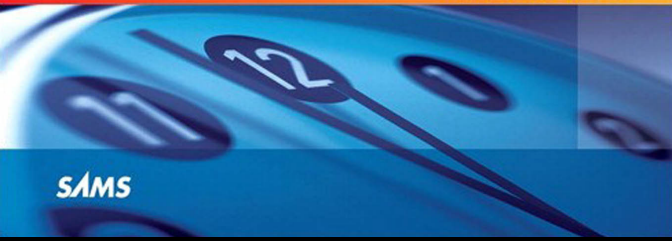
Timothy L. Warner

Sams Teach Yourself

Windows PowerShell®

in 24 Hours



SAMS

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Timothy Warner

Sams **Teach Yourself**

Windows **PowerShell®**

in **24 Hours**

SAMS 800 East 96th Street, Indianapolis, Indiana, 46240 USA

Sams Teach Yourself Windows PowerShell® in 24 Hours

ISBN-13: 9780672337284

ISBN-10: 0672337282

Library of Congress Control Number: 2015900973

Printed in the United States of America

First Printing May 2015

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Special Sales

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact international@pearsoned.com.

Editor-in-Chief
Greg Wiegand

Acquisitions Editor
Joan Murray

Development Editor
Sondra Scott

Managing Editor
Kristy Hart

Project Editor
Andy Beaster

Copy Editor
Keith Cline

Indexer
Cheryl Lenser

Proofreader
Katie Matejka

Technical Editor
Jeff Wouters

Publishing Coordinator
Cindy Teeters

Cover Designer
Mark Shirar

Compositor
Gloria Schurick

Contents at a Glance

Table of Contents

Part IV: Managing Computers Remotely with Windows PowerShell

Part V: Putting Windows Powershell to Work

About the Author

Timothy Warner is an IT professional and technical trainer based in Nashville, Tennessee. Tim became acquainted with information technology in 1982 when his dad bought the family a Timex Sinclair 1000 home computer and he taught himself BASIC programming. Today he works as an author/evangelist for Pluralsight and shares Windows PowerShell knowledge with anyone who'll listen at his Two Minute PowerShell blog: http://2minutepowershell.com. You can reach Tim directly via LinkedIn: http://linkedin.com/in/timothywarner.

Dedication

To all my students, past and present.
Thank you for giving me a professional calling,
and I hope that my work helps you attain your goals.

Acknowledgments

The Windows PowerShell community is a terrific group of people. Thank you, Jeffrey Snover, Bruce Payette, and Lee Holmes et al. for giving the world Windows PowerShell. Thanks to all the PowerShell experts in the world for being so kind and willing to share your knowledge. I seek to emulate your actions every day.

It may take a village to raise a child, but I know that it takes a large office full of talented professionals to publish a book. To that end, I want to thank my wonderful editor Joan Murray for having faith in my abilities. Thanks to my publisher, Greg Wiegand, for being so receptive to my ideas.

Editorial and production staff rarely receive the credit they deserve. Thanks so much to Windows PowerShell MVP Jeff Wouters, my technical editor, for being so thorough with the manuscript. Truly, this book is at least twice as good as it originally was thanks to you.

Thanks to Keith Cline, my copyeditor, for making my writing easier to follow. Keith knows that I gave his editing skills quite a workout, for sure. Sorry, Keith!

I extend my gratitude as well to Andy Beaster, my production editor, and to the ever-helpful Cindy Teeters for streamlining the entire book publishing process. Andy and Cindy are professionals in every sense of the word.

Thanks to my family, friends, and colleagues for your never-ending love and support. Finally, thank you, my reader. I hope that this book helps you accomplish your next professional goal.

We Want to Hear from You!

As the reader of this book, you are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

We welcome your comments. You can email or write to let us know what you did or didn't like about this book—as well as what we can do to make our books better.

Please note that we cannot help you with technical problems related to the topic of this book.

When you write, please be sure to include this book's title and author as well as your name and email address. We will carefully review your comments and share them with the author and editors who worked on the book.

