as a Web-based pull server, if you want.

```
[DSCLocalConfigurationManager()]
configuration LCMConfig
{
    Node localhost
    {
                Settings {
                        RefreshMode = 'Pull'
                }

                ConfigurationRepositoryWeb MainPull {
                        AllowUnsecureConnection = $false
                        ConfigurationNames = @('Server2Configuration')
                        RegistrationKey = '140a952b-b9d6-406b-b416-e0f759c9c0e4'
                        ServerURL = 'https://mypullserver.company.pri/PSDSCPullServer.svc'
                }

                ResourceRepositoryWeb ModuleSource {
                        AllowUnsecureConnection        = $false
                        RegistrationKey = '140a952b-b9d6-406b-b416-e0f759c9c0e4'
                        ServerURL = 'https://mypullserver.company.pri/PSDSCPullServer.svc'
                }

                ReportServerWeb CheckInPoint {
                        AllowUnsecureConnection = $false
                        RegistrationKey = '140a952b-b9d6-406b-b416-e0f759c9c0e4'
                        ServerURL = 'https://mypullserver.company.pri/PSDSCPullServer.svc'
                }

        }
}
```

You can also specify a CertificateID, if the server requires client certificate authentication. Notice that the service endpoint - PSDSCPullServer.svc - is the same as it was for the regular pull server function. That's a change in WMF v5; previously, the "compliance server," as it was then known, had its own endpoint.

Partial Configurations

The last thing you might want to do with the LCM is to configure partial configurations. We've outlined previously what these are (and why we're not sure they're a good idea).

It's worth noting that partials can get really complex. What we're showing here is an illustration; we're not sure it's something you'd find in a real-world environment. We're just trying to convey the sense of the thing.

Here's how you set them up:

```
[DSCLocalConfigurationManager()]
configuration LCMConfig
{
    Node localhost
    {
                Settings {
                        RefreshMode = 'Pull'
                }

                ConfigurationRepositoryWeb MainPull {
                        AllowUnsecureConnection = $false
                        ConfigurationNames = @('PartialOne','PartialTwo')
                        RegistrationKey = '140a952b-b9d6-406b-b416-e0f759c9c0e4'
                        ServerURL = 'https://mypullserver.company.pri/PSDSCPullServer.svc'
                }

                ResourceRepositoryWeb ModuleSource {
                        AllowUnsecureConnection        = $false
                        RegistrationKey = '140a952b-b9d6-406b-b416-e0f759c9c0e4'
                        ServerURL = 'https://mypullserver.company.pri/PSDSCPullServer.svc'
                }

                ReportServerWeb CheckInPoint {
                        AllowUnsecureConnection = $false
                        RegistrationKey = '140a952b-b9d6-406b-b416-e0f759c9c0e4'
                        ServerURL = 'https://mypullserver.company.pri/PSDSCPullServer.svc'
                }

                PartialConfiguration PartialOne {
                        ConfigurationSource = @('[ConfigurationRepositoryWeb]MainPull')
                        Description = 'PartialOne'
                        RefreshMode = 'Pull'
```

```
33                      ResourceModuleSource = @('[ResourceRepositoryWeb]ModuleSource')
34                  }
35
36                  PartialConfiguration PartialTwo {
37                      ConfigurationSource = @('[ConfigurationRepositoryWeb]MainPull')
38                      DependsOn = '[PartialConfiguration]PartialOne'
39                      Description = 'PartialTwo'
40                      ExclusiveResources = @('OrderEntryAdmin','OrderEntryConfig')
41                      RefreshMode = 'Pull'
42                      ResourceModuleSource = @('[ResourceRepositoryWeb]ModuleSource')
43                  }
44
45          }
46  }
```

Here's what to pay attention to:

- I've defined two partial configurations, which I've arbitrarily named PartialOne and PartialTwo. If you go back and look at the pull server configuration, you'll see that these are now listed as the ConfigurationNames for the node. PartialOne.mof and PartialTwo.mof must live on that pull server, along with their checksum files.
- Each partial configuration defines which pull server it'll come from. You have to refer to a pull server that you already defined - in this case, I've only defined MainPull, and so both partials must come from there. Again, PartialOne.mof and PartialTwo.mof must live on that pull server, along with their checksum files.
- PartialTwo won't execute until PartialOne has finished - the DependsOn sees to that.
- PartialTwo is the only configuration that can contain settings using the OrderEntryAdmin and OrderEntryConfig DSC resources. No other configuration may use those resources, due to the ExclusiveResources setting.
- Both partials specify my ModuleSource definition as the place to find any missing DSC resource modules.
- Both partials specify that the configuration be pulled. However, it's legal for some partials to expect to be pushed. Any partials set to Push mode will simply sit and wait for you to push the MOF manually. They wouldn't have a ConfigurationSource, in that case.

As you can see, these can become *insanely* complicated and hard to keep track of. In addition, as I've pointed out before, the actual partial configuration MOFs must be carefully - manually! - evaluated to ensure they don't contain conflicting keys or other problems. The LCM will download all the MOFs, combine them, and then scan them for problems. Any problems will result in the entire configuration - all partials! - being abandoned.

Note that you can also configure a partial configuration to be pushed, making it possible to push some, and pull some, of the total configuration. When a partial is configured for push, the node

will always use the most recently pushed one when it combines all of its partials into the total configuration.

```
[DSCLocalConfigurationManager()]
configuration PartialConfigDemo
{
    Node localhost
    {

        PartialConfiguration PushPart {
            Description = 'Configuration being pushed'
            RefreshMode = 'Push'
        }
    }
}
```

That's how the LCM would be configured to understand that one of its partial configurations will be pushed to it. The configuration you push - e.g., the filename - must match. So, in this case, I'd have to push pushpart.mof. I'd use **Publish-DscConfiguration** to push the file to the node, and then run **Update-DscConfiguration** to let any additional partials (obviously not shown in the above snippet) be pulled, and for the consistency check to begin.

And actually... that's an oversimplification, the file naming bit. Assuming the node is set to use ConfigurationNames, the filename above would be pushpart.mof. But, if you'd set the LCM to use a ConfigurationID (that is, a GUID), you'd name it pushpart.GUID.mof, inserting the GUID in the filename.

You can also sequence your partial configurations. Check this out:

```
[DSCLocalConfigurationManager()]
configuration PartialConfigDemo
{
    Node localhost
    {

            ConfigurationRepositoryWeb MyPull {
                    ServerURL = 'https://whatever:8080/PSDSCPullServer.svc'
            }

        PartialConfiguration PushPart1 {
            Description = 'Configuration being pushed'
            RefreshMode = 'Push'
        }

```

```
        PartialConfiguration PushPart2 {
            Description = 'Configuration being pulled'
            RefreshMode = 'Pull'
            ConfigurationSource = '[ConfigurationRepositoryWeb]MyPull'
            DependsOn = '[PartialConfiguration]PushPart1'
        }
    }
}
```

I'm still omitting the Setting{} block in this snippet, but you can see that PushPart2 won't be applied until after PushPart1.

Versions and Troubleshooting

The syntax I've used in this chapter is all WMF v5-specific. That said, v5 LCMs can be configured using the older v4 syntax, as well. Keep in mind, however, than a v5 LCM cannot be configured to use ConfigurationNames (and registration keys) if it's talking to a v4 pull server.

If you run into problems configuring the LCM, **simplify**. Start by seeing if you can just get it running in Pull mode, using a normal configuration ID rather than a configuration name. Doing so eliminates the whole registration key process, for example, thus eliminating a possible point of trouble.

Setting Up a Pull Server

By now, you should have almost all the basics that you need to begin using DSC. At this point in the book, I'm going to start shifting from simple examples over to more production-scale ones, and that means we're going to need a pull server. *True*, you won't always use a pull server in production. Sometimes, you'll doubtlessly have nodes to which you simply push configurations. That's fine - push mode is easy. But in what I expect will be many more cases, you'll want to use pull mode, and so that's where I'll start.

Before You Begin

Know that Microsoft has designated the "native" Pull Server as EOL, or "End of Life."[5] That means they're not developing any further features, and may not support it for as long as you might want. Their "official" answer is for you to use the DSC services built into Azure Automation, or to find a third-party alternative (like the "Tug" open-source pull server).

Reprising the Roles

Remember that a single computer, physical or virtual, can run more than one "pull server." After all, a pull server is just an IIS website, and IIS can certainly host more than one website. Should you want to host multiples, you'll have to figure out how to differentiate them. That's a purely IIS question, though. Each website needs some unique combination of IP address, port, and host name, so you can set that up however you want. Since this book isn't about getting gnarly with IIS, I'm going to stick with a single website.

However, DSC does support a second feature, the Reporting Server, which I'll also set up. If you don't remember all the caveats about the Reporting Server, do go back and read up, so that you have a reasonable expectation for what it can do.

A Word of Caution

We'll be using the xPSDesiredStateConfiguration module to create a pull server. Thing is, some versions of that module have been epically broken, especially when running on Server Core. Microsoft resolved the problems, but this highlights the **absolute necessity** of checking PowerShell Gallery for the latest bits, **before** you try and do anything. And, if you run into problems, hop into that module's repo on GitHub.com/powershell and see if anyone's reported a relevant bug against that version. If not… file a bug yourself.

[5] https://blogs.msdn.microsoft.com/powershell/2018/04/19/windows-pull-server-planning-update-april-2018/

Step 1: Install the Module

Run this:

```
1 Install-Module xPSDesiredStateConfiguration
```

Or, if it's already installed, make sure it's updated (Update-Module).

Step 2: Get an SSL Certificate

There is no legitimate reason, even in a lab environment, to run a non-secure pull server. None. "But, it's only a lab!" Yeah, but the point of a lab is to set up *what you plan to deploy in production*. So use SSL. Even if you're just "playing around," you should be playing around with it the right way. You don't drive without a seatbelt and you don't run pull servers without SSL. Seriously. Unless your *specific goal* is to get your company on CNN when you get hacked, then you don't run a non-SSL pull server. If you *do* run a non-SSL pull server, I kind of hope you go to jail. I really do.

A self-signed certificate will not work. You need an SSL certificate issued from a Certificate Authority that all of your nodes will trust. That means either a commercial CA, or more likely an internal CA that your computers are all already configured to trust. Don't have an internal CA? Time to stop operating like its 1999. Get with the program and set up a root CA. It's fun and easy.

Install the certificate to the local *machine* store, not your *user* store. That means you should be able to find the certificate by looking in CERT:/LocalMachine/My in PowerShell. Once you verify that the certificate is installed, copy its thumbprint (a GUID) to the clipboard.

Step 3: Make a GUID

Run [guid]::newGuid() to generate a new GUID. This is going to be the registration key for the pull server. In the chapter on configuring the LCM, I cover where you'll use this GUID again, so that your nodes can "log in" to the pull server.

Step 4: Set Up DSC

I'm starting by assuming that your pull server computer is running WMF v5 or later already. I'm going to build a configuration that turns on the pull server functionality, and I assume that SERVER1 is the name of the new pull-server-to-be.

```
configuration GoGoDscPullServer
{
    param
    (
            [string[]]$NodeName = 'SERVER1',

            [ValidateNotNullOrEmpty()]
            [string] $certificateThumbPrint,

            [Parameter(Mandatory)]
            [ValidateNotNullOrEmpty()]
            [string] $RegistrationKey
     )


     Import-DSCResource -ModuleName xPSDesiredStateConfiguration

     Node $NodeName
     {
         WindowsFeature DSCServiceFeature
         {
             Ensure = 'Present'
             Name   = 'DSC-Service'
         }

         xDscWebService PSDSCPullServer
         {
             Ensure                  = 'Present'
             EndpointName            = 'PSDSCPullServer'
             Port                    = 8080
             PhysicalPath            = "$env:SystemDrive\inetpub\PSDSCPullServer"
             CertificateThumbPrint   = $certificateThumbPrint
             ModulePath              = "$env:PROGRAMFILES\WindowsPowerShell\DscServi\
ce\Modules"
             ConfigurationPath       = "$env:PROGRAMFILES\WindowsPowerShell\DscServi\
ce\Configuration"
             State                   = 'Started'
             DependsOn               = '[WindowsFeature]DSCServiceFeature'
             UseSecurityBestPractices = $True
         }

        File RegistrationKeyFile
        {
```

```
            Ensure          = 'Present'
            Type            = 'File'
            DestinationPath = "$env:ProgramFiles\WindowsPowerShell\DscService\Regist\
rationKeys.txt"
            Contents        = $RegistrationKey
        }
    }
}

GoGoDscPullServer -cert paste_ssl_thumbprint_here -reg paste_reg_key_here
```

At the very bottom is the line of code that runs the configuration. You'll notice I'm passing in two parameters: the certificate thumbprint of the SSL certificate, and the registration key GUID you generated earlier. Let's walk through some of what's hardcoded in the configuration.

SPECIAL NOTE: There can be times when it's helpful to have the pull server running in HTTP, and that's when you're trying to troubleshoot it and need to do a packet capture using a tool like Wireshark. Specify a CertificateThumbprint of "AllowUnencrypted" to do that. DO NOT do that anywhere but in a walled-off lab environment, please! And, don't forget that the LCM on your nodes must be configured to allow unsecure communications, and you'll need to change the download manager URLs in the LCM to use http:// instead of https://. This is how I created some of the packet captures in the Troubleshooting chapter.

To begin with, there are several paths:

```
PhysicalPath = "$env:SystemDrive\inetpub\PSDSCPullServer"
ModulePath = "$env:PROGRAMFILES\WindowsPowerShell\DscService\Modules"
ConfigurationPath = "$env:PROGRAMFILES\WindowsPowerShell\DscService\Configuration"
```

PhysicalPath is where the pull server endpoint will be installed. ModulePath is where you'll drop your ZIPped-up DSC resource modules so that nodes can download them. ConfigurationPath is where you'll drop all your MOF files.

You can also specify the port on which the pull server will run. Notice that 8080 is nonstandard (it isn't 80 or 443); you can use whatever you like.

```
Port = 8080
```

You also can specify the location of the Registration Keys file. You'll notice in the example that it's being pre-populated with the Registration Key you generated previously; the file can actually contain as many keys as you want.

```
DestinationPath = "$env:ProgramFiles\WindowsPowerShell\DscService\RegistrationKeys.t\
xt"
```

There's some argument in the community about how to handle these. Some folks are comfortable creating one key and letting all their nodes use it. Others want to generate a unique key for each node. I think the latter is a lot of work and maintenance waiting to happen, and in a large environment I have some concerns about the overhead of a large key file. It's not presently clear what the potential risks could be in using one, or a small number of, registration keys.

There's another additional required setting for the pull server configuration: **UseSecurityBestPractices**. This is set to either $true or $false. This setting, when set to $True, will disable all insecure protocols (at this time, all SSL protocols and TLS 1.0). We've seen some people have problems with the pull server when configured this way (specifically, some nodes not being able to register), requiring them also to add **DisableSecurityBestPractices='SecureTLSProtocols'** to disable TLS (which does not disable SSL). Note that these may not affect an *existing* pull server; if you're having problems and need to reconfigure, you may need to remove an existing pull server entirely and rebuild it. Or, go into the registry key affected by the setting and remove any keys containing "TLS" in them. Essentially, UseSecurityBestPractices sets that registry key, HKLM:\SYSTEM\CurrentControlSet\Control\SecurityProviders\SCHANNEL, to enforce stronger encryption - but it can negatively affect older clients that don't support those. https://support.microsoft.com/en-us/kb/245030 and

https://technet.microsoft.com/en-us/library/dn786418(v=ws.11).aspx contain more detail.

Step 5: Run and Deploy the Config

Running the above script (assuming you've adjusted any values you want to) will produce ./GoGoDscPullServer/SERVER1.mof. You can now push that to your prospective pull server:

```
Start-DscConfiguration -Path ./GoGoDscPullServer -ComputerName SERVER1
```

This'll take a while to run; you may want to add -Verbose and -Wait to the command to watch it all happen. If IIS isn't installed on the server, it'll get installed (as a pre-requisite of the DSC pull server feature), and that's what can take several minutes.

Confirming the Setup

When everything's done, you should be able to view the PhysicalPath on the pull server to confirm that the DSC service endpoint (a .svc file) exists, and you can remotely use IIS Manager to attach to the server, confirm that IIS is running, and confirm that the website details (port, certificate, and so on) are correct.

And I'm deadly serious about the SSL thing. There's an upcoming chapter on DSC security where I'll dig into more specifics on why SSL is so important, and even make a case for client certificate authentication on the pull server.

Life Choices

By default, Pull Server (in v5 and later) uses an EDB database. That's the database architecture used by Exchange Server and Active Directory, and it was the only option supported on early versions of Nano Server (when we all thought Nano Server would be a standalone install option one day). EDB databases *require maintenance*, meaning that if you don't clear the logs or set them to auto-cycle, it'll fill up the disk. Yikes. Not a wholly popular decision.

It can also be configured to use the "other" JET database format, MDB, which is what you know and love from Microsoft Access. The less said about that the better, although at least it doesn't fill up your disk.

More recently (April 2018, in Windows Server build 17090), they added the ability to instead use a SQL Server, which is a great option that we recommend heartily. You can use SQL Express, or an existing SQL Server installation on your network. When you create your xDscWebService configuration items, add two keys:

```
1 SqlProvider = $true
2 SqlConnectionString = "Provider=SQLNCLI11;Data Source=(local)\SQLExpress;User ID=SA;\
3 Password=Password12!;Initial Catalog=master;"
```

That connection string obviously needs to be valid in your environment; this example is using a local SQL Express instance, and logging in with the horrible, you-should-never-do-this "SA" account. We strongly recommend creating a dedicated account for the pull server to log into SQL Server with or, best yet, getting Windows Integrated authentication to work. But that's a SQL Server issue, not a PowerShell one, so we'll leave it to you and your DBA.

Opting Out of the Pull Server Approach

The way we've been going on, you may assume that a pull server is a mandatory part of the DSC infrastructure. It isn't. A pull server is useful when - and *only* when - you plan to create updated configuration MOFs from time to time, especially if that's frequently, and you want nodes to grab them automagically and implement them.

We tend to assume that this is the "best" way because we use Group Policy that way. But for server workloads in particular, you may decide *you don't even want a pull server at all*. For example, DSC currently doesn't support anything like maintenance windows or scheduled downtime, so constantly pulling and implementing configurations *might* be disruptive in your environment.

Instead, you could choose simply to *push* new configurations whenever you want. And this isn't necessarily a manual procedure, either - keep in mind, DSC is a *platform*, not a *solution* that includes ready-made tools. We're aware of at least one organization that has constructed its own (sadly proprietary) "push server." That server is configured with maintenance window information and a database of configurations. When a configuration changes, the server waits until a window is open, and then *pushes* the new configuration to the node. It has a lot of error handling and alerting so that if a push attempt fails, everyone's informed.

The point of this short chapter is to emphasize that *DSC is what you need it to be*, provided you're willing (A) to be creative, and (B) to implement your own solutions on top of DSC, as needed.

Part 3: Testing the Infrastructure

In this Part, we'll walk through a couple of very basic scenarios. We'll assume that you have at least a client computer and a server, and that they're both running PowerShell v5. We'll set up a very basic configuration MOF and push it to the server. In doing so, that'll create a very basic (and not secure - so do this in an isolated lab) pull server, which we'll then use to test *that* functionality quickly.

It's worth noting that we assume you have a Windows 10 client, and a Windows Server 2016 server, available. This should also work with Windows Server 2012 R2, and with older clients running PowerShell v5 or later, but we haven't specifically tested it. We do *not* cover setting up a complete lab environment; if you're playing with DSC, you should already be comfortable with that.

Testing Push Mode

As we described in the Part opener, in this chapter we will show you how to:

- Create a very basic configuration MOF that sets up a very basic, not-secure Pull server (securing it will require certificates, which we haven't gotten to, yet), and
- Push the configuration to a server

Whether you follow along on your own computer or not is entirely optional.

Creating the Configuration

First, in order to set up a pull server, you need to have the xPSDesiredStateConfiguration module installed from the Powershell Gallery. Locate the module with command:

```
Find-module xPSDesiredStateConfiguration
```

Note that you may be prompted to download a NuGet provider.

Ensure that the module is found, and that the current version is the desired version (e.g., the latest). You could specify a specific version using the `-RequiredVersion` parameter, or list all versions using the `-AllVersions` parameter before choosing a version. This example uses the current version as of this writing (5.1.0.0). Once you find the module, install it:

```
Install-module xPSDesiredStateConfiguration
```

The module will be placed in %systemdrive%Program FilesWindowsPowershellModules, if you want to verify its existence. Once the module is installed, you can use it in the configuration. After the `Configuration` keyword, insert the following line:

```
Import-DscResource -ModuleName xPSDesiredStateConfiguration -ModuleVersion `
  5.1.0.0
```

Second, this simple example is setting up a non-secure - that is, HTTP instead of HTTPS - pull server. This is not a recommended configuration for any production infrastructure setup - or for any lab that tests anything that might one day go into production. This configuration is provided solely as an example for learning purposes. Remember, *you do not want your compromised pull server to be the reason your company ends up on the evening news.* If you decide to follow along with this example, do so *in an isolated, disconnected lab environment.* Seriously, OK?