Email: consumer@samspublishing.com

Mail: Sams Publishing
ATTN: Reader Feedback
800 East 96th Street
Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at informit.com/register for convenient access to any updates, downloads, or errata that might be available for this book.

This page intentionally left blank

Introduction

"Try not. Do...or do not. There is no try"

—Yoda, *Star Wars Episode V: The Empire Strikes Back*

Hello, and welcome to the world of Windows PowerShell. I'm your instructor, Tim Warner. To me, it's a good sign that you're actually reading this Introduction (so few readers of tech books do, in my experience). Perhaps your first question is, "What's in it for me?" and I'm here to give you those details with minimal muss and fuss.

If you work as a Windows systems administrator or hope to in the future, learning Windows PowerShell is no longer an option. Likewise, if you plan to advance your career in IT administration, you need to know your way around Windows PowerShell scripting and automation. This, then, is what's in it for you: By learning how to harness Windows PowerShell, you make yourself a more effective and valuable Windows systems administrator. And if you have value in the IT workplace, you have a means of having a stable, lucrative career and an equally stable and lucrative life.

Who Should Read This Book

The first thing I do when I teach "stand up" training classes is to get a feel for my student. What is your background? What do you hope to get out of this training? As I wrote this book, I had the following audiences in the forefront of my mind:

- **Microsoft certification candidates**: I'm here to tell you that if you don't understand Windows PowerShell, you have a high likelihood of failing your Microsoft Certified Professional (MCP) exams. And I'm not just talking about Windows Server 2012 R2 certification, either. Microsoft Learning stresses PowerShell-based administration in all of their products nowadays, so you simply cannot escape the technology, no matter how hard you might try.
- **Windows systems administrators**: I'm sure that you've been aware of Windows PowerShell over the past several years, and maybe you've been avoiding learning the technology because the tech appeared too "programmy" or math heavy. Let me assure you that by the time you complete this book, you won't be afraid of that anymore because you'll be convinced how much easier PowerShell makes your life as a "boots on the ground" sysadmin.

- **IT newcomers**: If you are working on a transition into full-time IT work, whether you're entering IT from an unrelated field or preparing to graduate from trade school or college, then welcome! You have an advantage in learning Windows PowerShell at the outset of your IT career because you'll be able to seamlessly integrate PowerShell automation into your vision of IT.

If you find that you don't belong in any of the previous three classifications, don't worry about it. Set your sights on learning as much as you can and, above all else, having fun, and you'll be fine.

How This Book Is Organized

These "24 Hour" books begin with the premise that you can learn a technology (Windows PowerShell, in this case) by studying the material in 24 one-hour sessions. Maybe you can use your lunch break as your training hour; then again, the hour after your children finally fall asleep at night might work better.

In any event, allow me to present hour-by-hour details on how I structured the content:

- Hour 1, "Getting to Know Windows PowerShell," makes the case that knowing Windows PowerShell is mandatory and not optional for Windows systems administrators. You'll also learn how PowerShell works from an architectural/design standpoint.
- In Hour 2, "Installing and Configuring Windows PowerShell," you understand the Windows PowerShell release cycle, backward-compatibility basics, and how to upgrade your installed Windows PowerShell version.
- In Hour 3, "Mastering the Windows PowerShell Help System," you learn how to learn Windows PowerShell. Believe me, this chapter is one of the three most important chapters of the book because you'll use the help system every day.
- In Hour 4, "Finding and Discovering Windows PowerShell Commands," you master the **Get-Command** cmdlet. This is the second of the three most important chapters, again based on how often you'll use these skills.
- In Hour 5, "Thinking in Terms of Objects," you use **Get-Member** to list the methods and properties of PowerShell objects. This hour completes the "triad" of three core chapters that comprise your foundational understanding of Windows PowerShell.
- In Hour 6, "Mastering the Windows PowerShell Pipeline," you begin to understand that in Windows PowerShell, you're always working from within a command pipeline, and you also recognize that in a PowerShell pipeline, you're almost always dealing with objects.

- In Hour 7, "Sorting, Filtering, and Measuring Windows PowerShell Output," you learn how to cut down your output to separate only the data you need.
- In Hour 8, "Managing Windows PowerShell Providers," you learn how you can access and browse various data stores, from environment variables and the Registry to the certificate store and Active Directory, in the same way that you browse your file system from the command line.
- In Hour 9, "Formatting, Exporting, and Converting Windows PowerShell Output," you pick up some valuable skills on creating submission-quality output of your PowerShell pipelines.
- In Hour 10, "Implementing One-to-One Windows PowerShell Remoting," you get a grip on the wonderful remoting architecture in Windows PowerShell. Here we examine how to set up remoting and establish remote sessions with other Windows computers on our network.
- In Hour 11, "Implementing One-to-Many Remoting," you learn how to send PowerShell commands and even entire scripts to an unlimited amount of target computers in parallel. This chapter demonstrates the raw power you have at your fingertips when you use PowerShell to manage your Windows networks.
- In Hour 12, "Deploying PowerShell Web Access," you learn how to set up PowerShell to be accessed from any remote device—even mobile phones and tablets from outside your corporate firewall. This is a cool technology, for sure.
- In Hour 13, "Multitasking Windows PowerShell," you discover the Windows PowerShell jobs architecture, in which you can send simple or complex PowerShell operations to the background of your session. By mastering jobs, you (and PowerShell) can multitask with aplomb.
- In Hour 14, "Harnessing Windows PowerShell Workflow," you take the next step with PowerShell jobs and learn how to design and deploy durable PowerShell tasks that respond to state changes, such as system reboots. Very cool stuff here.
- In Hour 15, "Introducing WMI and CIM," you finally come to terms with two acronyms many Windows systems administrators hear all the time but rarely understand: Windows Management Instrumentation (WMI) and Common Information Model (CIM). By the end of this chapter, you'll be crystal-clear on how to fetch system state data from the WMI repository by using PowerShell code.
- In Hour 16, "Searching and Filtering with Regular Expressions," you put your string searches on steroids by learning how to use the .NET Framework's regular expression syntax to perform highly specific find and replace operations on your data—all with PowerShell code.

- In Hour 17, "Installing and Managing Software with OneGet," you learn how to install and manage software, all from the PowerShell console command line. If you've used command-line software package management in Linux or OS X, what you learn during this hour will be immediately familiar.
- In Hour 18, "Desired State Configuration Basics," you learn what will doubtless become the next generation of Windows Server systems configuration: Desired State Configuration, or DSC, in Windows PowerShell.
- In Hour 19, "Introduction to Windows PowerShell Scripting," you take everything you've learned over the previous 18 hours of training and apply that knowledge toward code reuse. In other words, you'll learn the basics of writing, configuration, and running Windows PowerShell script files.
- In Hour 20, "Making PowerShell Code Portable with Modules," you build upon what you learned in the preceding hour of training concerning PowerShell scripts and target that knowledge toward packing your code into modular...well, modules.
- In Hour 21, "Managing Active Directory with Windows PowerShell," you embark on a four-hour journey of PowerShell domain-specific management. Here the "domain" is Active Directory Domain Services (AD DS) itself.
- In Hour 22, "Managing SQL Server with Windows PowerShell," you learn how to use the SQL Server PowerShell module and SQL Management Object (SMO) to interact with SQL Server databases and objects through PowerShell code.
- In Hour 23, "Managing SharePoint Server with Windows PowerShell," you learn how to create SharePoint farm objects (web application, site collection, list, and so forth) by using the SharePoint Server 2013 PowerShell snap-in.
- In Hour 24, "Managing Microsoft Azure with Windows PowerShell," we complete the training by applying Windows PowerShell to Microsoft's public cloud service: Azure.