The configuration to set up a simple pull server is (and please, remember our earlier note about wacky code formatting - we've included this in the GitHub sample code repo for your convenience):

```
Configuration PullServer {

Import-DscResource -ModuleName xPSDesiredStateConfiguration `
    -ModuleVersion 5.1.0.0

  Node Pull {

    WindowsFeature DSCService {
      Name = "DSC-Service"
      Ensure = 'Present'
      }

    xDSCWebService Pullserver {
      Ensure = 'Present'
      EndpointName = 'PSDSCPullServer'
      Port = '8080'
      PhysicalPath = "$env:SystemDrive\inetpub\wwwroot\PSDSCPullServer"
      ModulePath = "$env:PROGRAMFILES\WindowsPowerShell\DscService\Modules"
      ConfigurationPath = `
        "$env:PROGRAMFILES\WindowsPowerShell\DscService\Configuration"
      State = "Started"
      DependsOn = "[WindowsFeature]DSCService"
      UseSecurityBestPractices = $false
      CertificateThumbprint = "AllowUnencryptedTraffic"
      }

    File RegistrationKey {
      Ensure = 'Present'
      DestinationPath = `
        "$env:PROGRAMFILES\WindowsPowershell\DscService\registrationKeys.txt"
      Contents = '0f9ae841-785d-4a2d-8cdf-ecae01f44cdb'
      Type = 'File'
           }
  }
}
PullServer
```

In this example, the configuration name is `PullServer`, and the node's name is `Pull` (obviously, you can't run this as-is unless that's how you've named the server upon which you plan to inflict this configuration). It will install the DSC-Service Windows Feature, which includes IIS and other bits required for the Pull Server to work. After that, it uses the xDSCWebService resource, which is located in the xPSDesiredStateConfiguration resource that we downloaded, to configure the pull server endpoint. It will also create a file, registrationKeys.txt. This file can contain a single GUID or

multiple GUIDs. The GUIDs are the **shared secrets** that will allow nodes to register with the pull server.

A couple of additonal notes about this configuration:

- The `CertificateThumbprint` setting is set to "AllowUnencryptedTraffic" so that HTTP can be used instead of HTTPS, but to use HTTPS, the thumbprint of the SSL certificate would be provided as the value instead. The certificate would need to be pre-installed in the node's Machine store by other means - this neither generates, nor installs, a certificate.
- The `UseSecurityBestPractices` setting is set to False, because you cannot use security best practices with unencrypted traffic.
- The `ConfigurationPath` setting is the location where the MOFs will be stored for download on the pull server. You use a local folder path that exists (or can be created) on the server.
- The `ModulePath` setting is the location from which to pull the modules that a given configuration needs.

Running the Configuration to Produce a MOF

Compile the configuration by loading the configuration into memory, and then executing it. The configuration block is similar to a function, where a function is loaded and then the name of the function is invoked to call the function.

In the example, the configuration portion starts at the configuration keyword and ends at the final curly brace. The last line (`PullServer`) invokes the configuration to create the MOF. Compiling this configuration results in a `Pull.MOF` file, located in the `PullServer` subdirectory of the current working directory. Notice that the configuration name is taken for the subdirectory name, and the node name is taken for the MOF filename.

Pushing the MOF

Assuming that the server on which you authored the configuration is also the pull server, deploy this configuration by running:

```
1 Start-DscConfiguration -Path .\PullServer -Verbose -Wait
```

If your authoring machine is separate from your pull server, you can deploy the configuration from the authoring machine to the pull server remotely by adding the `-ComputerName Pull` parameter to the above command.

This will configure IIS, firewall rules, and the PSDSCPullServer endpoint. Once the configuration has completed successfully, you can verify it from the client computer - you don't even need to log on to the server! Assuming the client computer has basic network connectivity to the pull server (and again assuming you named it "pull" like we did, and that your network knows how to resolve "pull" to an IP address), launch a browser and navigate to the pull server URL:

```
http://pull:8080/PSDSCPullServer.svc
```

If the endpoint is reachable and functioning, you will receive an XML document in response.

Testing Pull Mode

This chapter picks up from the previous one. Here, we will:

- Create a basic configuration MOF and deploy it to a Pull server
- Configure a node LCM to pull the configuration
- Verify that the node pulled and applied the configuration

This section assumes that you are authoring your configurations from a machine that is neither the pull server nor the target node.

Creating the Configuration

This example will set the timezone on a Windows 10 client computer named Cli1. It will use the xTimezone module from the Powershell Gallery. We'll need to download it first.

```
find-module xTimezone | install-module
```

Once the module is installed, you can see what resources it contains - and the syntax for each - by using get-DSCResource.

```
get-DSCResource -module xTimezone -syntax
```

The output of this command shows that there is one resource in this module (xTimeZone), and it also shows you the valid properties for this resource and the syntax for using the resource in the configuration.

```
xTimeZone [String] #ResourceName
{
    IsSingleInstance = [string]{ Yes }
    TimeZone = [string]
    [DependsOn = [string[]]]
    [PsDscRunAsCredential = [PSCredential]]
}
```

You may need to do additional investigation in the resource's documentation to find out exactly what needs to be set. For example:

- What does “IsSingleInstance” do?
- What string value does TimeZone accept?

To obtain further information, check out the module itself, typically located in %systemdrive%Program FilesWindowsPowershellModules. In the xTimeZone\1.6.0.0 folder, there is a readme.md file and upon reading it, you discover that not only is there information about how to find the valid strings for timezone values, there is also a sample configuration included. Not all downloaded resources are this helpful, so your mileage may vary.

Continuing with the example, Paris seems like a great vacation destination, so we’re going to configure the client computer to be in the timezone for Paris (Central European Time). Using the command provided in the readme, we find a valid string (Central European Standard Time) to use in the configuration.

```
Configuration ParisTZ {

Import-DscResource -ModuleName "xTimeZone" -ModuleVersion "1.6.0.0"

  Node TimeZoneConfig {

    xTimeZone Paris {
      IsSingleInstance = 'Yes'
      TimeZone = 'Central European Standard Time'
      }
  }
}

ParisTZ
```

Notice that the node definition uses “TimeZoneConfig” as the node “name”, and not the actual node name CLI1. This is because you will use configuration names on the LCM definition, and the configuration name for this config will be “TimeZoneConfig”.

Running the Configuration to Produce a MOF

Similar to running the configuration in the previous chapter, the configuration is compiled by loading the configuration into memory, and then executing it. Executing this code produces a MOF named TimeZoneConfig.mof in the ParisTZ subdirectory of the current directory.

Deploying the MOF and Module to a Pull Server

Next, we create a checksum for the MOF file in the same folder as the MOF. This creates the TimeZoneConfig.Mof.Checksum file.

```
new-dsccheckSum -path ./ParisTZ/TimeZoneConfig.mof
```

If we refer to the Pull Server configuration for a moment, we note that we specified the Configuration Path (where the configurations go) and Module Path (where the modules go) in the pull server configuration:

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

You will need these paths to know where to deploy the MOFs and modules on the pull server. You'll also need the proper rights on the pull server in order to copy the files to it. For this example, because we're using the administrative share (C$) to connect, that means you need to have administrative rights. **This is not a best practice, and is only for use in this isolated lab. Configurations provide a blueprint for your entire environment, so access to the pull server and its configurations and modules should be locked down properly using a least-privilege security model.**

Copy the MOF/checkum files to the Configuration folder.

```
copy-item -Path .\parisTZ\* -Destination `
  "\\pull\C`$\Program Files\WindowsPowerShell\DscService\Configuration" `
  -recurse -Force
```

Then zip up the xTimeZone module and copy it to the pull server (all using the compress-archive command below) and create a checksum file for the zipped module. The naming convention for the zipped module on the pull server MUST be `ModuleName_VersionNumber.zip`.

```
Compress-Archive -Path `
  "$Env:ProgramFiles\WindowsPowershell\Modules\xTimezone\1.6.0.0\*" `
  -DestinationPath `
  "\\pull\C$\program `
  Files\WindowsPowershell\DSCService\Modules\xTimeZone_1.6.0.0.zip" `
  -force

New-DscChecksum -Path `
"\\pull\C$\program `
  Files\WindowsPowershell\DSCService\Modules\xTimeZone_1.6.0.0.zip"
```

Creating a Meta-Configuration

In order to configure the client's LCM to get its configuration from a pull server, we'll revisit a few settings from the *Configuring the LCM* chapter.

```
[DSCLocalConfigurationManager()]

Configuration LCM_Pull {

    Node CLI1 {

        Settings {
            ConfigurationMode = 'ApplyAndAutoCorrect'
            RefreshMode = 'Pull'
            }

        ConfigurationRepositoryWeb PullServer {
            ServerURL = 'http://pull:8080/PsDscPullserver.svc'
            AllowUnsecureConnection = $True
            RegistrationKey = '0f9ae841-785d-4a2d-8cdf-ecae01f44cdb'
            ConfigurationNames = @('TimeZoneConfig')
            }

        ResourceRepositoryWeb PullServerModules {
            ServerURL = 'http://pull:8080/PsDscPullserver.svc'
            AllowUnsecureConnection = $True
            RegistrationKey = '0f9ae841-785d-4a2d-8cdf-ecae01f44cdb'
            }
    }
}

LCM_Pull
```

We're using the `AllowUnsecureConnection = $True` setting since the pull server is only configured for HTTP. The `RegistrationKey` setting must match a registration key that exists in the RegistrationKeys.txt file on the pull server. This is the same GUID that we used when creating the pull server. This setting is checked at registration time to ensure that the node is authorized to connect to the pull server, and later on subsequent communication with the pull server to retrieve configurations and modules.

As with all previous examples, execute this configuration in order to create Cli1.meta.mof in the LCM_Pull subdirectory.

Pushing the Meta-Configuration to a Node

To push the LCM configuration to Cli1:

```
Set-DscLocalConfigurationManager -ComputerName Cli1 -Path .\LCM_Pull `
  -Verbose -Force
```

This will register the LCM with the pull server!

Pulling the Configuration from the Pull Server

The client is now ready to receive the configuration from the pull server. We could do nothing, and at the next LCM consistency check, the LCM would pull down and apply the new configuration. But that's not really any **fun**. You want to see the configuration get applied, so you can issue the following command to apply the configuration right now:

```
Update-DscConfiguration -ComputerName cli1 -Verbose -Wait
```

This will apply the configuration, and if all the above instructions were completed successfully, the configuration will deploy successfuly, and your client node will be in the Central European Standard Time timezone.

Verifying the Node's State

To check the node's status:

```
get-dscconfigurationstatus -cimsession Cli1
```

This command will show the current status of the configuration (success if it was successful), along with the `Refresh Mode` (Pull or Push) and type of configuration update (Initial, Consistency, LocalConfigurationManager).

Part 4: Authoring Configurations

Configurations wind up being the biggest unit of work in DSC, meaning they're where you'll spend most of your time. We'll be spending a lot of time, in this book, covering them and the various ways in which you can author and deliver them. Let's get started.

Basic Configuration Authoring

In this chapter, we'll start digging into configuration authoring. There'll be a more advanced chapter on this later in the book, so we'll stick with the basics for now, and then build on those as we go.

If you're looking to try some of this stuff on your own, it's not hard to set up a lab environment. I'm using a Windows 10 virtual machine, running the latest version of Windows PowerShell (and I can't stress enough how important it is to make sure you've updated to the *latest and greatest*). I'm also running a Windows Server 2012 R2 server, also with the latest version of PowerShell. At the time of this chapter's writing, that's PowerShell v5.1 (which you can check by running $PSVersionTable in a console window). Neither machine belongs to a domain, and for this chapter I'm not even going to be using the server. I'll author, compile, and run my configurations entirely locally on the Win10 computer.

Getting Started: The Configuration Block

This is the simplest part of your configuration script:

```
configuration MyConfig {
}
```

While it's technically possible to include multiple configuration blocks in a single .ps1 file, I avoid doing so. For one, Azure DSC doesn't like it. For another, I like to have my script filenames match the configuration names - so the above would be in MyConfig.ps1.

Adding Nodes

The next part of the configuration identifies the node or nodes that you plan to create MOFs for. At the simplest level, it can look like this:

```
configuration MyConfig {
        node SERVER2 {
        }
}
```

When I run MyConfig, I'll get SERVER2.mof as the output. Of course, this isn't the most efficient way to create configurations, because you'll often have multiple computers that need the same configuration settings.

Adding a Parameter Block

One way you can start to make your configurations more dynamic is to add input parameters:

```
configuration MyConfig {
        param(
                [string[]]$ComputerName
        )

        node $ComputerName {
        }
}
```

Now, I could run something like **MyConfig -ComputerName SERVER1,SERVER2,SERVER3** and get three MOFs.

Adding Settings

Within the Node section, you identify the configuration settings that you want to go into the MOF. Let's start with something really simple:

```
configuration MyConfig {
        param(
                [string[]]$ComputerName
        )

        node $ComputerName {

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

        }
}
```

Let's look at what that's doing:

- I've specified a configuration setting that I've named *Backups*. This is an arbitrary name, meaning it's only going to have meaning to me. I could have named it Fred, or Rock, or whatever.

- The setting uses the *WindowsFeature* DSC resource. This happens to be a native resource on Windows Server computers that have WMF 4 or later installed, so I won't need to worry about deploying the resource.
- Within the setting, I've specified two *properties*. The properties of a setting will vary based on the DSC resource it uses. This particular DSC resource offers Ensure and Name as properties; other resources will be different.
- I've specified values - Present and Windows-Backup - for my properties. Again, the valid values will vary depending on the DSC resource that you're using. I know that, for the WindowsFeature resource, Ensure will either be *Absent* or *Present*, and I know that Name wants the feature name as shown by running Get-WindowsFeature. The chapter on "Finding and Using Resources" will explain how I found that out.

At the most basic level, this is all a configuration is: a bunch of settings, in a Node block, in a Configuration block.

Adding Basic Logic

Configurations can use PowerShell's entire scripting language to make decisions. For example:

```
configuration MyConfig {
        param(
                [string[]]$ComputerName
        )

        node $ComputerName {

                $features = @('windows-backup','dns-server','dhcp-server')
                ForEach ($feature in $features) {
                        WindowsFeature $feature {
                                Ensure = 'Present'
                                Name = $feature
                        }
                }
        }
}
```

When this is run, it'll produce a MOF with three Windows features. Notice that I've used $feature for the name of the *setting* as well as using it to identify the *feature* name. That's because *the name of every setting must be unique within the configuration*. This probably isn't an incredibly practical example, but it does show you how you can use logic in a configuration.

But here's the important bit: This logic only executes when the configuration is run on your authoring machine. The final MOF won't contain the ForEach loop; the MOF will contain three

static settings. MOFs aren't scripts - they don't contain logic. MOFs don't ask questions and make decisions, for the most part; they just do what they contain.

Adding Node-Side Logic

There's one exception to the "MOFs don't contain logic" rule, and it's the special, built-in Script resource.

```
configuration MyConfig {
        param(
                [string[]]$ComputerName
        )

        node $ComputerName {

                Script MyScript {
                        GetScript  {}
                        TestScript {}
                        SetScript  {}
                }

        }
}
```

Within the TestScript script block, you can put whatever you want, provided it returns $True or $False as its only output. If it returns $False, then the target node's LCM will run whatever's in the SetScript script block. The GetScript script block should return a hash table that describes the current configuration. GetScript can either return an empty hashtable, or a hashtable with a single key-value pair. The key must be called "Result", and the value must be a string that represents the state of the configuration.

The **fun thing here** is that these script blocks don't run when you run the configuration and create MOFs. Instead, the PowerShell code in those three script blocks goes right into the MOF as-is. When the MOF is applied to a target node, it's the *LCM on the target node* that will run these. So this is a sort of way to make "dynamic" MOFs. Here's a quick example:

```
configuration MyConfig {
        param(
                [string[]]$ComputerName
        )

        node $ComputerName {

                Script MyScript {
                        GetScript  { write @{} }
                        TestScript { Test-Path 'c:\Program Files\Example Folder' }
                        SetScript  { Mkdir 'c:\Program Files\Example Folder' }
                }

        }
}
```

My GetScript returns an empty hash table; that's not uncommon. My TestScript will return $True if a folder exists, and $False if it doesn't (because that's what Test-Path itself returns). If the folder doesn't exist, my SetScript block creates the folder. Again, a very simple example, but shows you the idea.

However…

You need to be a little careful with Script resources. They can be harder to maintain and, if they're complex, really difficult to troubleshoot. For all but the simplest purposes, I usually prefer creating my own custom DSC resource. Honestly, doing so requires like four extra lines of code on top of just writing a Script resource, and you end up with something that's easier to maintain and debug. We'll obviously be covering that in an upcoming chapter.

Documenting Dependencies

I've touched on this before, but you can help the LCM understand the order in which settings must be applied by using the DependsOn property. This is a common property, available for all DSC resources.

```
configuration MyConfig {
        param(
                [string[]]$ComputerName
        )

        node $ComputerName {

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

                WindowsFeature DHCP {
                        Ensure = 'Present'
                        Name = 'DHCP-Server'
                }

                CustomDHCPLoader LoadDHCP {
                        SourceLocation = \\companyroot\admin\dhcp\sources.xml
                        DependsOn = [WindowsFeature]DHCP,[WindowsFeature]Backups
                }

        }
}
```

In this example, the CustomDHCPLoader setting (which is something I made up; it's not native to Windows) won't run until after the two WindowsFeature settings are confirmed to be correct.

Running the Configuration

I've sort of already covered this, but in most cases you'll actually run the configuration right at the bottom of the script file that the configuration lives in:

```
MyConfig -ComputerName ONE,TWO,WHATEVER
```

But in addition to any parameters *you* define, you also get a handful of free parameters that you can use:

- **-InstanceName** isn't something you'll probably ever use. It lets you override the ID that DSC uses to manage composite configurations. and it's honestly best to not mess with it.

- **-OutputPath** accepts a folder path in which your MOF files will be created. By default, PowerShell will create a folder named after your configuration (/MyConfig, in the example I've used), and put the MOFs in there.
- **-ConfigurationData** accepts a hash table of configuration data, which you can use to drive logic decisions within the configuration. This is a big topic, and there's a whole chapter on it later.

Deploying the MOF

Once you've got the MOF authored, you're ready to deploy it. You might do that in Push mode by using `Start-DscConfiguration`, or in Pull mode, which we cover in the "Deploying MOFs to Pull Servers" chapter.

Wrapping Up

This should give you a basic idea of what configurations look like. You can do a *lot* more with them, and we'll have a whole chapter on more advanced options later in the book. For now, I just wanted to set the stage for this portion of DSC, so that we can cover some of the other major players.

Going Further with Configurations

To this point, we've basically treated configurations and nodes as a 1:1 thing. One configuration script gets you one MOF, which goes to one node. For some environments, that'll be totally fine. For others, you'll want a bit more modularization and re-use. After all, many nodes will be similar, so why copy-and-paste all that script code?

Again: DSC isn't Tooling

I feel it's important at this stage to remind you that DSC is a platform, not a solution set. That is, there's no tooling. *Ideally*, many of us want some kind of System Center-y thing, where we can assign "roles" to our nodes, and have some magical back-end database spew out the necessary MOF files directly to a pull server. That'd be great, and I bet we'll get it one day. But that isn't what this book is about, anyway. So we're going to stick with what's "in the box," and you'll likely end up using a mix-and-match of the techniques I'm covering to meet your needs.

Understanding ConfigurationData

One of the first ways that you can have a single configuration script produce multiple, unique MOFs, is to supply the script with *configuration data*. This is something that lives *apart from* the script itself, and is fed to the script when the script is run. So, just to set the stage broadly, a script file might look like this:

```
configuration MyMasterConfig {
  # a bunch of stuff goes here
}

$configData = @{}

MyMasterConfig -ConfigurationData $configData
```

I've created a configuration named **MyMasterConfig**. Separately - and this could have been in a totally different file that I dot-sourced, for example - I created a $configData hash table. I'm not showing you the structure of that yet, but I will in a bit. Then, I run the config, and pass in the hash table using a built-in parameter. Now, my configuration data is available "inside" MyMasterConfig.

Let's start digging into the actual details.

Defining Configuration Data

The top level configuration data hash table needs to contain two elements, AllNodes and NonNodeData. AllNodes is an array, and it'll contain all of the unique-to-each-node information that you need to provide. NonNodeData is a kind of "global" section, and we'll see later how it's used.

I mentioned that AllNodes is an array. Each element within the array is a new hash table, which must at least contain a NodeName element. Each can then contain whatever other elements you want, and you basically get to make them up - there are no requirements or specifics. So, as a starting point:

```
$MyData =
@{
    AllNodes =
    @(
        @{
            NodeName = 'NODE1'
            Role     = 'WebServer'
        },

        @{
            NodeName = 'SERVER2'
            Role     = 'WordPress'
        },

        @{
            NodeName = 'MACHINE3'
            Role     = 'WebServer'
        }
    );
    NonNodeData = ''
}
```

Here, I've defined three nodes, named NODE1, SERVER2, and MACHINE3. For each, I've defined a "Role" property. That's a property I made up; DSC doesn't care what other properties I provide. I've specified a value for each Role, and again, this is data that's only meaningful to me. This configuration data isn't "connected" to any configuration script at this point. In order to create that connection, I would run the configuration, and use its built-in -ConfigurationData parameter to pass in $MyData. You do not need to define -ConfigurationData in a Param() block; it's a magic parameter that is automatically available for all configuration scripts.

Referencing and Using Configuration Data

Okay, so you've created a data block, and you've passed it into a configuration. How do you use it? Inside your configuration - and remember, you've passed the Configuration Data *into* the configuration script by means of the -ConfigurationData parameter - you'll use three built-in variables:

- $AllNodes automatically gets you the AllNodes@() portion of your ConfigurationData block. In the next section, I'll share some of its super powers. One of those powers is the ability to filter down the nodes - for example, only grabbing nodes that have a certain property set to a certain value.
- Once you've filtered the $AllNodes collection, you can use $Node to refer to just the current node.
- $ConfigurationData represents the entire block of data.

So let's see a bit of that in action. Remember that, in a configuration script, you start out with a Node block. That's where you can use $AllNodes, and it does a pretty impressive bit of magic. Pretend that the above three-node data block has been defined in the same script as this example:

```
configuration AllMyServers {
  Node $AllNodes.Where({$_.Role -eq "WebServer"}).NodeName {
  }
}

AllMyServers -ConfigurationData $MyData
```

Microsoft uses this example a lot, but rarely explains the magic (there are also a lot of older examples that have the wrong syntax). The Node{} construct wants you to provide a list of node names. That's it - just computer names, and no other data. So we're taking $AllNodes, which contains everything. We're using its special Where method to grab only those nodes whose Role property is set to WebServer. Remember that *I made up* the Role property, right? So I really could have filtered on any property that I'd put into the data block. Having done that filtering, I'm left with two nodes. Each node has a NodeName and a Role property - but the Node{} block here in the script *only wants the names*. So that's why the ".NodeName" is there - it takes the filtered nodes and grabs just the *contents* of the NodeName property.

Now let's add a bit more:

```
configuration AllMyServers {
  import-DSCResource xWebAdministration

  Node $AllNodes.Where({$_.Role -eq "WebServer"}).NodeName {
    xWebSite TheWeb {
      Name = $Node.NodeName
      PhysicalPath = 'c:\inetpub\wwwroot'
      Ensure = 'Present'
    }
  }
}

AllMyServers -ConfigurationData $MyData
```

Within the Node{} construct, PowerShell will automatically run the contents once for each node you've given it. It's kind of like an invisible ForEach wrapped around the whole thing. Inside the Node{} construct, I can use $Node to refer to "the current node." So on the first pass, $Node will refer to NODE1. On the second pass, it'll refer to MACHINE3. Go back and look at my data block if you need to remind yourself why.

You can see that this is going to produce two MOF files, one named NODE1.mof and another named MACHINE3.mof, both (by default) in a folder named /AllMyServers. Each MOF will define a website, which will have the same name as the server itself. That's actually not realistic, so let's revisit my data block and add some stuff to it:

```
$MyData =
@{
    AllNodes =
    @(
        @{
            NodeName = 'NODE1'
            Role     = 'WebServer'
            Site     = 'CustomApp'
            SitePath = 'c:\inetpub\approot'
        },

        @{
            NodeName = 'SERVER2'
            Role     = 'WordPress'
        },

        @{
            NodeName = 'MACHINE3'
```

```
            Role     = 'WebServer'
            Site     = 'App2'
            SitePath = 'c:\inetpub\app2root'
        }
    );
    NonNodeData = ''
}
```

Notice that I've added Site and SitePath properties only to the nodes that have a WebServer value for Role. That's totally legal - you do not need to provide consistent properties for each node. You simply have to make sure you're providing whatever properties your configuration script is expecting. So now I can use those two new properties:

```
configuration AllMyServers {
  Node $AllNodes.Where({$_.Role -eq "WebServer"}).NodeName {
    xWebSite TheWeb {
      Name = $Node.Site
      PhysicalPath = $Node.SitePath
      Ensure = 'Present'
    }
  }
}

AllMyServers -ConfigurationData $MyData
```

You can see how the data block allows me to use a single configuration script to produce multiple, unique MOFs for multiple nodes.

All-Nodes Data

You can also specify properties to apply to all nodes:

```
$MyData =
@{
    AllNodes =
    @(
        @{  NodeName = '*'
            SystemRoot = 'c:\windows\system32'
         },

        @{
```

```
            NodeName = 'NODE1'
            Role     = 'WebServer'
            Site     = 'CustomApp'
            SitePath = 'c:\inetpub\approot'
        },

        @{
            NodeName = 'SERVER2'
            Role     = 'WordPress'
        },

        @{
            NodeName = 'MACHINE3'
            Role     = 'WebServer'
            Site     = 'App2'
            SitePath = 'c:\inetpub\app2root'
        }
    );
    NonNodeData = ''
}
```

Now, all three of my nodes also have a SystemRoot property. This is exactly the same as if I'd manually listed SystemRoot for each node individually, but it saves some room. The * is a special character in this case; don't think of it as a usual wildcard, because other wildcard expressions aren't allowed.

Using the $AllNodes Variable

The $AllNodes variable, then, becomes a key for making those MOFs unique. It has two special powers, one of which you've seen: Where and ForEach. You've seen how Where can be used to select just a subset of nodes. For example, using my most recent $MyData data block example, all of the following are legal:

```
$AllNodes.Where({ $_.Site -eq 'CustomApp' }) # returns 1 node
$AllNodes.Where({ $_.NodeName -like 'NODE*' }) # returns 1 node
$AllNodes.Where({ $_.SystemRoot -eq 'c:\windows\system32' }) # returns 3 nodes
```

I also mentioned the ForEach() method of $AllNodes. Frankly, I don't find myself using that for nodes, because the Node{} construct is already an implicit ForEach. However, there are certainly times when it can be useful. Imagine this configuration data block:

```
{
AllNodes = @(
@{
  NodeName = '*'
  FilesToCopy = @(
    @{
      SourcePath = 'C:\SampleConfig.xml'
      TargetPath = 'C:\SampleCode\SampleConfig.xml'
    },
    @{
      SourcePath = 'C:\SampleConfig2.xml'
      TargetPath = 'C:\SampleCode\SampleConfig2.xml'
    }
}
```

Now you might use a ForEach loop:

```
configuration CopyStuff {

  Node $AllNodes.NodeName {

    $filenumber = 1
    ForEach ($file in $Node.FilesToCopy) {
      File "File$filenumber" {
        Ensure = 'Present'
        Type = 'File'
        SourcePath = $file.SourcePath
        DestinationPath = $file.TargetPath
      }
      $filenumber++
    }

  }

}
```

Except I didn't actually use $AllNodes.ForEach(), right (grin)? That's just a plain ForEach loop, but it does show how it can sometimes be beneficial to do that. Honestly... I really do struggle to find use cases for $AllNodes.ForEach(). I've always found it simpler to filter using Where(), and to just use a normal ForEach loop - as above - when I need to enumerate something from the data block.

Configuration Script Strategies

Returning back to the AllMyServers configuration example from above, if you were really paying attention to all of the foregoing examples, you might have noticed that I only generated MOFs for NODE1 and MACHINE3. What about SERVER2?

It depends.

Option 1: Multiple Configuration Scripts

One option is to keep your configuration data block in a separate file, and then create multiple configuration scripts for different kinds of nodes. That same block can be fed to multiple configuration scripts. For example:

```
$MyData = @{
  $AllNodes = @(... data goes here ...)
  $NonNodeData = ''
}

. ./Config1.ps1 # Contains "MyFirstConfig" configuration
. ./Config2.ps1 # Contains "MySecondConfig" configuration

MyFirstConfig -ConfigurationData $MyData
MySecondConfig -ConfigurationData $MyData
```

So I've got two configurations, which I've put into separate script files for my convenience. I've dot-sourced those into the same script that contains my data block, and then passed the same data block to each. Presumably, each configuration is filtering the nodes in some fashion, so they're not necessarily duplicating each other's efforts. Obviously, I haven't completely filled in the $AllNodes section with actual node properties - I was just trying to lay out a concise framework to illustrate the approach.

Option 2: Multiple Node{} Blocks

Very similar to Option 1 is to simply include more than one node block in a configuration script.

```
configuration UberConfig {
  Node $AllNodes.Where({ $_.Role -eq 'WebServer' }).NodeName {
  }

  Node $AllNodes.Where({ $_.Role -eq 'WordPress' }).NodeName {
  }
}
```

That's totally legal, and running UberConfig with my $MyData block would produce three MOFs - one for NODE1, one for SERVER2, and one for MACHINE3.

Option 3: Lots of Logic

You can also take a more complex approach where you *don't* filter the nodes in the Node{} construct, and instead use a lot of logic constructs.

```
configuration LogicalConfig {
  Node $AllNodes.NodeName {

    if ($Node.Role -eq "WebServer") {
      # add settings here
    }

    if ($Node.Role -eq "WordPress") {
      # add settings here
    }

  }
}
```

This gives you a more declarative kind of control over what goes into each MOF, but it can obviously get harder to read and maintain if you go nuts with it.

Using NonNodeData

There's a couple of important things to know about this sometimes-misused section. First, you do not have to call the section "NonNodeData", and second, you can have multiple NonNodeData sections, which makes a NonNodeData section a good option for role-specific settings that do not pertain to all nodes. The snippet below shows two NonNodeData sections, one for domain controller settings (DCData) and the other for DHCP server settings (DHCPData).

```
$MyData = @{
  AllNodes = @( ... )

  DCData = @{
            DomainName = "Test.Pri"
            DomainDN = "DC=Test,DC=Pri"
            DCDatabasePath = "C:\NTDS"
            DCLogPath = "C:\NTDS"
            SysvolPath = "C:\Sysvol"
            }

  DHCPData = @{
            DHCPName = 'DHCP1'
            DHCPIPStartRange = '192.168.3.200'
            DHCPIPEndRange = '192.168.3.250'
            DHCPSubnetMask = '255.255.255.0'
            DHCPState = 'Active'
            DHCPAddressFamily = 'IPv4'
            DHCPLeaseDuration = '00:08:00'
            DHCPScopeID = '192.168.3.0'
            DHCPDnsServerIPAddress = '192.168.3.10'
            DHCPRouter = '192.168.3.1'
            }
}
```

Keep in mind that you have the special $ConfigurationData variable, and $ConfigurationData.<NonNodeData> will access the "global" NonNodeData keys. Previously, we'd just set NonNodeData to an empty string (""), but in fact it can be a hash table or multiple hash tables. So now, the configuration can use $ConfigurationData.DCData.DomainName to access the domain name property, or $DHCP-Data.DHCPDNSServerIPAddress to access the DHCP server's DNS IP address.

Poor Man's Configuration Modularization

Unless you're in the very smallest of environments, or only using DSC for a very small subset of nodes, you're going to end up with more than one configuration script. Most of the time, you'll probably use some gut-instinct measurement of the configuration's difference, or "delta," from other configurations to decide when to make a new one.

For example, suppose you have two kinds of application web server. For each kind, you have a half-dozen actual machines. The two are very similar: they have the same firewall settings, basic IIS install, and so on. The main difference is perhaps some application pool settings, and the source files they use. So let's say they're about 90% the same, with a 10% delta. You might decide, "hey, 10% ain't so much, I'll just make one configuration, and use some Configuration Data filtering logic stuff to handle that 10%."

The percentage of delta that falls within your "comfort level" for having one configuration, versus two different ones will be very much a personal thing. But at some point, the "delta" between the two configurations will be large enough that you'll want to make two separate configuration scripts - while still modularizing, as much as possible, the common elements of them.

This chapter will present the easiest approach, at least conceptually. The next two chapters, which cover composite and partial configurations, will present different approaches. All of these have their own pros and cons, and you'll simply have to decide which works best for *your* organization.

Dot Sourcing

Suppose you have a configuration that's something like the following. Notice that I'm not using real DSC resources here - this isn't meant to be an exhaustive syntactic example, but rather a structural one.

```
configuration MyConfig {

  Node $AllNodes.Where({ $_.ServerType -eq 'AppServer' }) {

    WindowsFeature IIS {
      Name = 'Web-Server'
      Ensure = 'Present'
    }

    SourceFiles AppFiles {
      Source = '\\master\distribution\appserver'
      Destination = 'c:\inetpub\approot'
      SyncMode = 'checksum'
    }

  }

  Node $AllNodes.Where({ $_.ServerType -eq 'SalesServer' }) {

    WindowsFeature IIS {
      Name = 'Web-Server'
      Ensure = 'Present'
    }

    SourceFiles AppFiles {
      Source = '\\master\distribution\salesserver'
      Destination = 'c:\inetpub\salesroot'
      SyncMode = 'checksum'
    }

  }

}
```

So I'm configuring two kinds of nodes. They both have a common element - the IIS bit - and an element that differs. Because this is such a simple example, the delta between the two is around 50% - meaning half the configuration is different. So I could modularize this a bit.

I'll start by creating a file called ./IISStuff.ps1. I'm going to put it in the same folder as the actual configuration script, and it will look like this:

```
    WindowsFeature IIS {
      Name = 'Web-Server'
      Ensure = 'Present'
    }
```

That's not a valid configuration script! The ISE won't even color-code it or error-check it correctly. It's just a *fragment*. Now, in my main file, which I'll call .AppServers.ps1, I'll do this:

```
configuration MyConfig {

  Node $AllNodes.Where({ $_.ServerType -eq 'AppServer' }) {

    . .\IISStuff.ps1

    SourceFiles AppFiles {
      Source = '\\master\distribution\appserver'
      Destination = 'c:\inetpub\approot'
      SyncMode = 'checksum'
    }

  }

  Node $AllNodes.Where({ $_.ServerType -eq 'SalesServer' }) {

   . .\IISStuff.ps1

    SourceFiles AppFiles {
      Source = '\\master\distribution\salesserver'
      Destination = 'c:\inetpub\salesroot'
      SyncMode = 'checksum'
    }

  }

}
```

I've taken a first step at "modularizing" the common code. I've moved it into an external file, which is then dot-sourced into the configurations. Now from here, you could take the approach much further. You could have an entirely different configuration script file for each node type, each using IISStuff.ps1 as needed. You could obviously dot-source *multiple* files containing commonly used fragments.

Approach Analysis

I'm presenting this approach *not* because I see it used very often or because I think it's a great idea. I'm presenting it *mainly* because it's conceptually easy for most PowerShell scripters to understand. But we need to do some analysis of its pros and cons.

- PRO: Unlike a Composite configuration, dot-sourcing doesn't require you to deploy anything to your nodes.
- PRO: Unlike a Partial configuration, you'll be producing a single MOF-per-node that's complete and ready to deploy; you'll reduce errors like duplicate keys.
- CON: You can't ever look at the entire configuration "in one place, in front of your face," because it's spread across multiple files. That's also true with Partial configurations and, to a point, with Composite configurations.
- CON: Dot-sourcing just feels a little… messy. It's not as structured as approaches like Composite configurations or, to a point, Partial configurations.
- CON: You can't test the individual fragments in a standalone fashion, so you're not *truly* modularizing, here. That's not the case with either Composite or Partial configurations, which *can* be tested independently. Sort of. You certainly *could* work up "test configurations" that dot-sourced and tested individual fragments, but it can get messy.

But with this dot-sourcing "modularization" in mind, we can start exploring the more "official" ways of modularizing DSC configurations.

Composite Configurations

Composite configurations are one way to modularize commonly used groups of configuration settings. You start by building one or more "sub-configurations." That isn't an official term, but rather one I made up and find useful. A sub-configuration contains that group of commonly used configuration settings. It can be parameterized, which means it can slightly alter its output based on the input values you feed it. You then save that sub-configuration in a special way *that makes it look like a DSC resource.*

Next, you write the actual *composite* configuration. This is just a normal configuration that uses both regular DSC resources, and the looks-like-a-resource sub-configurations.

The official term for sub-resources is a *composite resource*, because it can take multiple DSC resources and merge, or "composite", them together into a single looks-like-a-resource. Technically, I don't think Microsoft is using the term "composite configurations" anymore, and "composite resource" is more accurate. But "composite configuration" has been thrown around since DSC was invented, so I'm gonna run with it.

I'm going to run through an example that uses code from this link, and add my own explanations and ideas as I go.

https://msdn.microsoft.com/en-us/powershell/dsc/authoringresourcecomposite

Creating a Composite Resource

Start by writing a perfectly normal DSC resource. You can actually run this on its own, produce a MOF, and test it. Here's the Microsoft example:

```
Configuration xVirtualMachine
{
    param
    (
        # Name of VMs
        [Parameter(Mandatory)]
        [ValidateNotNullOrEmpty()]
        [String[]] $VMName,

        # Name of Switch to create
        [Parameter(Mandatory)]
        [ValidateNotNullOrEmpty()]
```

```
13          [String] $SwitchName,
14
15          # Type of Switch to create
16          [Parameter(Mandatory)]
17          [ValidateNotNullOrEmpty()]
18          [String] $SwitchType,
19
20          # Source Path for VHD
21          [Parameter(Mandatory)]
22          [ValidateNotNullOrEmpty()]
23          [String] $VHDParentPath,
24
25          # Destination path for diff VHD
26          [Parameter(Mandatory)]
27          [ValidateNotNullOrEmpty()]
28          [String] $VHDPath,
29
30          # Startup Memory for VM
31          [Parameter(Mandatory)]
32          [ValidateNotNullOrEmpty()]
33          [String] $VMStartupMemory,
34
35          # State of the VM
36          [Parameter(Mandatory)]
37          [ValidateNotNullOrEmpty()]
38          [String] $VMState
39      )
40
41      # Import the module that defines custom resources
42      Import-DscResource -Module xComputerManagement,xHyper-V
43
44      # Install the Hyper-V role
45      WindowsFeature HyperV
46      {
47          Ensure = "Present"
48          Name = "Hyper-V"
49      }
50
51      # Create the virtual switch
52      xVMSwitch $SwitchName
53      {
54          Ensure = "Present"
55          Name = $SwitchName
```

```
56          Type = $SwitchType
57          DependsOn = "[WindowsFeature]HyperV"
58      }
59
60      # Check for Parent VHD file
61      File ParentVHDFile
62      {
63          Ensure = "Present"
64          DestinationPath = $VHDParentPath
65          Type = "File"
66          DependsOn = "[WindowsFeature]HyperV"
67      }
68
69      # Check the destination VHD folder
70      File VHDFolder
71      {
72          Ensure = "Present"
73          DestinationPath = $VHDPath
74          Type = "Directory"
75          DependsOn = "[File]ParentVHDFile"
76      }
77
78      # Create VM specific diff VHD
79      foreach ($Name in $VMName)
80      {
81          xVHD "VHD$Name"
82          {
83              Ensure = "Present"
84              Name = $Name
85              Path = $VHDPath
86              ParentPath = $VHDParentPath
87              DependsOn = @("[WindowsFeature]HyperV",
88                            "[File]VHDFolder")
89          }
90      }
91
92      # Create VM using the above VHD
93      foreach($Name in $VMName)
94      {
95          xVMHyperV "VMachine$Name"
96          {
97              Ensure = "Present"
98              Name = $Name
```

```
            VhDPath = (Join-Path -Path $VHDPath -ChildPath $Name)
            SwitchName = $SwitchName
            StartupMemory = $VMStartupMemory
            State = $VMState
            MACAddress = $MACAddress
            WaitForIP = $true
            DependsOn = @("[WindowsFeature]HyperV",
                          "[xVHD]VHD$Name")
        }
    }
}
```

Notice the huge Param() block right at the beginning. It wants the name of a virtual machine (VM), the name of a virtual switch, the switch type, a couple of paths, startup memory size, and the state of a VM. Basically, all the core things you need to define a new virtual machine. You'll notice those parameters - $SwitchName, $Name, and so on - being used throughout the rest of the configuration.

Turning the Configuration into a Resource Module

This gets saved into a special directory structure. It needs to start in one of the folders in the PSModulePath environment variable, such as /Program Files/WindowsPowerShell/modules. So, let's say we wanted this thing called MyVMMaker - that'll be the resource name we use to refer to this thing from within another configuration. We want the overall resource *module* to be called MyVMStuff. Remember, a *module* can contain multiple *resources*. I'll save:

/Program Files/WindowsPowerShell/modules/MyVMStuff/MyVMStuff.psd1

/Program Files/WindowsPowerShell/modules/MyVMStuff/DSCResources/MyVMMaker.psd1

/Program Files/WindowsPowerShell/modules/MyVMStuff/DSCResources/MyVMMaker.schema.psm1

So I need three files. MyVMMaker.schema.psm1 is the configuration that I wrote above. The other two files are *manifests*, and I need to create those. MyVMMaker.psd1 only needs one line:

```
RootModule = 'MyVMMMaker.schema.psm1'
```

To create the other .psd1, just run:

```
New-ModuleManifest `
    -Path \Program Files\WindowsPowerShell\modules\MyVMStuff\MyVMStuff.psd1
```

What's even easier is to grab the helper function from

https://blogs.technet.microsoft.com/ashleymcglone/2015/02/25/helper-function-to-create-a-powershell-dsc-composite-resource/, which includes a New-DSCCompositeResource function. It'll automatically set up the correct folder structure, .psd1 files, and template schema.psm1 file you can use as a starting point.

Using the Composite Resource

Now I can refer to the composite resource in a normal configuration:

```
configuration RenameVM
{

    Import-DscResource -Module MyVMStuff
    Node SERVER1
    {
        MyVMMaker NewVM
        {
            VMName = "Test"
            SwitchName = "Internal"
            SwitchType = "Internal"
            VhdParentPath = "C:\Demo\VHD\RTM.vhd"
            VHDPath = "C:\Demo\VHD"
            VMStartupMemory = 1024MB
            VMState = "Running"
        }
    }

}
```

You can see where I've explicitly imported the DSC resource *module* MyVMStuff, and referred to the MyVMMaker *resource*, passing in all the settings defined in the composite resource's Param() block.

Deploying the Composite Resource

To deploy the composite resource to a pull server, ZIP it up. Basically, in my example, you'd ZIP up the MyVMStuff folder under /Program Files/WindowsPowerShell/Modules, producing /Program Files/WindowsPowerShell/Modules/MyVMStuff.zip. Rename that to include a version number, like MyVMStuff_1.0.zip.

Then, create a checksum file by running:

```
New-DscCheckSum -Path MyVMStuff_1.0.zip -force.
```

Both the ZIP file and the checksum file would go on the pull server, in its designated Resources path. If you plan to manually deploy the resource, you don't need to do any of this - you can just get the correct folder hierarchy, and the files, in place in whatever fashion you desire.

Approach Analysis

As I wrote in the previous chapter, every modularization approach has pros and cons.

- CON: Like any other DSC resource you use that doesn't ship with the operating system, you have to deploy your composite resource. That can mean ZIPping it up, creating a checksum for it, and deploying it to a pull server. Or, it might mean manually deploying it to nodes, if you're not using a pull server.
- PRO: The "contents" of the composite resource are distinct from the configuration that calls it. So things like duplicate keys aren't a problem in the way that they can be for a Partial configuration.
- PRO: It's a little easier to see your entire configuration "in one place," because while the composite resource itself is separate, your "main" configuration *shows what the composite resource is doing*. That is, in my example, you can see that it's creating a VM, what the VM name is, and so on. Now, it's certainly possible to have a composite resource with a bunch of hardcoded information that won't be "seen" in the main configuration - but that's your decision to take that approach.
- PRO: Composite configurations are very structured.
- CON: There are some limitations on using DependsOn within composite resources. From the main configuration, any setting can have a DependsOn for any other setting *in the main configuration*. But you can't take a DependsOn to *an individual setting within the composite resource*. If you're designing composite resources well, you can minimize this as a problem. Simply keep each composite resource as "atomic" as possible, meaning whatever it's doing should be tightly coupled and not designed to be separated.
- PRO: It's relatively easy to test a composite resource, by simply building a "main" configuration that uses the composite resource and nothing else. You can also just "run" the composite resource standalone, because, after all, *it's just a configuration script*. Supply whatever parameters it wants, produce a MOF, and deploy that MOF to a test machine to validate it.