Conventions Used in This Book

In my experience as an author and a teacher, I've found that many readers and students skip over this part of the book. Congratulations for reading it. Doing so will pay off in big dividends because you'll understand how and why we formatted this book the way that we did.

Try It Yourself

Throughout the book, you'll find Try It Yourself exercises, which are opportunities for you to apply what you're learning right then and there in the book. I do believe in knowledge stacking,

so you can expect that later Try It Yourself exercises assume that you know how to do stuff that you did in previous Try It Yourself exercises.

Therefore, your best bet is to read each chapter in sequence and work through every Try It Yourself exercise.

About the Bitly Hyperlinks

Whenever I want to point you to an Internet resource to broaden and deepen the content you're learning, I provide a uniform resource locator (URL, also called an Internet address) in the following form:

```
http://bit.ly/uaKpYD
```

You might wonder what the heck this is. The way I look at the situation, if I were reading this title as a print book and needed to type out a URL given to me by the author, I would rather type in a "shortie" URL than some long, crazy URL with all sorts of special characters, you know what I mean?

The most important thing I have to tell you concerning the bitly short URLs is that the ending part is case sensitive. Therefore, typing the previous URL as, say, http://bit.ly/UaKpyD isn't going to get you to the same page as what I intended.

About the Code Images

For most Try It Yourself exercises, you'll see one or more source code images that are annotated with alphabetic letters. The Try It Yourself steps are then cross-referenced with parts of each code image. Hopefully, you find this format convenient to your learning. Remember not to fall into the trap of blindly copying the provided code; instead, remember that learning to program requires (yes, requires) lots and lots of trial and error.

That actually is a point well worth repeating: To become effective with Windows PowerShell, you need to use it daily. Don't complain about retyping my code examples. Instead, look at it as an opportunity for you to practice.

System Requirements

You don't need a heck of a lot, computer-wise, to perform all the Try It Yourself exercises in this book. However, if you do not meet the necessary system requirements, you are stuck. To that end, make sure that you have the following met prior to beginning your work:

- **A Windows-based computer**: Technically, you don't need a computer that runs only Microsoft Windows. For instance, I use VMware Fusion to run Windows 8 virtual machines (VMs) on my OS X computer. No matter how you slice it, though, Windows PowerShell has *Windows* in its name for a reason, so you'll be stuck at the starting gate unless you have a Windows machine at your disposal.

- **An Internet connection**: In learning Windows PowerShell, you'll be hitting the Web all the time to gain additional insight, obtain code examples, and so forth. Moreover, because Windows PowerShell doesn't ship with local help files, you'll need an Internet link to download those at least once.
- **A VM network and an Azure subscription**: You can build a two- or three-node practice network for free. Windows 8.1, for instance, includes Hyper-V. You can also download Oracle VM VirtualBox to deploy a VM-based network. Microsoft is kind enough to offer full-feature evaluation editions of their software, so you shouldn't have to pay big bucks for licenses. Along those lines, Microsoft offers trial subscriptions of their Microsoft Azure subscription service. As I wrote this book, I made sure that replicating my network environment was as painless as possible for you because I want you to work through every single example in the book to maximize your learning.

Design Elements Used in This Book

Some code statements presented in this book are too long to appear on a single line. In these cases, a line-continuation character (➥) is used to indicate that the following line is a continuation of the current statement.

NOTE

Items of Interest

Notes offer interesting information related to the current topic.

TIP

Useful Tidbits

Tips offer advice or show you an easier way to perform a task.

CAUTION

Potential Pitfalls

Cautions alert you to a possible problem and suggest ways to avoid it.

Okay, that's enough of the preliminaries. It's time to learn how to use Windows PowerShell.

HOUR 18

Desired State Configuration Basics

What You'll Learn in This Hour:

- Historical background of DSC
- Basic tenets of DSC
- DSC authoring environment
- Configuring the DSC environment
- Writing your first configuration script
- A word on DSC push configuration

Desired State Configuration, also called DSC, is the marquee feature in Windows PowerShell v4 and later. Imagine being able to send configuration instructions to your servers such that, with no tedious mouse clicking on your part, the target servers simply (to quote Jean-Luc Picard from *Star Trek: The Next Generation*) "Make it so."