Design Considerations

As I noted above, the key to designing composite resources is to have them do one thing, and one thing only. Whatever's happening inside any given resource should all be intended to happen together, or not at all. For example, Microsoft's sample - which I used - creates a VM. You'd never create a VM without assigning memory, setting up a virtual disk, and so on, and so everything in that resource *goes together*.

You might have a set of security configurations - firewall stuff, maybe some anti-malware packages, and Windows Update settings - that you always want applied as a collection to your nodes. That's fine - that might make a good resource. What you want to avoid, however, are monolithic composite resources that perform several unrelated tasks, like making a machine a domain controller *and*

installing DHCP *and* installing something else. Those might all be common elements of your organization's "infrastructure servers," and each DC and DHCP server might be configured exactly the same, but here's how I'd handle it:

- A composite resource that sets up a company-standard DC
- A composite resource that sets up a company-standard DHCP server
- A "main" configuration that calls the DC and DHCP composite resources, and also sets up any node-unique information like computer name.

The idea is to keep each composite resource as tightly scoped as possible, and let the "main" configuration bring them together into a complete node config.

Partial Configurations

Honestly, we've already covered almost everything you need to know about partial configurations. In the chapter on Configuring the LCM, you saw how to tell the LCM to pull partials from a pull server, or to accept them as a push, or a mix of the two. You saw how to set up dependencies to control the order of partial configuration application. There's not a whole lot more to cover, really. The actual creation of a partial MOF is the same as creating any ordinary configuration; it's only a "partial" because a node is being told to use more than one of them. So, in this chapter, we'll kind of summarize the capabilities and cover the last couple of details.

The code samples in this chapter aren't meant to be real-world, runnable code. Partials can get a little complex, and so we've taken an oversimplified approach here so that we can get the point across.

Summarizing Partial Configuration Pros and Cons

We've covered this before, but to review briefly:

- Partials allow you to modularize chunks of your configuration, making it easier to have multiple contributors.
- Partials get combined and evaluated by the LCM, which means it's very difficult to detect potential conflicts ahead of time. This can make troubleshooting harder.
- Partials only work with v5 nodes, and if a Pull server is involved, it must also be v5.

We're presently in a kind of avoid-and-wait-and-see mode on partials. We'd prefer to have a configuration composed server-side and sent to the node, rather than the node having all the business logic needed to pull multiple items and combine them. Fellow MVP Steven Murawski wrote DSC Partial Configurations are the Devil's Workshop[6], which you might also find helpful.

Authoring a Partial Configuration MOF

Authoring a partial MOF is no big deal - it's exactly the same as authoring a "full" MOF. Period, full stop. Any legal MOF - including one generated by a configuration script - can be used as a partial. Nothing more to see here, move on.

[6] http://stevenmurawski.com/powershell/2016/03/dsc-partial-configurations-are-the-devils-workshop/

Configuring the LCM for Partial Configurations

We covered this in "Configuring the LCM," and aside from the Authoritative bit, below, there's nothing more to say here.

Partial Configuration Dependencies

As you saw in the chapter on Configuring the LCM, one partial can depend upon another. However, that's an all-or-nothing affair. One partial cannot depend on just some things within another partial.

Partial Configuration Authoritative Resources

When you configure the LCM to get a partial, you can tell it a list of resources *that only that particular MOF may contain*. So, you might say, "grab this partial, and it's the only one that can contain WindowsFeature settings. If any other partial tries to use the WindowsFeature resource, explode." It looks like this:

```
...
PartialConfiguration Bob {
        RefreshMode = "Pull"
        ConfigurationSource  = @("[ConfigurationRepositoryWeb]MyPullServer")
        Description = "Something"
        ExclusiveResources = @("Script","WindowsFeature")
}
```

This would allow only the Bob partial configuration to use the Script or WindowsFeature resources.

It's important not to treat this as a security thing. It's more of a "help our team not do the wrong thing, but don't stop us." Keep in mind that this "rejecting" of "unallowed" resources *happens at the LCM*, and the LCM's configuration can in fact be changed. Also keep in mind that, when you've got multiple people working on partial configurations *and not talking to each other,* you're basically just asking for trouble. Some kind of hard-to-troubleshoot conflict is eventually going to enter the situation. Partials shouldn't be used to rectify DSC's lack of native tooling, and this whole "authoritative resources" thing is essentially trying to do just that.

Also note that class-based resources can't be used as ExclusiveResources before WMF5.1.

Mix 'n' Match

It's possible to tell an LCM that some partials will be pulled, while others will be pushed. If you're going this route, you might do so because some configuration items (those being pushed) will be

changed only rarely, while others (being pulled) may be updated more often. There's a good look at how to configure the LCM this way on PowerShellMagazine.com[7].

File Naming Details

In WMF 5.0, let's say you configured the LCM as follows:

```
PartialConfiguration Bob {
        RefreshMode = "Pull"
        ConfigurationSource  = @("[ConfigurationRepositoryWeb]MyPullServer")
        Description = "Something"
        ExclusiveResources = @("Script","WindowsFeature")
}
```

In this case, the LCM will ask the pull server for a configuration named "Bob." The pull server - again, in WMF 5.0 - would look *on disk* for "Bob.mof" and "Bob.checksum.mof." The problem is that the pull server *in Azure Automation* stored files differently. It would use *ConfigurationName.NodeName.mof* instead. Chucking that nodename in there meant that the LCM couldn't request partial configurations from Azure Automation! So in WMF 5.1, the protocol was changed slightly, and *either* file naming convention is acceptable. Microsoft provides this example of an LCM configuration which uses both a local pull server and Azure Automation:

```
[DscLocalConfigurationManager()]
   Configuration RegistrationMetaConfig
   {
        Settings
        {
            RefreshFrequencyMins = 30
            RefreshMode = "PULL"
        }

        ConfigurationRepositoryWeb web
        {
            ServerURL =  $endPoint
            RegistrationKey = $registrationKey
            ConfigurationNames = $configurationName
        }

        # Partial configuration managed by Azure Automation service.
        PartialConfiguration PartialConfigurationManagedByAzureAutomation
```

[7] http://www.powershellmagazine.com/2015/02/20/mixed-refresh-mode-support-for-dsc-partial-configurations/

```
19          {
20              ConfigurationSource = "[ConfigurationRepositoryWeb]Web"
21          }
22
23          # This partial configuration is managed locally.
24          PartialConfiguration OnPremisesConfig
25          {
26              RefreshMode = "PUSH"
27              ExclusiveResources = @("Script")
28          }
29
30      }
```

Deploying MOFs to a Pull Server

Once you've created a MOF, you can obviously deploy it in Push mode by running `Start-DscConfiguration` - we've covered that. But you may also wish to deploy it to either an HTTP(S) or an SMB Pull server. That's straightforward: when you configured your pull server, you specified the folder in which configurations would live (see, "Setting Up a Pull Server" in this book). Simply copy the configuration MOF files to that location. You'll also need to create a checksum file for each MOF; the `New-DscChecksum` command can do that for you.

MOF naming depends on how you've configured your LCMs. If you're using ConfigurationIds (which is the the only option in v4), then each MOF needs to have the GUID - that is, the Configuration ID - that you've assigned the node (or nodes) that will pull the configuration. Checksum files have the same filename, with the addition of a ".mof.checksum" filename extension. However, if you're using Configuration Names in v5 or later, then MOF filenames take the configurations' names - like "Bob.mof," with "Bob.mof.checksum" being the checksum.

Part 5: Using and Authoring Resources

To this point, we've tried to stick with the simple, built-in resources that Microsoft provides. Eventually, you'll need to figure out what all of those are and how to use them, so we'll cover that next. And *eventually*, you'll probably run into some situation where the resource you need doesn't exist, so we'll also cover how to create your own.

Finding and Using Resources

At this point in the book, I feel that we've gotten all the basics out of the way. You should know who the major players are in the DSC process, and you should have all the core terminology in your head. So now we can get serious.

I think the first thing people tend to ask about DSC is, "What can I do with it?" Legit question. And although I wanted to make sure you knew about configurations, MOFs, the LCM, and all that, for me, "What can I do with it?" is the right starting point for DSC. Figuring out what DSC resources are available, and figuring out how to use them, is pretty much where DSC really begins.

One major upside to DSC is that so much is available as open source projects. Nearly everything Microsoft has created for DSC is open source. Tons of open source projects from community groups and enthusiasts are out there. And many of those projects have dozens (or more!) of contributors, all improving and debugging everything, all the time. Things move and improve fast - which is great. But the corresponding downside is that documentation can be slim or nonexistent. So figuring out how to use what's out there - that's critical.

Finding What's Out There

Let's be clear: there is no way I, or anyone, can document all the open source DSC resources out there. What I can do, however, is point you to some of the major locations where you'll find them.

First, use the PowerShell Gallery[8]. This is a huge repository of PowerShell modules, including DSC resources. These are usually the "production ready" modules that you want to use. It's meant to be accessed from within PowerShell itself, using PowerShellGet and PowerShell Package Manager. The site contains download links for PowerShell Package Manager in the event you don't have it, although it's a native part of WMF5, so you should. Using the Find-Module command within PowerShell, you can even limit your results to just DSC resources. There are actually a few ways to do that filtering, here's the one I use a lot:

[8] http://powershellgallery.com

```
PS C:\> find-module -tag dsc

Version    Name                                Repository           Description
-------    ----                                ----------           -----------
1.11.0.0   xWebAdministration                  PSGallery            Module with DSC \
Resources for Web Administration
3.10.0.0   xPSDesiredStateConfiguration        PSGallery            The xPSDesiredSt\
ateConfiguration module is a part...
2.9.0.0    xNetworking                         PSGallery            Module with DSC \
Resources for Networking area
2.11.0.0   xActiveDirectory                    PSGallery            The xActiveDirec\
tory module is originally part of...
1.9.0.0    xDSCResourceDesigner                PSGallery            This module is m\
eant to assist with the developme...
1.6.0.0    xComputerManagement                 PSGallery            The xComputerMan\
agement module is originally part...
1.6.0.0    xSQLServer                          PSGallery            Module with DSC \
Resources for dep
```

I've truncated those results, by the way - you can run the command yourself to see them all.

Note - You may still see online references to the "DSC Resource Kit." Some of these references may include downloads that existed before Microsoft moved to PowerShell Gallery; you should download resources from the Gallery and not from DSC Resource Kit download pages to ensure you're getting the latest and greatest resources.

Many of the DSC resource authors in the community publish their modules to PowerShell Gallery. If you happen to run across their source code repositories - often GitHub - you should use what's in Gallery rather than downloading the code directly. Often times, what's in Gallery represents a "stable" release, while what's in the source code repo may be a "development branch."

And aside from resources authored by Microsoft (the full output of Find-Module can tell you that), you should treat any code in Gallery as untrusted. That is, you should review it and be comfortable with it before deploying it in your production environment.

Not *everyone* publishes to the Gallery, though, and so Google remains one of your best friends when it comes to finding new DSC resources.

Installing What's Out There

OK: you've found the awesomeness you need online. How do you install it?

If the DSC resource module you're after happens to be in an online gallery - like PowerShell Gallery - simply use the Install-Module command to install it to your local computer. Note that this will *not* necessarily give you something you can immediately deploy to all the target nodes that will need it;

right now, we're just getting the module onto your authoring machine. We'll worry about deploying it to other nodes later.

If the module you need isn't in a Gallery that Install-Module can get to, then you're likely just downloading a ZIP file. That's fine. Unzip it to your /Program Files/WindowsPowerShell/Modules folder. Now, be a little careful, here. Some folks will ZIP up *that entire path*, meaning you'll end up with /Program Files/WindowsPowerShell/Modules/Program Files/WindowsPowerShell/-Modules/WhateverTheModuleNameIs. That won't work. Supposing the module is named Fred (for some reason), you ultimately want /Program Files/WindowsPowerShell/Modules/Fred. Inside the /Fred folder, all of the module's files should exist, and that may include one or more subfolders, too.

Finding What's Installed

Here's the test: making sure PowerShell can actually "see" the modules you've installed. This is super-important, because if this step fails, then nothing will work. So to test that, run Get-DscResource.

```
ImplementedAs   Name                    ModuleName                    Version    \
Properties
-------------   ----                    ----------                    -------    \
----------
Binary          File                                                           \
{Destina...
PowerShell      Archive                 PSDesiredStateConfiguration   1.1        \
{Destina...
PowerShell      Environment             PSDesiredStateConfiguration   1.1        \
{Name, D...
PowerShell      Group                   PSDesiredStateConfiguration   1.1        \
{GroupNa...
```

As you can see, I have several resources - Archive, Environment, and so on - in a *resource module* called PSDesiredStateConfiguration, and its version is 1.1. After running Get-DscResource, you should see any resource modules you've installed; if you don't, *get that sorted out before you go any further.*

The main problem I've seen is simply having the resource module in the wrong location. Have a look at the PSModulePath environment variable on your computer - it should include the /Program Files/WindowsPowerShell/Modules folder as one of its locations. If it doesn't, fix it (which needs to be done outside PowerShell, like from the System Properties Control Panel applet). Or, locate your modules in one of the locations that PSModulePath *does* contain (ideally not the System32 location; that's meant for Microsoft only).

Figuring Out What a Resource Wants

Once a resource is installed and PowerShell can find it, you're ready to start figuring out how to use it. In the earlier chapter on authoring basic configurations, I noted that a configuration setting looks something like this:

```
ResourceName UniqueName {
    Property = Value
    Property = Value
}
```

The ResourceName is easy to figure out, because it's shown right in the output of Get-DscResource. But what properties does the resource support? What values do they accept? *Hopefully* there's some documentation out there that can clue you in! Microsoft has gotten better about publishing documentation for a lot of their resources, although as I write this they're still working on adding more docs. But don't rely on the Get-Help command in PowerShell - at this time, it's not wired up to provide help on DSC resource modules, unfortunately. Well, not directly.

Have a look at C:/Windows/System32/WindowsPowerShell/v1.0/Modules/PSDesiredStateConfiguration/en-US on your computer. You should see a whole bunch of help files. Some are for commands like Disable-DscDebug, and you can use the ordinary Help command to view that documentation. Some of the rest of it is a little harder, because you can't just run "Help PSDesiredStateConfiguration" and get that documentation. What you can run, however, is this:

```
PS C:\Windows\System32\WindowsPowerShell\v1.0\Modules\PSDesiredStateConfiguration\en\
-US> help -path .\PSDesiredStateConfiguration.psm1-help.xml
```

That is, by using the -Path parameter and providing the path of the XML help file, you can read the documentation right in the shell. So that's definitely a starting point. Now, let's use that to get resource-specific. Keeping with the same PSDesiredStateConfiguration module, if I look at its root directory (that is, a level up from en-US), I see these entries (amongst others):

```
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        7/10/2015   4:04 AM                DownloadManager
d---s-        7/10/2015   6:11 AM                DSCResources
d---s-       11/17/2015   9:13 AM                en-US
d---s-        7/10/2015   4:04 AM                WebDownloadManager
```

The *resource module* can contain one or more resources, and each actual *resource* is in the DSCResources subfolder:

```
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d---s-        7/10/2015   6:11 AM                en-US
d---s-        7/10/2015   6:11 AM                MSFT_ArchiveResource
d---s-        7/10/2015   6:11 AM                MSFT_EnvironmentResource
d---s-        7/10/2015   6:11 AM                MSFT_GroupResource
d---s-        7/10/2015   6:11 AM                MSFT_LogResource
d---s-        7/10/2015   6:11 AM                MSFT_PackageResource
d---s-        7/10/2015   6:11 AM                MSFT_ProcessResource
d---s-        7/10/2015   6:11 AM                MSFT_RegistryResource
d---s-        7/10/2015   6:11 AM                MSFT_RoleResource
d---s-        7/10/2015   6:11 AM                MSFT_ScriptResource
d---s-        7/10/2015   6:11 AM                MSFT_ServiceResource
d---s-        7/10/2015   6:11 AM                MSFT_UserResource
d---s-        7/10/2015   6:11 AM                MSFT_WaitForAll
d---s-        7/10/2015   6:11 AM                MSFT_WaitForAny
d---s-        7/10/2015   6:11 AM                MSFT_WaitForSome
d---s-        7/10/2015   6:11 AM                MSFT_WindowsOptionalFeature
```

Those are not the actual *resource names* that you'd use inside a configuration script, but they're obviously similar. Here's what's in MSFT_GroupResource:

```
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d---s-        7/10/2015   6:11 AM                en-US
-a---l        7/10/2015   4:01 AM          57990 MSFT_GroupResource.psm1
-a---l        7/10/2015   4:01 AM            834 MSFT_GroupResource.schema.mof
```

The .psm1 file is the actual code that makes the resource work, while the .schema.mof file tells DSC how to use the resource. The en-US folder…

```
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a---l        7/10/2015   6:11 AM           2080 MSFT_GroupResource.schema.mfl
-a---l        7/10/2015   6:11 AM           4850 MSFT_GroupResource.strings.psd1
```

Er, doesn't contain an XML help file. It looks like it contains English versions of whatever text strings the resource uses. In theory, other folders could provide translations. So, no help there. Only… there *is*.

```
PS C:\Windows\System32\WindowsPowerShell\v1.0\Modules\PSDesiredStateConfiguration\DS\
CResources\MSFT_GroupResource> Get-Content .\MSFT_GroupResource.schema.mof

[ClassVersion("1.0.0"), FriendlyName("Group")]
class MSFT_GroupResource : OMI_BaseResource
{
  [Key] string GroupName;
  [write,ValueMap{"Present", "Absent"},Values{"Present", "Absent"}] string Ensure;
  [write] string Description;
  [write] string Members[];
  [write] string MembersToInclude[];
  [write] string MembersToExclude[];
  [write,EmbeddedInstance("MSFT_Credential")] string Credential;
};
```

That's what's in the .schema.mof file. As I wrote, this tells DSC how to *use* the resource, and that includes listing the properties that the resource requires. The resource's name - it's "friendly name," which you'd use in a configuration script - is "Group." It accepts several properties, including GroupName, Ensure, Description, Members, and so on. The Ensure property accepts "Present" or "Absent" as legal values. The GroupName property is the *key*, meaning each instance of this resource that you use within a configuration script must have a unique GroupName.

WAIT! Does that mean *you can only reference a given user group one time per configuration?* Yup. That restriction prevents two different settings within the configuration from "fighting" with each other. This is the thing I mentioned being troublesome for partial configurations, because you could easily send two partials that each attempted to configure the same group. Once the target node LCM merged those partials, it'd realize there was a duplicate key, and fail the entire configuration.

But actually... there's no need to dig into the schema MOF. I mean, feel free, and it's always good to know what's going on under the hood, but PowerShell will parse this for you a bit.

```
PS C:\> Get-DscResource Group | Select -ExpandProperty properties

Name                 PropertyType   IsMandatory Values
----                 ------------   ----------- ------
GroupName            [string]              True {}
Credential           [PSCredential]       False {}
DependsOn            [string[]]           False {}
Description          [string]             False {}
Ensure               [string]             False {Absent, Present}
Members              [string[]]           False {}
MembersToExclude     [string[]]           False {}
MembersToInclude     [string[]]           False {}
PsDscRunAsCredential [PSCredential]       False {}
```

See? Same info, only here you also see the "magic" properties that PowerShell adds, like PsDscRunAsCredential. The Group resource predates the existence of that, which is why the resource provides its own Credential property.

OK - now you know what properties exist, and maybe some of the values they take. But what about the rest? What about Members? What format should that be in?

Well… er… the thing is, Microsoft is really developing resources at a fast clip. They're very "agile" these days, which translates to "lots of code, not so much documentation." In Microsoft's *specific* case, they're trying hard, though. Browsing their GitHub repo at https://github.com/PowerShell/PowerShell-Docs/blob/staging/dsc/groupResource.md, I found some documentation and examples. And so… you kinda hope that, for whatever resource you're using, whoever wrote it has done something similar.

Look, let's just be honest about something for a minute. We're on the bleeding edge of computing with this DSC stuff. Things are in constant flux, and in a continuous state of improvement. If you're going to play in this space, you're going to have to accept that it's a bit more like being an open source coder, and a lot less like being a Wizard-clicking administrator. Expect to hunt for docs in GitHub. Expect to "figure things out" on your own, or by asking for help on community sites. If you're cool with that, you'll have a lot of fun with DSC. If you're not so cool with that… well… it's kinda gonna suck a little. We're in the hack-y side of Microsoft now, the side that's embraced open source and agile development. This isn't like Windows 95, where everything you needed came in a box or floated down through Windows Update. Hopefully, you're excited to be in the brave new world of Microsoft.

Custom Resources

The time will come - probably quickly - when you need to do something with DSC that isn't supported by a resource that Microsoft or someone else has conveniently provided you. That's when you'll need to write a custom resource.

Before We Begin: Function-Based vs. Class-Based

In WMF v5 and later, there are two types of resources that you can create. A *function-based resource*, which we've had since WMF v4, is just what the name implies: it's more or less a regular PowerShell script module, and it must contain three functions that have specific names and follow a specific usage pattern in their parameters. Get any of the specifics wrong, and it won't work. But aside from being careful about the naming, they're pretty straightforward, and they're what I'll be demonstrating in this chapter.

WMF v5 introduces *class-based resources*. Classes are an artifact of object-oriented programming. Honestly, from a code perspective, they're not all that different from functions. They look a little different, and you sort of lay out the code a little bit differently, but they're really similar. In object-oriented programming, one class can *inherit* from another class. So, suppose you have a base class called "vehicle" (all programming concepts can be illustrated through a vehicle analogy, it turns out). This "vehicle" class defines some of the common things all vehicles have, like a color, a model, and a manufacturer. The Vehicle class might also define some basic capabilities, like turning left or right, accelerating, decelerating, and so on. You might then code up a "car" class, which inherits from the Vehicle class. So the Car class magically has a color, model, and manufacturer property, because it inherited those from Vehicle. The Car class would also be *required* to implement capabilities like Turn, Accelerate, and Decelerate, because the base class demands that those be dealt with. Similarly, a Bicycle class would have to *implement* the same things, although its actual code for decelerating would obviously be different from that of the Car class.

So, classes in DSC resources inherit from a base class defined by Microsoft. This forces your class-based resource to have certain specific things, and editors like the PowerShell ISE can "check" you, to make sure you have those, before you even run the code. This helps eliminate the possibility of you mis-naming something, whereas in a function-based resource you're pretty much on the honor system to name things correctly. In other words, by using a class-based resource, you're less likely to screw up.

Professional developers love classes, and object-oriented programming, because in larger software projects a good OOP design can really make life easier. Code reuse is easier, screwing up is harder, and even debugging can be easier if you're doing it right. All my developer friends went nuts when PowerShell v5 added support for classes. Frankly, though, as an administrator, I was nonplussed. I

always used snippets, for example, to start my function-based resources, and so I didn't care much about being "forced" to use the right naming conventions. My DSC resources (as you'll see) are extremely simplistic from a code perspective, and so classes just didn't seem to offer an advantage to me. But, whatever. I'm easygoing, and if you want to use classes, have fun.

But there's a downside in the current iteration of WMF v5, and it's a downside specifically for classes. You see, for a function-based resource, there's a special file naming convention when you ZIP up the resource for distribution to nodes. That convention lets the ZIP filename include the module's version number. Due to that convention, you can actually store multiple versions of the same module on the same pull server. Configuration scripts can specify which version of a module they want, and the nodes will grab the right version from the pull server. It's really nifty - it means you can deploy a revised module to a small number of nodes for testing, and then gradually roll it out simply by changing the configuration MOF that your nodes are pulling. Classes, however, don't support that versioning in 5.0. It wasn't so much an oversight on Microsoft's part as it was the company's desire to "figure out versioning" in a much broader sense. The good news is that proper versioning for side-by-side deployment of classes is supported in 5.1. It's a good thing, too, because doing any kind of continuous integration with an automated build pipeline would be impossible without versioning. Now with this change in 5.1, I can roll out a new version on a schedule, like I can with function-based resources.

Writing the Functional Code

In the next chapter, I'll be covering class-based resources, and after that I'll get into my best practices for resource design. For now, the short version is, "put all of your actual functionality into a normal PowerShell module, and just call those commands from the resource." The idea is that a resource, whether class- or function-based, is just an "interface." An adapter, if you will. Just as a GUI is an adapter between code and your eyeballs, a resource is an adapter between code and the LCM. When Microsoft wrote the Service resource, they didn't make low-level API calls from within the resource to start, stop, and configure services, right? No, the resource merely calls Start-Service, Set-Service, and so on. Those commands are external to the resource. And so now, we'll create a simple PowerShell module that will provide the functionality for an example resource.

Let me once again stress that the focus here is on *how to structure this*, not on how to write proper PowerShell functions and modules. If you're not already up on those practices, I heartily recommend *Learn PowerShell Toolmaking in a Month of Lunches* by myself and Jeffery Hicks.

```
Function Test-FileContent {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$True)]
        [string]$Path,

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

    if (Test-Path $Path) {
        # file exists; get it as a single string
        $ExistingContent = Get-Content -Path $Path | Out-String

        # compare
        if ($ExistingContent -ceq $DesiredContent) {
            Write $true
        } else {
            Write $false
        }

    } else {
        # file does not exist
        Write $false
    }

}

Function Set-FileContent {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$True)]
        [string]$Path,

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

    Set-Content -Path $Path -Value $DesiredContent

}
```

So, as you can see, I have a “Test” function that’s comparing a file’s contents to some desired set

of content (yes, I realize the native File resource can do this, but the idea here is to make a simple example). I also have a "Set" function - and no "Get" function. I'm doing this deliberately to make a point, which will be explored in more detail in the "best practices" chapter coming up. Briefly, though, here's my philosophy: I want my actual resources to contain *as little code as possible*, because resources are harder to test and troubleshoot on their own. Ideally, I think of resources as just running one or two commands, and maybe applying a tiny bit of logic. Anything more complex than that, I separate into a "functional" module, which is just a normal PowerShell script module that I can code, test, and troubleshoot on its own.

In this case, I didn't include a Get-FileContent command because, frankly, it'd just be a thin wrapper around Get-Content. I know Get-Content already works, so I don't need to put a wrapper around it. In fact, my Set-FileContent function *is illustrating that point*, because it's nothing but a thin, thin, thin wrapper around Set-Content. Really, there's no need for my Set-FileContent - it isn't doing anything more than running a command that's already known to work. I included it here so that you could *see* how minimal it is.

I would perhaps put this script module in my usual Modules folder. That way, when the LCM needs it, PowerShell will be able to find it. However, it's also possible to deploy the script module along with the resource module. I'll discuss that later in this chapter.

Writing the Interface Module

What I'm calling the "interface module" is the actual DSC resource. There are some requirements for function-based resources that are a bit hard to create manually, such as the schema MOF. I'll get to all of that - and an easier way to do it - in a bit; for right now, let's just look at the code.

```
function Get-TargetResource {
    [CmdletBinding()]
    [OutputType([System.Collections.Hashtable])]
    Param(
        [Parameter(Mandatory=$True)]
        [string]$Path
    )

    If (Test-Path $Path) {
        $content = Get-Content -Path $path | Out-String
    } else {
        $content = $null
    }

    $output = @{
        Path = $Path
        Content = $content
```

```
    }
    Write $output
}

function Test-TargetResource {
    [CmdletBinding()]
    [OutputType([System.Boolean])]
    Param(
        [Parameter(Mandatory=$True)]
        [string]$Path,

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

    Write Test-FileContent -Path $Path -DesiredContent $DesiredContent

}

function Set-TargetResource {

    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$True)]
        [string]$Path,

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

    Set-FileContent -Path $Path -Content $Content

}
```

Notice that my script module includes three functions, which have specific names I must use: Test-, Get-, and Set-TargetResource. The LCM only knows to call these names, and you must use those names, although the functions can appear in any order within your module.

In the Get- function, notice that I'm returning a hashtable that contains whatever configuration currently exists. You provide the path, and I'll get you the file content. This is used when you run Get-DscConfiguration.

Next is the Test- function, which *uses the function I created earlier* and returns either True or False. This is used when you run either Start-DscConfiguration or Test-DscConfiguration, and of course

when running a consistency check. If the LCM is configured to autocorrect, then Test- returning a False will cause the LCM to immediately run Set-.

Set- illustrated how pointless my Set-FileContent function is, since it isn't any more complex than just running the underlying Set-Content directly.

There's an important procedural point, here: if the LCM runs Test, and if Test returns False, then the LCM may (if configured) run Set. *The LCM assumes the Set to have been successful if the Set function does not throw an error*. That is, the LCM doesn't run Set, and then run Test again to verify. It simply runs Set. So it's important to write a *good* Set function, one that can return errors if it fails. In this case, I'm not trapping any errors, so if one occurs - like a permission denied, or "Path does not exist," or something, then that error will be thrown, and the LCM will "see" it and know that the Set didn't work.

So this is now a minimally functional DSC resource.

Preparing the Module for Use and Deployment

First, you need to make sure that your DSC resource is in the right file structure. **This applies to the DSC resource, not to the "functional module" that I created first**. That functional module gets stored just like any other PowerShell script module. Well, with an optional exception that I'll get to in a sec. The base folder for a resource module needs to go into one of the folders listed in the PSModulePath environment variable, like /Program Files/WindowsPowerShell/modules. For example, suppose my DSC resource module is named MyDSCStuff, and I've written a function-based resource named AppControl, which will live within that resource module. I've also written a second resource named AppUsers, which will live in the same resource module, so that they're distributed as a set.

Note that you should **not** store resources in your Documents folder. The LCM runs as SYSTEM, and won't be able to find stuff in Documents. Use the /Program Files location unless your PSModulePath defines some other globally-accessible location. A UNC also won't work, as SYSTEM can't normally access non-local files.

- /Program Files/WindowsPowerShell/Modules/MyDSCStuff/MyDSCStuff.psd1 is the "root" for the resource module. You can run New-ModuleManifest to create MyDSCStuff.psd1.
- /Program Files/WindowsPowerShell/Modules/MyDSCStuff/DSCResources is the folder where the individual resources will live.
- /Program Files/WindowsPowerShell/Modules/MyDSCStuff/DSCResources/AppControl/AppControl.psd1 and AppControl.psm1 are the function-based resource AppControl. The .psm1 file contains my three -TargetResource functions, and the .psd1 should simply define the .psm1 file as the root module, and specify the three -TargetResource functions to be exported (technically, the .psd1 is optional, so you can omit it). You also need AppControl.schema.mof, which we'll come to in a moment.

- /Program Files/WindowsPowerShell/Modules/MyDSCStuff/DSCResources/AppUser/AppUser.psd1 and AppUser.psm1 are the function-based resource AppUser. You also need AppUser.schema.mof, which we'll come to in a moment.

The .psd1 files are always a good idea to include, because they can speed up certain background operations PowerShell has to perform, especially when the .psd1 file explicitly exports the three -TargetResource functions. This isn't as much a huge deal as it is for normal PowerShell modules, but having a manifest - the .psd1 file - is a good habit to be in.

The .schema.mof file is essential for each resource. And… um, you kind of have to build it by hand. Normally. Fortunately, Microsoft has a Resource Designer that can actually make the schema MOF for you, in addition to setting up all the right files and folders for your resource. Check out https://msdn.microsoft.com/en-us/powershell/dsc/authoringresourcemofdesigner. It's a set of commands that can build out the necessary files and folders, as well as the right function names, all for you.

Now… I mentioned an optional exception for dealing with your "functional" module, remember? So far, I've told you that your functional module needs to be a normal PowerShell script module. It lives apart from the DSC resource, meaning you'll have to distribute it independently. That might be fine with you - if you're smart, and you already have a good module-deployment mechanism in place (like a private NuGet repository or PowerShell Gallery instance), then it's no big deal. But you can *also* include the functional module with the DSC resource module, since we're using function-based resources. Following the above folder structure example, /Program Files/WindowsPowerShell/Modules/MyDSCStuff/MyDSCStuff.psm1 would be the filename for your "functional" module. MyDSCStuff.psd1 should define that .psm1 as the root module, and should export whatever functions the module contains. With this approach, the functional module will still be an operationally independent entity, but it'll be "packaged" with the DSC resources.

So… with the right folder structure in place, just ZIP it all up. You should be creating MyDSCStuff.zip (in this example), and when you unZIP it, it should recreate the correct folder structure. And finally, change the filename to include a version number, such as MyDSCStuff_1.2.zip. Then, create a checksum file using New-DscChecksum, so that you also have MyDSCStuff_1.2.zip.checksum. Those two files go to your pull server's Resources folder, and your nodes will be able to find them there.

Triggering a Reboot

All resources run within a PowerShell scope created by the LCM. When it spins up each new scope, the LCM creates a global variable named $global:DSCMachineStatus. Setting this variable to 1, from anywhere in your resource's Set code, will tell the LCM that a reboot is needed. The LCM configuration's RebootNodeIfNeeded setting will determine if the node is rebooted immediately or not.

It's important to understand that this behavior only works within your Set code - a Get or Test cannot trigger a reboot. Further, your Set must complete without error (which is how the LCM knows it was successful).

Class-Based Custom Resources

Class-based resources were introduced in WMF5. They offer a few core differences over function-based resources:

- They don't require a schema MOF.
- The class structure is enforced by PowerShell, meaning you have to try harder to mess up the Get-, Set-, and Test- names.
- In a class, the **return** keyword writes something to the output pipeline (just like Write or Write-Output), and then immediately exits the function (outside of a class, **return** is essentially an alias to **Write** and does not immediately exit).

If you've put most of your code into a standalone script module, as I recommended in the previous chapter and will elaborate on in the next chapter, then switching from a function-based resource to a class-based resource is easy. In fact, all I'm going to do in this chapter is "translate" what I did in the previous chapter.

However, there's a downside to class-based modules in WMF 5.0, which is that they don't support filename-based versioning. This makes them a little problematic in real-world applications, because you can't have two versions of a class-based module living side-by-side on nodes or on a pull server. Filename-based versioning is supported in WMF 5.1.

Writing the Class-Based Interface Module

Here we go: a class-based version of the previous chapter's resource module.

```
1  [DscResource()]
2  Class FileContent {
3          [DscProperty(Key)]
4          [string]$Path
5
6          [DscProperty(Mandatory)]
7          [string]$DesiredContent
8
9          [FileContent] Get() {
10             If (Test-Path $Path) {
11                 $content = Get-Content -Path $path | Out-String
12             } else {
```

```
                        $content = $null
            }

                $this.DesiredContent = $content
                $this.Path = $Path
                return $this

        }

        [void] Set() {
                Set-Content -Path $this.Path -Content $this.DesiredContent
        }

        [bool] Test() {
                return Test-FileContent -Path $this.path -DesiredContent $this.DesiredContent
        }
}
```

The Get method, replacing Get-TargetResource, is the one to look at. Notice that, above it, we're defining the parameters for this resource. They don't go into a Param() block, and I have to define the key property explicitly (which was done in the schema MOF of the function-based resource, rather than in the function itself). Rather than constructing a hashtable to output, I use the special $this object. $this has properties which match the DscProperty items defined at the top of the class - so, in this case, it has Path and DesiredContent properties. I set those, and then return the $this object. The Get() method is declared as returning an object of the type FileContent, which is the name of the class I created.

The Set() method returns nothing (the [void] part), and accesses the $this object to get the input parameters. Here, I just went with the Set-Content command rather than my pointless Set-FileContent "wrapper" function.

The Test() method returns $True or $False (that is, Boolean, or [bool]), and is simply running the Test-FileContent function I wrote previously. Again, it uses $this to access the input parameters.

As you can see, if you're keeping your resources simple in terms of functionality, and relying on functionality written elsewhere (either in PowerShell commands or functions that you've written in a normal script module), then deciding on a class-based or function-based resource is easy. They're very similar, once you really strip down all the code they contain.

Preparing the Module for Use

Class-based resource modules get laid out more simply than function-based ones:

/Modules/MyModuleName/MyModuleName.psm1

/Modules/MyModuleName/MyModuleName.psd1

There's no "DSCResources" subfolder hierarchy, in this case. That means you can't distribute your separate, "functional" script module in quite the same way. In this case, you'd simply drop it into the same folder, perhaps naming it something like MyModuleName-Support.psm1 or something. Your class-based module (MyModuleName.psm1) would need to load it explicitly, or your manifest (the .psd1 file) would need to specify it as an additional module to load along with the MyModuleName.psm1 root module.

Best Practices for Resource Design

This chapter is intended to put some context around the previous two. While they focused on *how* to write resources, this chapter will focus on *what to put in* a resource. As you've seen, the physical act of creating a resource ain't hard. What's hard is deciding what code to put into it.

Now, I should point out that Microsoft's resources aren't necessarily a universally great example of good resources. Many of the original Microsoft "DSC Resource Kit" resources were cranked out with great rapidity, before "practices" really came into question. And while the practices I'm going to present here are the ones I've developed, I can toss out a litany of names: Murawski. Helmick. Snover. Gates. Clinton. Eastwood. All of those names have either endorsed my design ideas, or have no idea whatsoever that this stuff even exists. Heh.

Principle One: Resources are an Interface

There's a guiding concept in software development that *implementation* and *presentation* should be as separate as possible. There are, in fact, entire dev frameworks dedicated to making that separation easier, such as Model-View-Controller (MVC) frameworks. The idea is that all of your *functional* code should live in one place, and the code that users interact with - the "presentation" - should be entirely distinct. The only code in the presentation piece is the code that makes the presentation actually work.

Take the Exchange Management Console as an example. It's a GUI, so it's an *interface*, or presentation element. It has relatively little code in it. When you ask the EMC to create a new mailbox, *none of the code that creates the mailbox lives in the EMC itself.* Instead, the EMC is simply calling New-Mailbox. The EMC takes the results of that command and creates some kind of graphical display - updating a list, or displaying an icon, or whatever it does. The actual *functional* code lives in the command, which lives in a PowerShell module. You can, of course, run the same command from a different interface - such as the PowerShell console. That demonstrates true independence between the functional code and the interface, and it's a hallmark of solid PowerShell design.

DSC resources work the same way, or they should. Resources should contain as little code as humanly possible, and what code they contain should be focused entirely on making the resource itself work. You should be able to look at almost every line of code in a resource and say, "yup, that's only there because DSC needs it to be there." The rest of your code should live in a traditional PowerShell module, and the resource should simply call that module's commands.

Thinking About Design

The reason Principle One (and there are almost no other principles here) can be a little hard for people to understand is that we're not always accustomed to designing software from the end state, and a DSC resource is definitely the end state.

Let me explain that.

When you build a house, you tend to start from the end state. That is, you hire an architect to draw pictures of what the house will look like. You nearly always start out with conceptual drawings - sketches, and so on - that show the completed house from various angles. You might have a rough floor plan, a sketch of the front elevation, and so on. But you don't hand those to the general contractor and start swinging hammers. You take that end state, and start designing the bits that will lead to it. You design the foundation. You figure out where plumbing is going to go through the foundation. You architect and engineer the walls and the roof. You design openings for doors and windows. So you begin with the end state in mind, but then you go all the way back to basics - to, literally, the foundation. A house is made up of all those foundational bits - cement, walls, wires, pipes, and so on.

Microsoft did that with DSC resources, but they did it over the course of an entire decade, so you don't really focus on it. In *The Monad Manifesto*, written before PowerShell existed, inventor Jeffrey Snover envisioned DSC. The book is on GitBook and LeanPub; go review it if you don't believe me. Way back before PowerShell even existed *as a product name*, Snover sketched out what DSC would look like, and it was the "end state" he envisioned. But what we got in 2006, when PowerShell was first released to the world, wasn't DSC. It was a concrete foundation. Version 2 of PowerShell added some walls and roofs. Version 3, some plumbing. It wasn't until version 4 that we got DSC - some ten years later.

Microsoft started, in version 1, by creating *cmdlets* for us. Individually, these cmdlets were fairly useful, but also pretty basic. Version 2 of PowerShell gave us the ability to create our own "script cmdlets" in the form of advanced functions. We also got Remoting, a key communications technology. In wasn't until version 4 that we could start combining cmdlets (script or otherwise) into DSC resources. So we were walked through that progression - foundation, walls, roof - pretty slowly. Slowly enough that now, looking at the product we have today, it's easy to forget that the progression even happened.

But when you design a DSC resource, you need to have that progression firmly in mind. You need to start *not* with the resource, but with the *commands* that the resource will use.

For Example

Suppose you have a Sales Order Entry application, and you want to instrument it for DSC management. The first thing to do is start thinking about your *nouns*. What are the elements of the system that you will need to administer? Perhaps you have **users** which will need to be

maintained. Perhaps there are **servers** that need to be configured, with each server having multiple **configuration elements**. These nouns, these "things you configure," become the nouns of your commands. Now, what do you do with them? You **add** users. You **delete** users. You **set** the properties of a user. Right there, you've got three commands: New-AppUser, Remove-AppUser, Set-AppUser. So you write those commands. You test them.

When the time comes to use all of that in a DSC resource, it should be easy. Your "AppUser" resource, for example, simply needs to implement its Test/Set/Get functions or methods, and in those it simply calls the commands you've already written. The resource isn't doing anything novel, it's just reusing commands that you've already written and tested. This is exactly how Microsoft actually approaches *most* of their resources. The ADUser resource doesn't contain code to talk to Active Directory; it just calls New-ADUser or Remove-ADUser or Set-ADUser, commands which already existed.

Advantages of the Approach

The main advantage here is the separation of presentation and implementation. By having your commands stand alone in a normal PowerShell module, they can be written by someone who has domain expertise in whatever is being managed, but who might know nothing about DSC. The commands can be tested independently. They can be verified independently. Each command is a small, standalone, testable unit - and "small code" equals "easier to maintain code."

Once you start using those commands in a resource, you know that the commands themselves *already work*. Resources, because they run remotely on a target node, are already amongst the hardest elements to test and debug. But using this approach, you've eliminated almost all of the uncertainty and opportunity for bugs, because you've tested the actual functionality. The resource, with its minimal code, introduces few new moving parts, and therefore fewer opportunities for new bugs.

Another advantage is that you can flip between function-based resources and class-based resources pretty easily. I mean, literally a few keyword changes in the script file, and you're more or less done. That's because all the *actual code* lives elsewhere, and the resource - whether function-based or class-based - is just an "interface" between the LCM and your functional code. Should Microsoft change the way resources need to be written, you should be able to update quickly and easily, because none of it will impact the functional code you wrote.

Disadvantage of the Approach

A *potential* disadvantage is that, in addition to distributing your DSC resource to target nodes - something the pull server is more than happy to do for you, if you're using one - you also have to distribute your "functional module" that contains all the commands the DSC resource uses. I don't regard this as an enormous problem. Most organizations should already have some kind of NuGet repository, or a private "PowerShell Gallery," if you will. If you don't, set one up. It's relatively easy.

Then, your DSC configurations can simply start by ensuring all dependent modules are installed from that repository. Anything relying on a module should contain a DependsOn, to make sure it runs after the dependent module is installed. This isn't difficult, and it's a more structured and reliable way to run your infrastructure.

The Script Resource

One of the native resources in DSC is the Script resource. This is intended as a sort of catch-all, quick-and-dirty way of writing your own custom resource, without all the bother of building out a class- or function-based resource. It's worth noting that Script resources can be a great way of prototyping an idea, but they're terribly hard to maintain over the long haul. The strong preference is to use class- or function-based resources rather than Script resources. But… there are a couple of neat things you can do with them.