I'm not kidding, either. In this hour, you'll learn precisely what DSC is and how it works, and you'll see its value proposition with your own eyes. Many of my IT professional colleagues whisper that DSC may very well spell the future standard for Windows server configuration and administration. Let's make it so!

Historical Background of DSC

Windows PowerShell Principal Architect Jeffrey Snover wrote *The Monad Manifesto* in 2002, and in so doing outlined what he saw as the chief capabilities of a new command-line automation language for Windows.

Amazingly, Snover and his team at Microsoft realized every major point in that document. Specifically, with the Manifesto's fourth point, Monad Management Models, they describe the basic elements that the team ultimately delivered in Windows PowerShell v4.

DSC is a Windows PowerShell-based system configuration platform. Here's the scenario: You and/or your colleagues spend valuable hours manually configuring your Windows servers; I'm talking about tasks such as the following:

- Installing and configuring roles and features
- Installing and configuring other system software and services
- Deploying and maintaining file shares
- Managing Registry settings and environment variables

The preceding list barely scratches the surface of the myriad configuration events that must be performed on each server for that machine to be considered compliant by your organization.

However, if you have any degree of Windows systems administration experience, you know that "configuration drift" is a sad fact of life. Joe Administrator makes one setting, and then a week later Jane Administrator undoes said setting.

The configuration drift problem is all fun and games until questions of service level agreements (SLAs), licensure requirements, and industry/governmental regulations come knocking at your door, metaphorically speaking.

Long story short: DSC fills a need for us Windows server administrators, now more than ever before.

Competitive Landscape

Remember that Jeffrey Snover and the Windows PowerShell team are almost all longstanding experts in UNIX/Linux administration and systems programming. This fact should be patently obvious when you compare, say, the day-to-day operation of the Bash shell with how the Windows PowerShell command-line environment behaves.

To that point, there's no denying the fact that Snover & Co. took a leaf from the competition's playbooks with regard to this automated systems configuration framework "thing." Specifically, two market leaders in the systems configuration/automation space are also (partially) open source projects:

- Chef (http://www.chef.io)
- Puppet (http://puppetlabs.com)

Don't get too bent out of shape, though: Not only are Chef and Puppet compatible with Windows, but Microsoft Azure offers either configuration product as an option for their hosted virtual machines. Figure 18.1 shows a representative screenshot of Puppet.

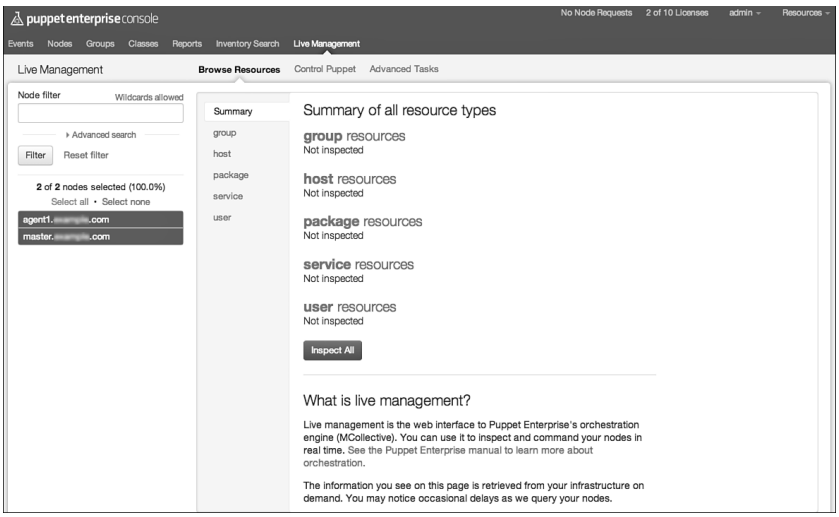


FIGURE 18.1
Puppet has a browser-based management console that makes it equally simple to autoconfigure Windows/Linux servers.

Going further yet, we'll learn shortly that DSC can actually be extended to support the autoconfiguration and remediation of Linux/UNIX boxes in addition to Windows machines. It's a "New Microsoft," to be sure.

Finally, as cool as Puppet and Chef are as cross-platform configuration management products, they cost money to license for most business scenarios. By contrast, Windows PowerShell comes to you "free" with the cost of a Windows Server license.

One final note before we delve into DSC: Although it's possible to leverage DSC for desktop OS configuration, I'm cleaving to the most common DSC use case: server configuration. After all, most of our compliance requirements focus on how we've set up our infrastructure server computers as opposed to our users' desktop PCs.

Basic Tenets of DSC

To begin, you should understand that most DSC configuration involves using Windows PowerShell and the vendor-neutral Managed Object Format (MOF) in a declarative fashion. In programming, declarative code does not spell out exactly how the computer should complete a task. Instead, the code essentially tells the computer to "make it so" however it sees fit.

Structured Query Language (SQL) is a good example of a declarative data access language. When you run a complex **SELECT** statement, for instance, you leave it to the database itself to determine the system of index/row lookups it uses to satisfy the query results.

Likewise, in DSC, we start by describing how we'd like our servers to look in a standard Windows PowerShell configuration script. Take a look at Figure 18.2, and I'll explain how DSC works step by step.

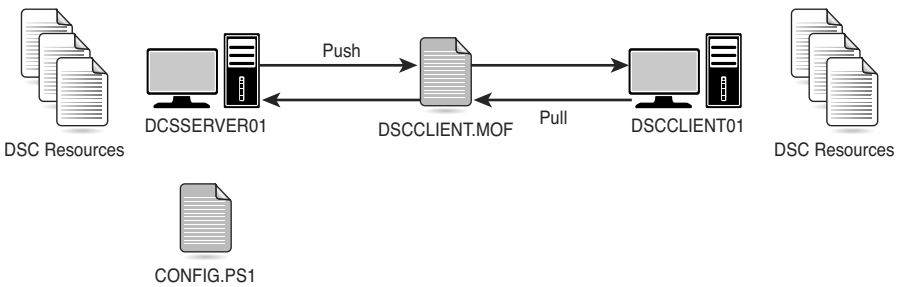


FIGURE 18.2
Windows PowerShell DSC architectural overview.

DSC Authoring Environment

As you saw in Figure 18.2, DSCSERVER01 represents our DSC authoring environment. It is on this box, which must be equipped with at least Windows Management Framework v4 or later, that we construct our configuration script.

The configuration script is a bread-and-butter Windows PowerShell file that contains the configuration instructions for one or several target systems. The configuration script is compiled into the vendor-neutral MOF and then transferred to the target systems for ingestion.

DSC Production Environment

A component of Windows Management Framework (WMF) 5 called the Local Configuration Manager (LCM) running on the target system is what receives the MOF and applies its configuration settings to the box.

DSC supports two modes for getting the MOF configuration file to the target computer. In the push model, we use the **Start-DSCConfiguration** cmdlet to initiate the MOF push.

In the pull model, the client computer polls an Internet Information Services (IIS) website running on your DSC deployment server and requests any MOF files that are specified for it.

In terms of query intervals, target nodes query the pull server every 30 minutes by default. In the push architecture, nodes reevaluate their MOF file settings every 15 minutes by default if the configuration file had autocorrection enabled. As with anything else in Windows PowerShell, you can edit those query defaults.

Finally, as you observed in Figure 18.2, something called "DSC resources" exist on both the authoring and production servers. We can consider DSC resources to be specialized Windows PowerShell modules that actually form the imperative "engine" that nodes use through their LCM to apply their desired state configurations.