The Basics

Take a look at this example (for your convenience, this is available in https://github.com/concentrateddon/TheDSCBookCode):

```
Configuration TestUsing {

param (
 [Parameter()]
 [ValidateNotNull()]
 [PSCredential]$Credential = (Get-Credential -Credential Administrator)
)

 node $AllNodes.Where({$_.Role -eq 'DC'}).NodeName {

  $DomainCredential = New-Object `
   -TypeName System.Management.Automation.PSCredential `
   -ArgumentList ("$($node.DomainName)\$($Credential.UserName)", `
    $Credential.Password)

  script CreateNewGpo {
   Credential = $DomainCredential
   TestScript = {
    if ((get-gpo -name "Test GPO" `
                 -domain $Using:Node.DomainName `
                 -ErrorAction SilentlyContinue) -eq $Null)
    {
     return $False
    } else {
```

```
      return $True
     }
    } #Test

    SetScript = {
     new-gpo -name "Test GPO" -Domain $Using:Node.DomainName
    } #Set

    GetScript = {
     $GPO= (get-gpo -name "Test GPO" -Domain $Using:Node.DomainName)
     return @{Result = $($GPO.DisplayName)}

    } #Get
  } #Script
} #Node
} #Configuration

TestUsing -OutputPath "C:\Using Example" -ConfigurationData ".\mj.psd1"
```

That'll create a simple MOF that uses a Script resource. Note that the Script setting has three properties: a GetScript, a TestScript, and a SetScript. These don't accept any input parameters, and they're run by the LCM just like the Get/Test/Set portions of a custom resource. The TestScript is expected to return $True or $False; the GetScript is expected to return a hash table with "Result" as the Key and a string value as the Value, and the SetScript is expected to set the desired configuration.

Cool Tricks

Notice the $Using trick in the example shown above? To make sense of that, let's first look at the ConfigurationData we fed to the configuration script. It's included in a separate file, and looks like this:

```
@{
    AllNodes = @(
        @{
            NodeName = '*'

            # Common networking
            InterfaceAlias = 'Ethernet'
            DefaultGateway = '192.168.2.1'
            SubnetMask = 24
            AddressFamily = 'IPv4'
            DnsServerAddress = '192.168.2.11'
```

```
12
13              # Domain and Domain Controller information
14              DomainName = "Company.Pri"
15              DomainDN = "DC=Company,DC=Pri"
16              DCDatabasePath = "C:\NTDS"
17              DCLogPath = "C:\NTDS"
18              SysvolPath = "C:\Sysvol"
19              PSDscAllowPlainTextPassword = $true
20              PSDscAllowDomainUser = $true
21
22              # DHCP Server Data
23              DHCPName = 'LabNet'
24              DHCPIPStartRange = '192.168.2.200'
25              DHCPIPEndRange = '192.168.2.250'
26              DHCPSubnetMask = '255.255.255.0'
27              DHCPState = 'Active'
28              DHCPAddressFamily = 'IPv4'
29              DHCPLeaseDuration = '00:08:00'
30              DHCPScopeID = '192.168.2.0'
31              DHCPDnsServerIPAddress = '192.168.2.11'
32              DHCPRouter = '192.168.2.1'
33
34             # ADCS Certificate Services information
35              CACN = 'Company.Pri'
36              CADNSuffix = "C=US,L=Phoenix,S=Arizona,O=Company"
37              CADatabasePath = "C:\windows\system32\CertLog"
38              CALogPath = "C:\CA_Logs"
39              ADCSCAType = 'EnterpriseRootCA'
40              ADCSCryptoProviderName = 'RSA#Microsoft Software Key Storage Provider'
41              ADCSHashAlgorithmName = 'SHA256'
42              ADCSKeyLength = 2048
43              ADCSValidityPeriod = 'Years'
44              ADCSValidityPeriodUnits = 2
45              # Lability default node settings
46              Lability_SwitchName = 'LabNet'
47              Lability_ProcessorCount = 1
48              Lability_StartupMemory = 1GB
49              Lability_Media = '2016TP5_x64_Standard_EN' # Can be Core,Win10,2012R2,na\
50 no
51                                                        # 2016TP5_x64_Standard_Core_EN
52                                                        # WIN10_x64_Enterprise_EN_Eval
53          }
54          @{
```

```
            NodeName = 'DC'
            IPAddress = '192.168.2.11'
            Role = @('DC', 'DHCP', 'ADCS')
            Lability_BootOrder = 10
            Lability_BootDelay = 60 # Number of seconds to delay before others
            Lability_timeZone = 'US Mountain Standard Time' #[System.TimeZoneInfo]::\
GetSystemTimeZones()
        }
<#
        @{
            NodeName = 'S1'
            IPAddress = '192.168.2.50'
            Role = @('DomainJoin', 'Web')
            Lability_BootOrder = 20
            Lability_timeZone = 'US Mountain Standard Time' #[System.TimeZoneInfo]::\
GetSystemTimeZones()
        }
        @{
            NodeName = 'Client'
            IPAddress = '192.168.2.100'
            Role = 'DomainJoin'
            Lability_ProcessorCount = 2
            Lability_StartupMemory = 2GB
            Lability_Media = 'WIN10_x64_Enterprise_EN_Eval'
            Lability_BootOrder = 20
            Lability_timeZone = 'US Mountain Standard Time' #[System.TimeZoneInfo]::\
GetSystemTimeZones()
        }
#>

    );
    NonNodeData = @{
        Lability = @{
            # EnvironmentPrefix = 'PS-GUI-' # this will prefix the VM names         \

            Media = @(); # Custom media additions that are different than the suppli\
ed defaults (media.json)
            Network = @( # Virtual switch in Hyper-V
                @{ Name = 'LabNet'; Type = 'Internal'; NetAdapterName = 'Ethernet'; \
AllowManagementOS = $true;}
            );
            DSCResource = @(
                ## Download published version from the PowerShell Gallery or Github
```

```
                @{ Name = 'xActiveDirectory'; RequiredVersion="2.13.0.0"; Provider =\
 'PSGallery'; },
                @{ Name = 'xComputerManagement'; RequiredVersion = '1.8.0.0'; Provid\
er = 'PSGallery'; }
                @{ Name = 'xNetworking'; RequiredVersion = '2.12.0.0'; Provider = 'P\
SGallery'; }
                @{ Name = 'xDhcpServer'; RequiredVersion = '1.5.0.0'; Provider = 'PS\
Gallery';  }
                @{ Name = 'xADCSDeployment'; RequiredVersion = '1.0.0.0'; Provider =\
 'PSGallery'; }
            );
        };
    };
};
```

Honestly, the important bit there is the DomainName property under the NodeName = '*' section. Because DomainName applies to all Nodes (e.g. the asterisk as a wildcard), each node that the configuration script processes will have a DomainName property applied.

Now go back and look at the $Using tricks in the configuration script. See how it's referring to **$Using:Node.DomainName**? We're telling the Script resource to grab the configuration data (represented by **Node**) and then get its DomainName property. In theory, this should return "Company.Pri" each time, because that's what our ConfigurationData block has for DomainName.

But keep something in mind: When the configuration script run, we're passing it the ConfigurationData block, right? So the computer running the configuration knows what the ConfigurationData is. The MOF it produces will contain the literal contents of the TestScript, SetScript, and GetScript blocks. *The code within those blocks is not executed or parsed when the MOF is produced!* Those three blocks aren't executed until the MOF is sent to a node, and that node's LCM loads up the MOF and begins a consistency check. *So how the heck does the node's LCM know what the configuration data is?* The answer is to look at the MOF generated by our configuration. This won't be pretty, and you're actually invited to run the code yourself to get the resulting MOF. It's actually almost impossible to reproduce the MOF in this book, in fact, because it's so wonky-looking.

What you'll find in the MOF, if you look at it in Notepad or something (keep in mind it's in the ZIP file, available as a book "extra" in LeanPub), is that each script block - GetScript, TestScript, and SetScript - has a serialized (e.g., translated to XML) version of the node's ConfigurationData. Keep in mind that GetScript, TestScript, and SetScript are all run independently, in separate PowerShell processes. That means the three of them don't "share" any data between them, which is why the ConfigurationData gets serialized into each block - meaning, the MOF contains three copies of the ConfigurationData. So when GetScript is run, for example, the LCM deserializes the ConfigurationData back into an object, and makes it accessible via **$using.Node**. In fact, right in the MOF, you can see this happen:

```
ResourceID = "[Script]CreateNewGpo";
 GetScript = "$Node = [System.Management.Automation.PSSerializer]::Deserialize(
```

Basically, when the MOF is created, PowerShell tacks on some extra code to the start of each script block, recreating the $Node variable. **$Using.Node** becomes a way to refer to $Node, which turns out to contain the ConfigurationData that was used to build the MOF. This is a cool way to pass information into the MOF, and then *use* that information during a consistency check on the node itself.

Part 6: Advanced Stuff

This is kind of our catch-all chapter for “extra information” that you may or may not find yourself needing. We’ll touch on troubleshooting, leading-edge stuff, some under-the-hood stuff, and stuff stuff. Note that many of these chapters will be under kind of constant expansion and revision, so if you purchased this book on LeanPub, make sure you’re opted into “new release” notifications via email!

Reporting

Sitting here writing this in July 2016, I have to make sure you understand that the default Report Server ain't much to look at. In fact, at PowerShell + DevOps Global Summit 2016, PowerShell team leads made it clear that the Report Server *was only ever intended as a sample* of what you could do. Azure, as I've already pointed out, takes reporting a lot further, and is a strong reason to use Azure Automation as your pull server. Maybe all that'll change someday, and maybe not. For now, my goal is to help you understand what the default server is and does. Note that this chapter only applies to WMF v5; the "Compliance Server" in v4 doesn't work the same way, and has been discontinued.

Understanding the Default Report Server

If configured to use a reporting server, each node's LCM will send a status packet to that reporting server. This happens each time the node runs a consistency check on its configuration, which means it can't happen if the LCM's application mode is disabled. The status information is sent in a specific format, and the Report Server's job is simply to accept it and store it in a database.

The default Report Server uses the Extensible Storage Engine, or ESE - the same database that Active Directory uses. *Presently*, the database schema is unknown, and it's not practical to move the data storage elsewhere. That's a potential weakness, as a lot of us would like to relocate that information to, say, a SQL Server that can be made more available and scalable. We're hopeful, as a community, that Microsoft can one day release the source code of their "sample" Report Server, so we can see about building our own.

Querying Report Data

The default Report Server lets you query node data by making calls to the same Web service that nodes report to. It's a REST call, which means you can simply use Invoke-WebRequest:

```
$requestUri = "https://my-report-server/PSDSCReportServer.svc/Nodes(AgentId='12345')\
/Reports"
$request = Invoke-WebRequest -Uri $requestUri  -ContentType "application/json;odata=\
minimalmetadata;streaming=true;charset=utf-8" `
                -UseBasicParsing -Headers @{Accept = "application/json";ProtocolVersi\
on = "2.0"} `
                -ErrorAction SilentlyContinue -ErrorVariable ev
```

That AgentId is the tricky bit. I've shown it here as 12345, which it will never be. It's a GUID, and it's configured as part of the LCM. Each LCM makes up its own AgentId. The result of this - stored in $request in the above example - is a JSON object.

```
$report = $request.content | ConvertFrom-Json
$report | Format-List -Prop *
```

In this case, $report will often contain multiple objects, since the Report Server keeps multiple check-ins for each node. Each of those objects will look something like this:

```
JobId                : 019dfbe5-f99f-11e5-80c6-001dd8b8065c
OperationType        : Consistency
RefreshMode          : Pull
Status               : Success
ReportFormatVersion  : 2.0
ConfigurationVersion : 2.0.0
StartTime            : 04/03/2016 06:21:43
EndTime              : 04/03/2016 06:22:04
RebootRequested      : False
Errors               : {}
StatusData           : {{"StartDate":"2016-04-03T06:21:43.7220000-07:00","IPV6Addres\
ses":["2001:4898:d8:f2f2:852b:b255:b071:283b","fe80::852b:b255:b071
                       :283b%12","::2000:0:0:0","::1","::2000:0:0:0"],"DurationInSec\
onds":"21","JobID":"{019DFBE5-F99F-11E5-80C6-001DD8B8065C}","Curren
                       tChecksum":"A7797571CB9C3AF4D74C39A0FDA11DAF33273349E11823855\
28FFC1E47151F7F","MetaData":"Author: configAuthor; Name:
                       Sample_ArchiveFirewall; Version: 2.0.0; GenerationDate: 04/01\
/2016 15:23:30; GenerationHost: CONTOSO-PullSrv;","RebootRequested":"False
                       ","Status":"Success","IPV4Addresses":["10.240.179.151","127.0\
.0.1"],"LCMVersion":"2.0","ResourcesNotInDesiredState":[{"SourceInf
                       o":"C:\\ReportTest\\Sample_xFirewall_AddFirewallRule.ps1::23:\
:9::xFirewall","ModuleName":"xNetworking","DurationInSeconds":"8.785",
                       "InstanceName":"Firewall","StartDate":"2016-04-03T06:21:56.46\
50000-07:00","ResourceName":"xFirewall","ModuleVersion":"2.7.0.0","
                       RebootRequested":"False","ResourceId":"[xFirewall]Firewall","\
ConfigurationName":"Sample_ArchiveFirewall","InDesiredState":"False
                       "}],"NumberOfResources":"2","Type":"Consistency","HostName":"\
CONTOSO-PULLCLI","ResourcesInDesiredState":[{"SourceInfo":"C:\\ReportTest\\Sample_xF\
irewall_AddFirewallRule.ps1::16::9::Archive","ModuleName":"PSDesiredStateConfigurati\
on","DurationInSeconds":"1.848",
                       "InstanceName":"ArchiveExample","StartDate":"2016-04-03T06:21\
:56.4650000-07:00","ResourceName":"Archive","ModuleVersion":"1.1","
                       RebootRequested":"False","ResourceId":"[Archive]ArchiveExampl\
e","ConfigurationName":"Sample_ArchiveFirewall","InDesiredState":"T
                       rue"}],"MACAddresses":["00-1D-D8-B8-06-5C","00-00-00-00-00-00\
-00-E0"],"MetaConfiguration":{"AgentId":"52DA826D-00DE-4166-8ACB-73F2B46A7E00",
                       "ConfigurationDownloadManagers":[{"SourceInfo":"C:\\ReportTes\
```

```
t\\LCMConfig.ps1::14::9::ConfigurationRepositoryWeb","A
                        llowUnsecureConnection":"True","ServerURL":"http://CONTOSO-Pu\
llSrv:8080/PSDSCPullServer.svc","RegistrationKey":"","ResourceId":"[Config
                        urationRepositoryWeb]CONTOSO-PullSrv","ConfigurationNames":["\
ClientConfig"]}],"ActionAfterReboot":"ContinueConfiguration","LCMCo
                        mpatibleVersions":["1.0","2.0"],"LCMState":"Idle","ResourceMo\
duleManagers":[],"ReportManagers":[{"AllowUnsecureConnection":"True
                        ","RegistrationKey":"","ServerURL":"http://CONTOSO-PullSrv:80\
80/PSDSCPullServer.svc","ResourceId":"[ReportServerWeb]CONTOSO-PullSrv","S
                        ourceInfo":"C:\\ReportTest\\LCMConfig.ps1::24::9::ReportServe\
rWeb"}],"StatusRetentionTimeInDays":"10","LCMVersion":"2.0","Config
                        urationMode":"ApplyAndMonitor","RefreshFrequencyMins":"30","R\
ebootNodeIfNeeded":"True","RefreshMode":"Pull","DebugMode":["NONE"]
                        ,"LCMStateDetail":"","AllowModuleOverwrite":"False","Configur\
ationModeFrequencyMins":"15"},"Locale":"en-US","Mode":"Pull"}}
AdditionalData       : {}
```

So you can see, there's some basic information before you get to the monster StatusData property, which includes a lot of detail about the node. It also contains a dump of all the configuration items that the LCM processed, so with a large configuration, this will be a LOT of data. The StatusData is itself a JSON object, so you could run it through ConvertFrom-Json to break that data out and make it a bit more readable. You'll also notice a HostName property buried in there, which is how you can start to map host names to AgendId GUIDs. Some things to note:

- You get a StartDate property that tells you when the consistency check ran. This allows you to find the most recent one.
- You get a DurationInSeconds property that tells you how long the check ran.
- You can tell if there's a reboot pending.
- You can see the Status, which will be Success *if the LCM finished without error*; Status does *not* necessarily mean the LCM is within compliance.
- ResourcesNotInDesiredState tells you which configuration settings aren't compliant. In this example, the Firewall resource isn't compliant. This can't tell you *why*, but it does tell you where to start investigating. ResourcesNotInDesiredState (and the companion ResourcesInDesiredState) are also JSON objects, and you can convert them to read them more easily.

The AgentId

The AgentId is perhaps the most vexing part of this setup. You'll notice, in the chapter on Configuring the LCM, that my LCM didn't display an AgentId. That's because the AgentId is dynamically generated the first time the LCM registers with a pull server. From then on, that AgentId is used to identify the node to the pull server and any Report Server that you've configured.

The problem is that most of us want to know the node's host name. I've seen a few novel approaches to solving this. One was a custom DSC resource which took the local AgentID, and then stored that in a "spare" attribute of the computer object in Active Directory. A similar approach registered DNS records, enabling you to query a host name to get its AgentId, or vice-versa. A third did basically the same thing in a custom database set up for that purpose.

I think a broader weakness just comes from the fact that the Report Server is a "sample." For example, you can't easily query it for just non-compliant nodes, or for nodes that need a reboot. You can't query for host names. Again, the Report Server included "in the box" just wasn't designed to be a robust tool. Until someone writes something better, though, it's what we've got to work with.

Security and DSC

There are a lot of security concerns with DSC. Perhaps the first that jumps to mind is, "our configurations contain a lot of important and proprietary information!" True. Also true is, "if someone can modify our configurations, then they can control our environment." So we'll try and address both of these concerns.

A Word on Certificates

Certificates play a strong role in DSC security. If you work in an organization that doesn't yet have a Public Key Infrastructure (PKI) set up, then you need to start there. Seriously, you can't do this stuff without PKI. Period. *Do not even think about using self signed certificates*, because they are Satan's handmaidens. Self-signed certificates will get you a false sense of security, but they will not, in fact, provide you with any protection. They are evil, misleading, wrong, and probably cause cancer.

DSC uses, for the most part, three types of certificates. Now, while all certificates are technologically the same in terms of the math and whatnot, certificates have to be "marked" for specific uses. These "use markings" indicate how much diligence was put into the certificate's issuance. A certificate used for encrypting email, for example, doesn't usually require much diligence in terms of verifying someone's identity. You just send the certificate to the mailbox it's intended to secure, and assume that whoever has access to that mailbox *should* have access. A code-signing certificate, on the other hand, requires a lot more diligence. You want the certificate indisputably to identify the developer (usually a company, not a person), so that if the code turns out to be malicious, you know who to go kill.

DSC uses:

- SSL certificates. These require moderately high diligence *not* because they provide encryption, but because they identify a server that is handing out configurations. You *do not* want your nodes connecting to some random web server to obtain their configurations! The SSL certificate ensures that the node is connecting to the right machine. For this reason, you should also avoid "wildcard" certificates - you don't want nodes connecting to "just any old" server, but rather to the specific server specified in their configurations.
- Client certificates. This isn't actually used by DSC per se, but you can certainly configure your web server to require client certificate authentication. This prevents unknown nodes from trying to grab your configuration data. Again - this isn't something you set up as part of DSC; client certificates would be issued to the node (perhaps through Active Directory auto-enrollment), and IIS (or whatever Web server you're using) would be configured to require the certificate.

- Document certificates. These are used to encrypt sensitive information like passwords in a MOF file. As of WMF5 and later, this is basically just a normal email encryption certificate, but it explicitly needs to be "marked" as usable for Document Encryption. This can be tough: in Windows Server 2012 R2, for example, ADCS doesn't provide a built in template for this kind of certificate, so you'll have to make your own.

So: SSL certificates on the pull server(s). Perhaps client certificates on the nodes, if you're doing that. Definitely document certificates for encrypting MOF data. No self-signed certificates. Got it?

Securing the Pull Server

The pull server is just a web server (well, usually; it could be a file server, which is actually a bit tougher to secure to the same level). You secure web servers by using SSL certificates. You also need to lock down your DNS infrastructure, to make sure nobody can hijack the pull server's DNS name and redirect traffic elsewhere.

Never run a non-HTTPS pull server, ever, anywhere, for any reason, even when you're just building a pilot or a lab. Never allow HTTP to become an easy habit. If HTTPS isn't easy, focus on making it easy. That's the investment that'll pay off.

You also need to lock down the actual files on the pull server. Make sure any SMB share permissions, and the NTFS permissions, are appropriate for your environment. "Everyone:Read" is not appropriate. Only designated configuration authors, and your web server, should be able to read the configurations. Only designated authors should be able to write those files.

Be smart with the pull server's security.

And some technical notes: Prior to WMF 5.1, the LCM only supported SSL 3.0 and TLS 1.0. If your server was doing anything else, the LCM would seize up. As of WMF 5.1, SSL 3.0 is no longer supported, and TLS 1.0, 1.1, and 1.2 are supported. You should definitely configure your server to use the most secure protocol available.

Securing Credentials in Configurations

DSC will not allow you, under normal conditions, to include clear-text passwords in a MOF file. However, you're very frequently going to need to include passwords, mainly when providing alternate credentials to a resource. So how do you do it? There are two options:

1. Flip the switch that tells DSC to allow plain-text passwords. This is blindingly stupid, and I'm not even going to show you how. It's on the Internet, if you're bound and determined to do the wrong thing.
2. Encrypt the password.

Option 2 is what we'll be pursuing. This requires, as I've outlined above, a certificate - and again, from WMF5 onward, the certificate must be "marked" for the "Document Encryption" purpose. There are a bunch of ways to generate such certificates, and in ADCS in Windows Server 2012 R2 onward, you can pretty easily create a certificate template that'll do this. There's a good overview of this encryption stuff[9] online; I'm going to walk through the basics and hit on some of the fine points.