Differences Between DSC and Group Policy

Some Windows systems administrators wonder, "What's the difference between DSC and Group Policy?" One difference is that DSC permanently "tattoos" the configuration settings of target nodes. You'll recall that once a Group Policy Object (GPO) no longer applies to a machine, those settings can revert to their pre-GPO values.

Another difference is that a single node can have only a single MOF file defining a particular configuration (installing and configuring IIS, for instance). By contrast, we can link multiple GPOs to each of the various Active Directory levels (site, domain, organizational unit, and local computer). Finally, GPOs grant management access principally to the computer's registry, while DSC MOF resources can "touch" any computer subsystem that's accessible by PowerShell and, by extension, the .NET Framework.

The bottom line is that DSC won't necessarily replace GPOs for systems configuration. Remember the focus with DSC, at least at this point, is to declaratively configure our servers such that "configuration drift" and deviation from compliance is no longer an issue for us.

Before we can test out DSC, we need to first prepare our environment.

Configuring the DSC Environment

Don't even think about testing, much less deploying, DSC unless all of the following are true:

- All participating computers have WMF 4.0 or later installed.
- All participating servers have Windows PowerShell remoting enabled.
- All Windows Server 2012 R2 and Windows 8.1 nodes have hotfix KB2883200 installed.

You can leverage Windows PowerShell to verify if that required hotfix has been applied to your system:

```
PS C:\> Get-HotFix -Id KB2883200

Source        Description      HotFixID      InstalledBy         InstalledOn
------        -----------      --------      -----------         -----------
DSCSERVER01   Update           KB2883200     COMPANY\trainer     9/30/2013
12:00:00AM
```

Windows PowerShell remoting is required because the deployment of DSC MOF files uses Web Services-Management / Windows Remote Management (WS-Man/WinRM).

A New Microsoft

In past years, Microsoft took a highly proprietary approach to how their own products interoperated (or didn't) with those of other vendors, especially open source community projects.

Jeffrey Snover went to great lengths to establish Microsoft corporate buy-in for interoperability, and this argument has paid huge dividends with cross-platform capabilities such as WS-Man, Windows Management Instrumentation / Common Information Model (WMI/CIM), and the MOF format. The idea that today we can use DSC to configure Linux computer was utterly inconceivable not too long ago.

You also need to enable the DSC bits on all participating nodes. (*Nodes* is a more descriptive term than *server* because technically DSC can be used in both server and desktop Windows versions.) From an elevated Windows PowerShell console prompt on a Windows Server box, you can run the following:

```
Install-WindowsFeature
```

Of course, we can also use Server Manager (on servers) or Windows Features (on clients) to enable DSC, as shown in Figure 18.3.