You Need a Certificate

For right now, I'm going to assume you know how to issue certificates to your nodes. This is a section I'll expand on in a future edition of this book, though. Whatever certificate you issue will need to be installed - and this is important - in the *machine* store of the node, not in a *user* store. If you look at the CERT: drive in PowerShell, the machine store is easy to distinguish.

The nodes need the actual certificate installed; *wherever you are authoring configurations* need the *public key* from the certificate, and the certificate's thumbprint. You'll use the public key to encrypt, and the thumbprint to tell the node which certificate was used. The node will grab its copy of the certificate, which includes the *private* key, and use that key to decrypt the data.

The certificate needs to be precisely created. Its "Key Usage" must contain 'KeyEncipherment' and 'DataEncipherment', and should not contain: 'Digital Signature'. Further, the "Enhanced Key Usage" section of the certificate must include "Document Encryption (1.3.6.1.4.1.311.80.1)", and should not contain: "Client Authentication (1.3.6.1.5.5.7.3.2)" or "Server Authentication (1.3.6.1.5.5.7.3.1)". The Provider for the certificate must be "Microsoft RSA SChannel Cryptographic Provider".

Got all that? An ADCS template is the easiest way to set this up for consistent certificate creation. Those "should not contain" bits are important to helping lock down the certificate and preventing mis-use by a bad actor.

Verify the Certificate

Once the certificate is installed on the node, you can run this - on the node - to verify the certificate and get its thumbprint:

[9] https://msdn.microsoft.com/en-us/powershell/dsc/securemof

```
function Get-LocalEncryptionCertificateThumbprint
{
    (dir Cert:\LocalMachine\my) | %{
        # Verify the certificate is for Encryption and valid
        if ($_.PrivateKey.KeyExchangeAlgorithm -and $_.Verify())
        {
            return $_.Thumbprint
        }
    }
}
```

Defining a Configuration Data Block

Check this out:

```
$ConfigData= @{
    AllNodes = @(
            @{
                NodeName = "targetNode"
                CertificateFile = "C:\publicKeys\targetNode.cer"
                Thumbprint = "AC23EA3A9E291A75757A556D0B71CBBF8C4F6FD8"
            };
        );
    }
```

This Configuration Data block defines the node ("targetNode" is the computer name, here), the location of the certificate file (public key only) on the authoring computer, and the certificate thumbprint. Obviously, you can have a ton of these, one for each node. And yes, it really should be one for each node - sharing certificates is like sharing contraceptives. Ew.

The Configuration Script

Whatever password you need to encrypt should be typed as a [PSCredential] data type - this triggers the encryption behavior during MOF creation. For example:

```
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
        }
    }
}
```

See where we created the credential? PowerShell will prompt for it when you run this, and you can see where it's used ($credential) in the configuration. You need to run this and pass in the Configuration Data block:

```
CredentialEncryptionExample -ConfigurationData $ConfigData -OutputPath .\CredentialE\
ncryptionExample
```

So, we ran the configuration (I'd normally have this at the end of the script containing the configuration), and passed in the $ConfigData block I created earlier (which you might read from a separate file). This will generate a MOF for each node in $ConfigData. Because the $ConfigData has the local (on the authoring system) path for the certificate public key, and because it also has the thumbprint, it'll all just magically work.

Sharing Certificates

We made a fairly strong statement earlier about not sharing certificates, and we hold to this statement, but there are some good reasons for considering sharing certificates. Maybe you have a web server farm, and all of those servers are to be configured identically, and you're using Configuration Names for the configuration. You have one MOF file to configure all the servers in the web farm. You can't deploy that configuration to multiple nodes that each have a different certificate. In that case, it may make sense to use one certificate for the web server configuration and deploy that certificate to all the web servers.

Keep in mind that you could choose to go all the way to the opposite extreme and just use a single certificate for all servers in your organization, and that's a design decision with some serious tradeoffs. On one hand, it makes certificate management really easy, but on the other hand, if your private key gets compromised or the certificate expires without you knowing, you're probably going to have a really bad day.

Certificate Management

So… how do you manage all these deployed certificates? Well… that's still an evolving story. Honestly. A bunch of us at DevOps Camp (an event I run in August in Las Vegas for about 20 folks; keep an eye on DonJones.com for dates) have been fooling with ideas for automating installation, dealing with expirations and renewals, and so on, especially in non-domain environments. As practices emerge and are refined for this, I'll loop back and include that information in this book.

PSDSCRunAsCredential

This is a universal property that, as of WMF 5.0, can be used with any resource. When a configuration is run, the LCM will launch the resource using the provided credential. You should follow the guidelines above for encrypting those credentials in the MOF!

Note that in WMF 5.0, you can only use PSRunAsCredential with "real" resources - that is, ones that are an actual compiled, class-based, or function-based resource module. You cannot use PSRunAsCredential with a *composite* resource. However, that was changed in WMF 5.1, and you can pass a PSDSCRunAsCredential to a composite resource as well. The entire composite resource will run using the credential you provide. However, if the composite resource *itself* specifies alternate credentials, then the LCM will explode and your configuration won't run.

Digital Signing

This feature was added in WMF 5.1, and allows you to digitally sign configuration (MOF) files. If a node pulls a configuration from a pull server, and the signature isn't valid, the LCM will refuse to run the configuration. *This is a very good idea to implement*, because it helps protect against a compromised pull server. I know, you think, "we have firewalls, nobody can get to our pull server," to which I say, "defense in depth, man," and, "that's what the US Office of Personnel Management thought until June 2016." Sign MOFs using Set-AuthenticodeSignature *on a machine running WMF 5.1* (the cmdlet in previous versions can't sign MOFs).

Modules can also be signed. You have to start by creating a *catalog* for the module, which is accomplished by the New-FileCatalog cmdlet. Sign the catalog file using Set-AuthenticodeSignature - again, this all works only in WMF 5.1 and later - and put the catalog file inside the module folder. If the module is named OurTools, then the catalog filename would be OurTools.cat, and it would live in the OurTools module folder.\

Then you have to tell the LCM to expect signatures - see the chapter on configuring the LCM for that. Signatures are validated on a per-pull-server basis, and only when the LCM downloads files; it does not re-validate signatures when performing a consistency check. You can also configure the LCM to expect signatures on pushed configurations.

DSC in Azure

In the Introduction, and in the previous chapter, I've made a big deal out of the fact that DSC is a *technology*, and that using it can require a lot of manual effort because, right now, we don't have a lot of *tools* built atop the technology. But lurking inside Azure Automation is a preview of how those tools might look.

By the way, this isn't a sales pitch for Azure; it's a sales pitch for *tools*, and how they can make using a complex technology a lot easier.

In essence, what Azure Automation provides (for the purposes of DSC) is a Pull Server in the cloud. It isn't the standard Pull Server that you can install on-prem; with access to the Pull Server source code, the Azure team has been able to customize their offering quite a bit. But, at the end of the day, it's a Pull Server. You can copy your configuration MOFs to it, and you can drop you ZIPped-up DSC resource modules on it. Your nodes - whether they're running Azure, AWS, on-prem, or someplace else - can be pointed to this Pull Server. They provide some nice authentication options to make sure only *your* machines are pulling configurations, too.

You can also *author* your configurations right in the cloud, run them, and generate MOFs right there. And Azure's Pull Server supports the Reporting Server function - only they've enhanced it *with actual reports and dashboards*, so you can use an Azure-based console to track the compliance of your nodes (in-cloud and on-prem), generate reports, and so on.

Azure's Pull Server runs only on HTTPS, which means your nodes can be assured that they're connecting to *your* pull server, and not some imposter.

Azure provides a lot of integration points, especially for Azure-based VMs, to make node enrollment easier. For example, the Azure Automation DSC console lets you configure an Azure-based VM's LCM from a graphical interface, enabling you to get machines pointed to the Pull Server more easily. That configuration can also be "injected" into new VMs, enrolling them automatically upon creation. Nodes are onboarded securely, which means Azure equips them with a client certificate. This helps ensure that only *your* nodes can request configurations from your Pull Server.

Azure also provides tooling for some of the more complex and difficult aspects of DSC, such as managing credentials in MOFs. And keep in mind, you can use Azure DSC *even if you don't have any VMs in Azure*. It's just a fancy Pull Server, although you do have different onboarding steps for nodes that aren't running in Azure VMs.

DSC on Linux

Microsoft offers a free DSC client - that is, a Local Configuration Manager, or LCM - for Linux. It's important to know that this is *not* the work of the PowerShell team, but rather of Microsoft's excellent Unix Tools team. Here's a great getting started guide[10], and you can access the open source code on GitHub[11].

Authoring configurations is just the same, although the available resources are obviously different. Bundled in the LCM are the following resources:

- nxArchive
- nxAvailableUpdates
- nxComputer
- nxDNSServerAddress
- nxEnvironment
- nxFile
- nxFileInventory
- nxFileLine
- nxFirewall
- nxGroup
- nxIPAddress
- nxLog
- nxMySqlDatabase
- nxMySqlGrant
- nxMySqlUser
- nxOMSAgent
- nxOMSCustomLog
- nxOMSKeyMgmt
- nxOMSPlugin
- nxOMSSyslog
- nxPackage
- nxScript
- nxService
- nxSshAuthorizedKeys
- nxUser

[10] https://msdn.microsoft.com/en-us/powershell/dsc/lnxgettingstarted
[11] https://github.com/Microsoft/PowerShell-DSC-for-Linux

Most are written in C. It's worth noting that DSC on Azure - that is, Azure Automation - also supports Linux by means of this LCM client. Writing your own modules for the Linux LCM is a bit more complex, as you can't necessarily just write a PowerShell script (PowerShell didn't exist on Linux when this LCM was written, so they assumed you'd be coding in C).

The team also provides MOFs for their resources that can be installed on Windows, so you can author (getting IntelliSense and whatnot in the ISE) Linux-targeted configurations on Windows machines.

Troubleshooting and Debugging

Getting Eyes-On

The key thing with troubleshooting and debugging is simply *figuring out what's happening*. There are a number of ways in which you can do that.

Interactive Push

Get a copy of a node's MOF, switch the node's LCM to Push mode, and push the MOF using **Start-DscConfiguration -Wait -Verbose** (adding in the parameter to specify the MOF, of course). You'll get some of the best diagnostic information that way, if you can take this temporary troubleshooting step.

Get the Failing Node's Configuration

Use **Get-DscConfigurationStatus** to query the node's LCM directly. You'll get a list of failed resources, anything requesting a reboot, and a lot more. The LCM can also return historical information, as opposed to including only the most recent consistency check, when you add the **-All** parameter.

The command will return an object, one good trick is to run:

```
1  Get-DscConfigurationStatus | Select -Expand ResourcesNotInDesiredState
```

You'd run this right on the node, and you'll get a dump of every resource that's reporting non-compliance, along with error information.

Check the Logs

The LCM can also report extensive information to the Debug, Operational, and Analytic event logs on the node. For example:

```
1  Get-WinEvent -LogName 'Microsoft-Windows-Dsc/Operational'
```

Substitute "Analytic" or "Debug" for "Operational" to see those logs; the Debug log tends to hold the most detailed information. You can also access these using the Event Viewer GUI tool, although you'll need to right-click to make the Debug and Analytic logs visible.

- The Operational log contains all outright error messages.
- The Analytic log goes into more detail, and can include any verbose messages output by resources or the LCM.
- The Debug log contains the greatest level of detail.

Each log includes a Job ID, which corresponds to a single DSC operation. Once you query the Operational log, for example, you can start to see the Job ID from the most recent consistency check. You can then limit further queries to just that job ID, if you want. You'll find some helpful commands in the xDscDiagnostics module (which is available in PowerShell Gallery, so use Install-Module to get it). One command is **Get-xDscOperation**, which includes a **-Newest** parameter that can grab just the most recent number of operations. Another command is **Trace-xDscOperation**, which can consolidate events for a given operation from all three logs.

Stop Resource Module Caching

Normally, the LCM caches DSC resource modules for better performance. But when you're troubleshooting or debugging a module, you may want to force resources to be re-imported each time the LCM runs. The LCM's **DebugMode** property can be set to $True, which forces the LCM always to reload DSC resources, ensuring a "fresh" run. Set this back to $False when you're done debugging, so that the engine can go back to caching resources for better performance. So, to be clear, changing this requires that you modify the LCM configuration by pushing in a new meta-MOF.

Check the Report Server

If you've configured a node to use a report server, it will forward any reports - often more useful than what appears on-screen on the node - to that report server. It's worth querying the server to see what's up.

Resource Debugging

This'll become a full chapter at some point, but keep in mind the **Enable-DscDebug** and **Disable-DscDebug** commands. These enable debugging of DSC resource modules, which enables you to force resources to "break" into debug mode when the consistency check runs them. This is a neat way to "step through" the resource module's code and see what's happening. You'll usually add the **-BreakAll** parameter when enabling; see the command's documentation for more information.

Stopping a Hung LCM

Normally, running **Stop-DscConfiguration -Force** (and of course adding the right computer name) will get a computer's LCM to stop processing. But the LCM, today, is a single-threaded process,

and if it's gone off the deep end - usually the result of a resource module going off and doing some long-running task, improperly expecting interactive input, or something else - then the LCM might not respond to your stop request. In that case, try this *on the offending node*:

```
Remove-Item $env:systemRoot/system32/configuration/pending.mof -Force
Get-Process *wmi* | Stop-Process -Force
Restart-Service winrm -Force
```

This will remove the MOF that the LCM *would have processed on its next run*; you may also need to delete current.mof and backup.mof from the same location. You can also try running **Remove-DscConfiguration**, which does essentially the same thing.

The second bit - stopping the WMI service and restarting Windows Remote Management (WinRM) - can be useful, especially on older systems. Everything in DSC actually runs out of Windows Management Instrumentation (WMI), and in cases where it's the underlying WMI service holding things up, killing the process can get the LCM to unstick. You *may* need to restart the WMI service manually after doing this (or reboot the node). Similarly, because DSC uses WinRM to communicate, sometimes it's those communications that hang things up. Restarting WinRM (or killing the process and then starting the service) can unstick things. Thanks to Peter Burkholder for suggesting this section!

Self-Modifying Configurations

This whole chapter grew out of a "Stupid DSC Tricks" session I did at a couple of conferences, including PowerShell + DevOps Global Summit 2016 http://powershellsummit.org. The idea was to try and push the boundary on DSC a little bit. One goal in particular was to create a configuration that could download pre-requisites for the actual, intended configuration.

Here's the scenario: you want to push a configuration down to a node. Fine. But perhaps the configuration will contain credentials, which you'll be encrypting of course, and so you *first* need to get the certificate installed locally. Or perhaps you need to pre-stage some pre-requisite modules or resources, and can't use a pull server for whatever reason. Problem is, if the LCM sees that you're using resources it doesn't have, it won't even run the configuration. So you want to run a small, short, "bootstrap" configuration first, to get everything all set up, and *then* run the "real" configuration.

Now, look: I'm not positioning all of this as a Good and Just Idea. This chapter is more about pushing the edges of DSC to see what we can do. Think of it as a thought experiment, of sorts. But over the course of thinking through this, I've come up with what I think are some clever ideas. So this chapter will be less a how-to, and more a freewheeling stream-of-consciousness. Oh, you'll see some actual tech, too, but it's possible that the tangential ideas will end up being the most useful.

Understanding the LCM's Processing

The LCM actually stores its configuration on disk, in /System32/Configurations. It's easy to get to. There are no registry calls or anything else - the MOF file is just a plain text file in that folder. If the LCM has run before, then there can actually be multiple files in that location:

- Pending.mof is the newest configuration the LCM has received, and is the one the LCM will use on its next consistency check.
- Current.mof is the one the LCM is running right now.
- Previous.mof is the one the LCM used on its last consistency check.
- Backup.mof is usually the same as Current.mof, and unless a new configuration was JUST received, will also be the same as Pending.mof.

Therefore, the LCM gets a configuration and saves it as Pending.mof. When the next consistency check runs, this is copied to Current.mof and Backup.mof, and the consistency check starts. When the check finishes, Current.mof is copied to Previous.mof. If there's a failure of some kind, Pending.mof won't exist, thus preventing the "failed" configuration from being picked up again.

So we can "inject" a configuration by simply writing a MOF to Pending.mof and waiting for the LCM to run its next consistency check.

Additionally, Metaconfig.mof can contain the LCM's meta-configuration. Replacing this file is a way to "inject" a new meta-configuration, thereby reconfiguring the LCM. Note that Metaconfig.mof file can only contain the **MSFT_DSCMetaConfiguration** and the **MSFT_KeyValuePair** sections. But MOF files produced by running a PowerShell configuration script may contain an **OMI_ConfigurationDocument** section, which must be removed.

The Basic Self-Modifying Workflow

So, the basic idea is to push (or make available on a pull server) a "bootstrap" configuration. This is intended only to get whatever pre-requisite stuff you need in place. It looks and works just like a normal configuration… except, at the end, it does something a little different. It "injects" the *real* configuration, and possibly "injects" a new LCM meta-configuration.

So, with the understanding that this is all a little theoretical, here's a model of the idea:

```
Configuration Bootstrap {

        Node whatever {

                Prerequisite One {
                        Property = Value
                        Property = Value
                }

                Prerequisite Two {
                        Property = Value
                        Property = Value
                }

                File [string] #ResourceName
                {
                    DestinationPath = pending.mof
                    Ensure = Present
                    SourcePath = \\server\folder\source.mof
                    DependsOn = [Prerequisite]One,[Prerequisite]Two
                }

        }

}
```

So in broad strokes, the idea is to have one or more resources - "Prerequisite" resources, in this example - set up whatever you need on the node. A final resource injects the "real" MOF. In this

case, I'm using the File resource to copy the MOF from a file server, so the presumption is that the MOF has already been created and staged there.

Options

So, the basic theme is to create a "bootstrap" configuration, which will set up any prerequisites, and then somehow engage the "real" configuration. You've got a lot of options for getting there.

1. Set up a pull server where the bootstrap configurations reside. Configure the node LCM to pull from that server. The bootstrap then modifies the LCM to pull from a different pull server, where the "real" configuration resides.
2. Push the bootstrap to the node, and the bootstrap configures the LCM to pull the "real" configuration from a pull server.
3. Have the node pull the bootstrap configuration by GUID. The bootstrap then reconfigures the LCM to use configuration names, allowing it to pull the "real" configuration from the same pull server.
4. Push the bootstrap to the node, and have the bootstrap directly inject a new, "real" configuration. This is basically what my above example does.
5. Push the bootstrap to the node, or have the node pull it. This does whatever it needs to do, and then runs a PowerShell configuration script, producing a MOF. This MOF is the "real" configuration, and could be written locally (i.e. injected), or written to a pull server.

All of this "reconfigure the LCM" stuff can be done by having the bootstrap write a new meta-configuration MOF directly on the node. The LCM will pick up the "injected" meta-configuration MOF pretty quickly, thereby reconfiguring the LCM.

My fifth suggestion in the list above offers some interesting possibilities. Because, in that situation, the configuration *script* is running *on the target node*, the script can include logic that dynamically adjusts the final MOF based on local conditions. This means that a single complex configuration script could, when run on different nodes, produce entirely different MOFs. And as I noted in the list, the MOF could be written right back to a pull server, overwriting the "bootstrap" configuration and allowing the node to pull the new, dynamically produced MOF.

The possibilities really are endless, which is why I've found this to be such an interesting line of experimentation.

A Problem to Consider

One problem with "injecting" a MOF into the local node is the encryption introduced in v5. The deal is that the various MOF files stored by the LCM - including pending.mof, which is the MOF it'll process on its next consistency check - are encrypted on disk. Both Start-DscConfiguration and

Publish-DscConfiguration know how to encrypt the file correctly (technically, the encryption is done *on* the target node, not before the file is sent to the target node), but I'm not currently clean on how you might manually encrypt the MOF. If pending.mof isn't encrypted correctly, then the LCM won't read it, so your "injected" configuration won't work.

I've toyed with writing the MOF locally, but not in pending.mof, and then using Publish-DscConfiguration to send the MOF to the local host. So, the bootstrap configuration would create a local MOF (copy it from someplace else, create it locally, or whatever), and then basically "publish" the MOF to itself. I've run into some credential problems with this, though. The bootstrap is being run by the LCM, which runs as SYSTEM, which doesn't have permission to remote to itself. My only solution has been to provide a credential for the publish operation - but that necessitates encrypting the credential in the MOF. In order for that to work, you have to have the decryption certificate already staged - and getting the certificate there is one of the things I'd want to use the bootstrap for. So it's a bit of a Catch-22. I'm still playing with it, and I believe it should be possible to encrypt the MOF directly on the node, without having to "publish" it.

Crazy Ideas for What the Bootstrap Can Do

So what sorts of prerequisites might a bootstrap configuration handle?

- Install certificates for decrypting credentials in a MOF
- Install DSC resources that can't be delivered from a pull server
- Install modules that can't be deployed in any other way

And in addition to deploying prerequisites, the ability - as I outlined already - to have a PowerShell configuration script run *on the node* opens up a world of interesting possibilities for producing more "locally aware" MOFs that adjust themselves based on local conditions.

Anyway… it's all an ongoing experiment, which is why I included this so near the end of the book, but I think it's a useful way of thinking about how DSC can be twisted to solve different needs. And certainly, it's a demonstration of how important it is to understand how DSC works under the hood!

The Scaling Question

"How can we make DSC scale?" is a question that I've struggled to help people answer. It wasn't until an early-adopter of this book emailed me, suggesting I answer the question in this book, that I started to wrap my head around what I think is a good narrative. He asked:

> Don, Something I know we would love to see is how to scale DSC to hundreds of nodes. What keeps coming up for us is where/how to keep our "raw" configuration data. Something that manages node information (name, IP, domain, standardRoleA, standardRoleB, roleC, roleCparam1, roleCparam2). Put it in a database (but we think it might not be flexible enough) or json files? Is it really just Puppet or Chef? Seems like there is a space out there for what we think we're looking for - and maybe an advanced chapter with your thoughts down the road. Cheers, Joe

This winds up being something I think I touched on at the beginning of the book, but it's an important narrative. It's also complex - so I hope you'll bear with me. Also, and this is important, it's *still evolving*. What I'm going to try and share is the long-term vision, which is steadily coming to fruition, and I hope you'll understand that the current "missing bits" won't stay missing for long.

DSC Already Scales - But You Don't

It's important that we draw a distinction between scaling the *technology* and scaling your *processes for managing the technology*. DSC, as a platform, already scales easily. There are only two bottlenecks in the system, and they're both a Web server: the pull server and the Report server. While the Report server is a bit of an evolving story, Azure Automation shows us that you can absolutely create scale there. And the pull server is *just a Web server*, so you can scale it like any other IIS box.

What doesn't scale - and what Joe's question crystallized in my mind - is our processes for managing everything up to the point a MOF goes onto the pull server. How do we keep track of all our configurations, partials, composites, and whatnot? How do we remember which roles and fragments have been assigned to each server? How do we track the unique, per-machine information - like IP addresses - that need to be injected into Configuration Data blocks?

There are two answers.

Answer 1: Tooling

What we need is tools. And, as Joe suggests in his question, those tools might already exist in the form of Chef, Puppet, and so on. Or, you could roll your own, creating a database and some

kind of management front-end for it. That latter suggestion isn't so far-fetched, either; back in the PowerShell 1.0 days, MySpace.com essentially rolled their own configuration management system that accomplished a lot of what DSC aims to do.

Microsoft hasn't yet produced their own tooling for this, although Azure Automation - more so than System Center - is starting to take steps in that direction. But I'd argue that Answer 1 isn't the right answer. In fact…

Answer 2: Your Brain

…I'll argue that the question is wrong. Instead of asking ourselves how to scale our management processes, and how to create or buy tools that will enable that scale, I suggest we should be asking *how to get rid of those processes*. We're all essentially managing our servers the same way we have been since the 1990s, and we've used tools to make that more scalable. Deploying software to 10 clients is easy, but 10,000? Get SCCM! Use Group Policy! But all we're doing there is *automating* the same processes - we haven't actually *created processes that work at the needed scale*.

And so let's begin the narrative. You're going to spot numerous potential flaws in my story here, but stick with it, because I've had pieces of this conversation with other people already. I'm going to try and anticipate the arguments and see if I can't counter them.

Let's Set the Stage

Imagine that we work for an organization like Joe's, where we have numerous different server roles to keep track of. You probably don't need to imagine very hard, since you probably have the same thing. You've got some infrastructure servers, running Active Directory, DNS, DHCP, and so on. You have some application servers ,and some SQL Server machines. And all of these probably have some common configuration elements, like management agents, security settings, and so on.

Let's focus on those infrastructure servers as a piece of our example. Let's say you have eight domain controllers, and some number of those run DHCP Server and provide DNS services. Why eight? Well, probably because you figured out that's how many DCs you need in that location to be able to support the morning logon slam. And so you provision those machines, and you keep track of them. Now, AD actually has one scale problem that Microsoft hasn't addressed, and that's the need for a copy of the AD database to be local on the domain controller. That's actually unusual for Microsoft, because almost every other "database" they have - SQL Server, Exchange Server, you name it - can have the actual data files exist elsewhere, like on a SAN or SMB 3.0 file server. In fact, it suggests that AD is either going to evolve soon, or change significantly. But for now, it's important to keep in mind the fact that each DC needs a local database, and it can only get one by replicating with other DCs. Fine.

Raise Cattle, Not Pets

The problem is that you think of each DC as a unique and special individual - a pet. You monitor each one. You know when each one is working, or when it isn't. You manage updates on each one. Each one has a name, and you know that name. Each one has an IP address, and you track that - probably, God help us, in an Excel spreadsheet.

Why?

Let me propose an alternative. Suppose you have a monitoring system - SCOM, or OMS, or whatever. It knows, because you've told it, that you need eight DCs online in the mornings. But perhaps it also knows that, for much of the rest of the day, you only need three or four DCs. So throughout the day, it monitors three or four running DCs (I'm assuming these are in virtual machines, because why not?). As load starts to increase, the monitoring solution spins up DCs five, and six, and seven, and so on. Perhaps you even have it do that automatically, early in the morning, anticipating the logon rush. Those DCs are never offline for more than a day, and you can ensure they have plenty of time to replicate changes.

But if one of those DCs is *running* but not *working*, your monitoring solution doesn't try to fix it. It just kills it. It removes it from the domain, deletes the VM, and spins up a new copy. Because this isn't a *pet*, it's a *cow*. And when cows get uppity, you shoot them, eat them, and buy another cow.

Why in the world do you care what the DC's IP address is? You shouldn't. Nobody else does. It registers itself with DNS, so you can easily find it. If you insist, you could create a reservation for it in DHCP, and then you'd always know its IP address, but why bother? You can certainly ensure it gets an IP address from a pool you've designated, so it'll have a *good* IP address, but honestly - why do you care what it is? "Well, what if DDNS doesn't work, and the DC can't register itself?" Well, then you shouldn't have built a DDNS layer that can break down entirely. Fix your infrastructure.

And why, for that matter, do you care what the DC's name is? Windows will make up a name for the machine when the OS installs. It'll register itself as a domain controller. Clients will be able to find it - it's not like you give your users a list of DC names anyway. Maybe you have a DSC resource that ensures the randomly-chosen name is in fact unique, but otherwise - why do you care?

When your orchestration/monitoring system spins up a new DC, it can certainly add it to a "pool" of some kind, so it knows to monitor that new machine as a DC. At the end of the day, all you should care about is whether the right number of DCs exist, are running, and are responding to requests - and existing monitoring solutions can do all of that for you. Existing monitoring solutions can kick off corrective actions, too - starting up more VMs or commissioning new DCs. What value, exactly, do *you*, as a slow-moving ape descendent, hope to bring to that process? People should be making *decisions*: "We need eight of these cows at all times." Let the computers do the rest, and don't sweat the details.

Enter Containers

With the rise of containers, this story becomes even more important. We stop loading multiple roles on a VM, and instead run everything in a container. Granted, we can't run *everything* that way today, but this story is the reason to move in that direction. DHCP server quit responding? Screw it, kill the thing and spin up a new container - it'll take 30 seconds. "High availability" becomes something you worry about in the storage and networking layers, not in the "client interface" layer - because you just spin up new containers as needed. And with containers spinning up and down all the time, you especially don't care about unique information like IP addresses and names. Do cows have names? No. Well, maybe they think they do, but I don't care.

Rock-Solid Infrastructure

This means your infrastructure - DHCP, DNS, and so on - needs to be solid. And those are easy to make solid, especially DNS. And frankly, DNS is *really* the key. With IPv6 in particular, computers are more than capable of making up their own IP addresses, *and that's fine*. This obsessive need to micro-manage IP addresses is outdated and insane. Want to remove a failure point from your environment? Start using IPv6 and ditch DHCP. Let computers make up their own addresses and register them with DNS. That's how we'll find them. Give computers a DSC resource that "registers" each computer's "role" in appropriate groups or pools or whatever, so we can manage them by role.

Yeah, I'm getting a teeny bit ahead of the state of the art, here - but you see where I'm going. We don't need to know more about the details of our IT estate. We need to know *less*, and we need to have processes that work with less detail. Let the computers keep track of the details, and we'll just paint the big picture for them.

Getting Back to DSC

So how does this relate back to DSC? Well, I'd argue that you don't need a Configuration Management Database. A CMDB represents a static view of your environment, and your environment is not static. Instead, you need to set *policies* in a monitoring system. "We need 12 Web servers running this application, they need to be spread between these two data centers, and they each need to conform to the following configuration document. Go." *And that's it.*

Yes, you're going to need tooling. DSC is part of that, in that it gives you a way of creating a configuration document that defines a certain role. But *you* don't worry about *which machines* that role is assigned to. Your monitoring and orchestration system does. "Hey, um, we only have zero of these, and we're supposed to have 12! Go-go-Gadget-deployment!" You define *what's needed*, not how to make it happen.

"But how can I tell which machines are running that role?" Why do you care? "Because if they break I have to fix them!" No, your monitoring system has to notice they are broken, and *kill* them. You

don't fix cows, you eat cows. And then buy new cows. So DSC plays an important role in defining "what a machine should look like for x role." But you don't need a big database keeping track of all those configuration fragments and documents; you just need them in a repository or gallery that your monitoring system can access.

Orchestration and monitoring is where our industry is going. And it isn't so that you can better automate the processes you have. It's so you can *delete* the processes you have, and adopt new processes that require less input, less granular human interaction, and less intervention. The reason people have trouble fitting DSC into their existing processes is because *those processes are outdated*, and DSC (and technologies like it) are trying to elevate the conversation to a different level.

Yes, it's going to take a lot of work and evolution to get to this end-game vision - but this *is* where the industry (not just Microsoft) is going.

The Perfect Example

And if you want a real-live working example, look no further than Amazon Web Services. I use this example a lot because it's a *very good example*, and it proves without a doubt that the vision is right. And this example *is a thing*. It's real, it exists, it's mature, and you can buy it right now. It's a model you can replicate in your own environment. This is not visionary or forward-thinking, it's physical and it's right here, right now.

Amazon already has a front-end load-balancing service. They actually have a few. And they already have a service called Elastic Beanstalk. Here's how it works: you create a simple text document, not unlike a DSC configuration, that describes what a server should look like. You define a Git repository where application code lives, pre-requisite packages the application needs, and all that. In their monitoring-and-load-balancing service - which is both a load balancer and a monitor - you lay out your desired performance characteristics for a server, like how much load a given server should be allowed to handle. *And then you go away.*

Beanstalk spins up a new VM, and configures it according to your document. It gets whatever code it needs, adds the VM to the load balancer, and starts monitoring it. Nobody knows what the machine name is, and nobody cares. Nobody knows what its IP address is, and nobody cares. Oh, sure, the *computers* know, but it's all in their heads. No human knows. After a bit, your server starts to hit max workload. Beanstalk spins up a new one, adds it to the load balancer, and configures it. Nobody had to push a button.

So now you need to make some updates to the environment - some pre-req package has a new version, and you've made some changes to the application code. You update your configuration document, run a command, and Beanstalk spins up *new* VMs that conform to the *new* configuration. *It doesn't try to fix the old VMs!!!!* It just *kills* them, removes them from the load balancer, and replaces them with the new, shiny VMs. Because *nobody cares about old VMs.* They're gone, and the new ones are running.

One day, one of the VMs stops responding correctly. You've configured the monitoring criteria, and perhaps it's failing some HTTP check or whatever. So the system *kills* it, and spins up a new one.

Load balancer updated, nobody's the wiser. And nobody cares about those VMs, because there is *nothing* unique about them that any human being knows. They're cattle.

And you can absolutely run almost every computer in your environment this way. Yes, I know you've got some odd legacy stuff that might not be able to hit this exact model, but you can certainly implement a lot of this model, even with old applications running in VMs. And you can certainly move toward this model going forward. You actually have most of the tools you need, and PowerShell can help you glue them together to achieve exactly this.

And that, ultimately, is what DSC is meant to help build.

LCM and Pull Server Communications

I find that a lot of troubleshooting is easier once you know what's *supposed* to be happening under the hood. So, I set up an unencrypted pull server and a node, and packet-captured their communications. You can do the same in a troubleshooting scenario. Note that I focused on Pull Server Model v2 communications, meaning this isn't applicable to a WMF4 client. Incidentally, you can check out https://github.com/powershellorg/tug, in the References folder, for copies of the WireShark traces I used to figure all this out.

By the way - doing this kind of sniffing is the only reason I will switch a Pull server to HTTP, and I do so only on a closed-off virtual network.

Node Registration

Whenever a node first contacts a pull server, it needs to *register*, using the RegistrationKey data you provide in its LCM configuration. This happens per pull server and per reporting server, even if you have them all running on the same server machine. Like most of these communications, the node will send an HTTP PUT request to the pull server. The URL will be in the format:

```
/PSDSCPullServer.svc/Nodes(AgentId='91E51A37-B59F-11E5-9C04-14109FD663AE')
```

Note that the AgentId is the one configured in the LCM; a brand-new LCM might not show one when you run Get-DSCLCMConfiguration, but after applying a meta-configuration to the LCM, an AgentId will always exist and be visible. The body will contain a JSON (JavaScript Object Notation) object:

```
{ "AgentInformation":
        {
          "LCMVersion": "2.0",
          "NodeName": "hostname",
          "IPAddress": "ip_address(1)"
        },

  "ConfigurationNames":
          [ (array of strings) ],

  "RegistrationInformation":
          {
            "CertificateInformation":
```

```
                    {
                      "FriendlyName": "name",
                      "Issuer": "issuer",
                      "NotAfter": "date",
                      "NotBefore": "date",
                      "Subject": "subject",
                      "PublicKey": "key",
                      "Thumbprint": "thumbprint",
                      "Version": "int"
                    },
          "RegistrationMessageType": "ConfigurationRepository(2)"
        }
}
```

(1) IP Address can be a semicolon-delimited list of IP addresses, including both IPv4 and IPv6.

(2) Will be "ReportServer" when registering with a Reporting Server; will not include ConfigurationNames in that case.

The certificate information included in the above is a self-generated certificate created by the LCM. By providing the pull server with a public key, the pull server can now encrypt things that the LCM, and only the LCM, will be able to decrypt. This certificate information isn't used in the native Windows pull server, but the Azure Automation pull server *does* use it. After initial registration, this certificate is used as a way to authenticate clients. The native pull server stores the information, but doesn't use it.

It's really important to know that this is the one request sent via HTTP PUT, not by POST. The node expects an HTTP 204 reply, with no content in the body of the request. It simply indicates a successful registration; a status of 400 (Bad Request) indicates a malformed request. It's possible for a Web server to block PUT requests, while allowing POST requests. Such a configuration would prevent new registrations.

The LCM actually performs this registration **each and every time** it checks in with the pull server. However, given that the LCM has created an AgentId now stored on the Pull Server (which Get-DSCLocalConfigurationManager will now show), the node inserts the RegistrationKey into the HTTP request *headers*:

```
Authorization: Shared xxxxxxxxx
```

The "xxxxxxxx" is where the authorization information goes - but it isn't just the RegistrationKey that you configured for the node. It consists of:

1. The body of the HTTP request, digitally signed using the node's self-generated private key. Bear in mind that the body of the request contains the public key - the pull server can therefore validate the signature. The current date - included in another HTTP header - is added into the signed information.

2. The signature itself is then run through an HMAC hash process, using the RegistrationKey.

So in order to validate the "Authorization:"" header, the pull server must repeat the signature process, and then create the HMAC using each RegistrationKey that the pull server knows about. If one of the the results matches what the node sent, then it is considered authorized.

This Authorization header is re-sent with *every registration packet*, which happens *every time the node runs a consistency check*. However, remember that the node doesn't actually "remember" that it successfully registered in the past, and the pull server isn't obligated to re-check the Authorization header. My understanding is that, for the native pull server, the Authorization header is ignored if the AgentId is known to the pull server. Given that the AgentId - a GUID - is pretty hard to guess, it seems that on the second-and-subsequent registrations, the AgentId alone is sufficient to re-authorize the node. That isn't the case on Azure Automation's pull server, which as I've mentioned relies on a sort of client certificate authentication, based on the last certificate the node sent.

The node's self-generated certificate is good for one year, is automatically renewed or regenerated, and is sent to the pull server on each registration. *After* authorizing the node, the pull server should take whatever certificate information is in the registration request as canonical, and store it.

Node Check-In

This occurs each time the node runs a consistency check.

The protocol specification states that the server can respond with "Ok" (you're fine, don't do anything); "Retry", "GetConfiguration" (your config is out of date, grab it again), or "UpdateMetaConfiguration." However, that last one, in my tests, generates an error on the LCM, stating that it's not a valid response.

The URL looks like this: `Nodes(AgentId='91E51A37-B59F-11E5-9C04-14109FD663AE')/GetDscAction`

The body of the request will contain a JSON object like this, when the server is configured to use a single configuration (e.g., not partials):

```
{
        "ClientStatus": [
    {
            "Checksum": "checksum",
        "ChecksumAlgorithm": "SHA-256"
    }
    ]
}
```

For nodes configured to use partials, this will look a bit more complex:

```
                {
                    "ClientStatus": [
                        {
                            "Checksum": "checksum",
                            "ConfigurationName": "name",
                            "ChecksumAlgorithm": "SHA-256"
                        },
                        {
                            "Checksum": "checksum",
                            "ConfigurationName": "name",
                            "ChecksumAlgorithm": "SHA-256"
                        }
                    ]
                }
```

The difference is that, with a single configuration, *the server is expected to know what the configuration name is*, because the node supplied it at registration. So the server must store that in its database. For partials, the node must supply the configuration names, since each will have a unique checksum. There's actually intent behind the single-configuration check-in not including the configuration name, and it's so that the server, if it wants to, *can assign a different configuration than the one the client expected*. The native pull server feature in Windows doesn't do this, but Azure Automation's pull server *can*. The idea here is to be able to supply a new, central configuration without having to tell the node. After all, the node shouldn't care - it should regard any MOF on the pull server as "authoritative" and just do what it says.

The server responds with an HTTP 200, and a JSON response:

```
                    {
                        "odata.metadata": "http://server-address:port/api-endpoint/$\
metadata#MSFT.DSCNodeStatus",
                        "NodeStatus": "Ok",
                        "Details": {
                            [
                                {
                                    "ConfigurationName": "config-name",
                                    "Status": "Ok"
                                }
                            ]
                        }
                    }
```

As near as I can tell, only the top-level NodeStatus matters. When testing with partial configurations, Details was often blank.

Requesting a Configuration

Once a node *knows* that it needs a new MOF, it requests one. For nodes using partial configurations, a separate request is sent for each. URLs take this form:

```
/Nodes(AgentId='agent-id-guid')/Configurations(ConfigurationName='desired-name')/Con\
figurationContent
```

There's no special information in the request headers apart from the standard ProtocolVersion (2.0) and Host headers. The body of the request is empty, as all needed information is in the URL. The server responds with 404 for a missing resource, and with a 200 for successful queries. In the event of a successful query, the server response is an application/octet-stream containing the requested MOF. This means the MOF is transmitted as a binary object, rather than as a stream of plain-text.

Requesting a Module

When a node has been configured to pull modules - which it can do from a "normal" pull server, or from a pull server that's only hosting modules - it submits this URL:

```
/Modules(ModuleName='module-name',ModuleVersion='x.y.z')/ModuleContent
```

You'll notice that none of the above includes the AgentId, which might make you wonder if it's possible to simply grab modules from a pull server by guessing module names. It isn't; the AgentId is actually passed in the HTTP headers for these requests:

```
AgentId: xxxxxxxxx
```

The pull server should validate the AgentId and make sure that the request is coming from a known (registered) node. As with a MOF request, the body contains no information, and the server responds with 404 or 200. In the case of a 200, the response is an application/octet-stream with the desired module's ZIP file.

Reporting

When configured to use a Report Server, nodes will routinely - often several times in one consistency check - report in. The URL looks like this:

```
/Nodes(AgentId='agent-id-guid')/SendReport
```

The body of the request is an extensive JSON object containing the entire report for that node. The chapter on Reporting contains more details on what this object contains. Note that it's possible for a report's body to exceed the size that the web server permits for requests; in that case, you need to reconfigure your web server.

One item included in the report is the "CurrentChecksum," which is included for each configuration that the node processes, and matches the contents of the MOF checksum file on the pull server. On its next check-in, this checksum will be sent.

The server responds with a 200 for successful reports, and actually sends a JSON response in the body:

```
                    {
                        "odata.metadata": "http://server-address:port/api-endpoint/$\
metadata#MSFT.DSCNodeStatus",
                        "value": "SavedReport"
                     }
```

So, you know, you at least know the report got through ;). The server can also respond with 400 (bad request) if it didn't accept the report, which usually means it didn't recognize the AgentId as one registered with it.

The Database

The Pull Server's Devices.edb database essentially holds two tables. One, RegistrationData, includes:

- AgentId
- LCMVersion
- NodeName
- IPAddresses
- ConfigurationNames

In other words, about everything the node sent during registration.

And, the StatusReport table, which is the Reporting Server role, you'll find:

- StartTime
- EndTime
- LastModifiedTime
- Id (the AgentId)
- OperationType
- RefreshMode
- LCMVersion

- ReportFormatVersion
- ConfigurationVersion
- IPAddresses
- RebootRequested
- Errors (a text field)
- StatusData (a text field containing JSON data)

Known Problems

Here, I document some of the known problems folks have run across, along with what solutions I know of.

Error configuring the LCM: "Specified Property does not exist," "MI RESULT 12"

This seems to occur on nodes that were upgraded from WMF v4, or from a pre-release of v5. A fix was documented at https://msdn.microsoft.com/en-us/powershell/wmf/limitation_dsc.

Registration Key Problems

When using a node with ConfigurationNames - and therefore, a registration key GUID, you may see failures to register, and log errors on the pull server. That was a known problem with an older version of the xPSDesiredStateConfiguration module (prior to 3.10), especially on Server Core (where you could also see errors related to a missing JET database engine).

maxEnvelopeSize Errors

DSC uses WS-MAN to send data between computers - such as when deploying a MOF in push mode to a node. WS-MAN has some configuration limits (exposed in the node's wsman:/ drive via PowerShell), including maxEnvelopeSize. As a rule, you should have this property set to roughly double the size of the MOF you plan to deploy.

Reporting Server and Large Configurations

The Reporting Server provided by Microsoft has an (undocumented) upper limit on how big of an HTTP request it can receive from nodes. With especially large configurations, it's possible to exceed this (currently unknown) limit. The node will log:

```
Job {%JobID%} :
Http Client {%AgentID%} failed for WebReportManager for configuration The attempt to\
 send status report to the server https://johntestx02/PSDSCPullServer.svc/Nodes(Agen\
tId={%AgentID%}')/SendReport returned unexpected response code RequestEntityTooLarge\
..
```

This has been logged as a bug with the team, at https://windowsserver.uservoice.com/forums/301869-powershell/suggestions/15563172-node-with-large-configurations-fails-to-send-statu

Class-Based Resources Can’t be ExclusiveResources in Partials

This is a known problem and is fixed in WMF5.1 onward.