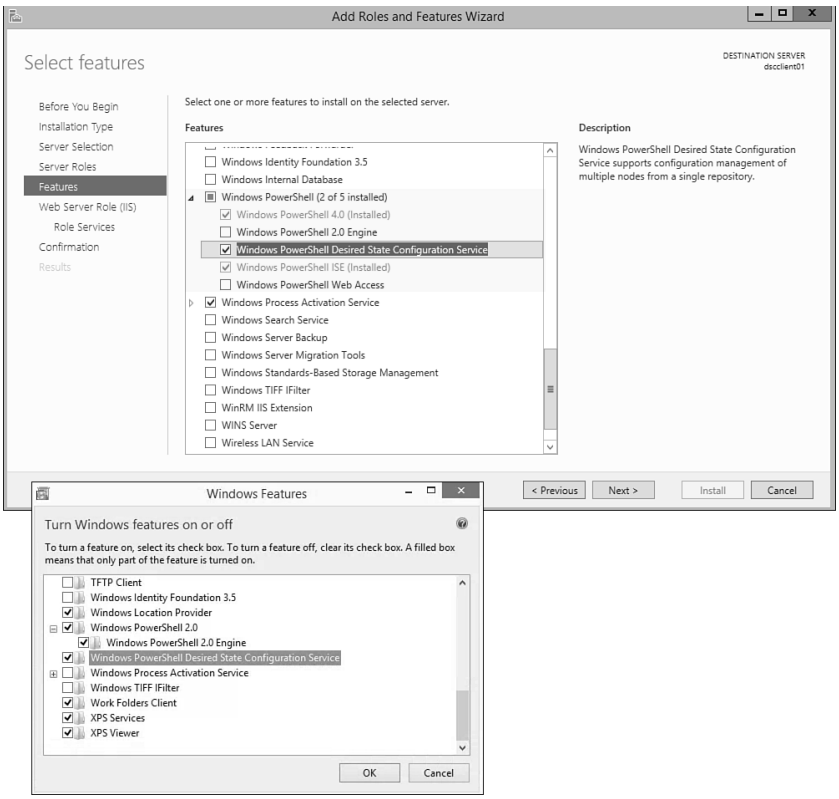


FIGURE 18.3
Here we enable Windows PowerShell DSC in Windows Server 2012 R2 (top) and in Windows 8.1 (bottom).

Loading Up DSC Resources

As of this writing, Microsoft gives us 12 in-box resources in WMF v4. These resources and their uses are as follows:

- **Archive**: Zipping and unzipping archives
- **Environment**: Managing environment variables
- **Group**: Managing local groups
- **Log**: Writes messages to the Microsoft-Windows-DSC/Analytic event log
- **Package**: Installs .msi or Setup.exe software
- **Registry**: Managing the computer and user Registry hives
- **Script**: Excellent as a "catchall" resource when you can't get what you need from an existing DSC resource

- **File**: Managing files and folders
- **WindowsProcess**: Controlling process objects
- **WindowsFeature**: Managing server roles and features
- **Service**: Controlling service objects
- **User**: Managing local users

If you run the following command:

```
Get-DSCResource | Select-Object { $_.parentpath }
```

you'll see that your built-in DSC resource folders are placed deep in the Windows\system32 hierarchy:

```
C:\Windows\System32\WindowsPowerShell\v1.0\Modules\<modulename>
```

That's all well and good, but when you need to install your own modules, you should place them in this path:

```
C:\Program Files\WindowsPowerShell\Modules
```

Specifically, you should place the unzipped resource folder directly inside Modules. For instance, in Figure 18.4, I show you where I placed the **xActiveDirectory** experimental module that I downloaded to my server via **OneGet**.

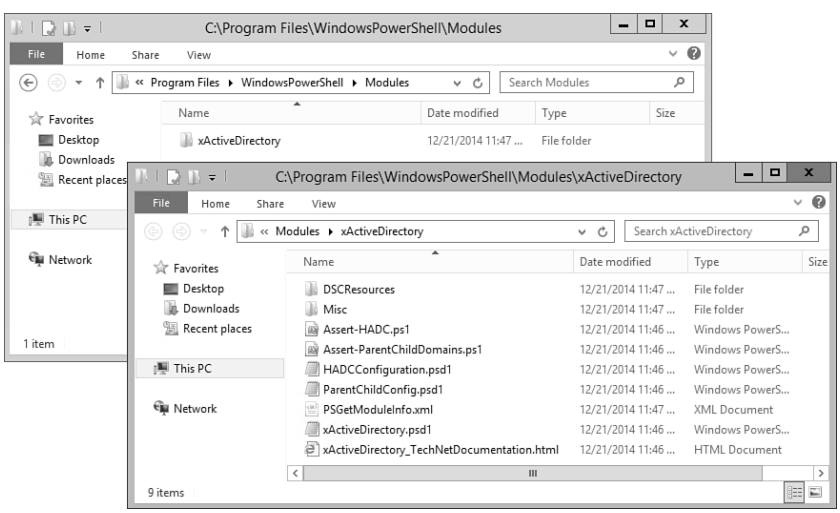


FIGURE 18.4
Here we see where to place additional DSC resources on a node's file system. Notice that a DSC resource looks and "feels" an awful lot like a traditional Windows PowerShell script module.

If your nodes are equipped with PowerShell v5 preview (which they shouldn't unless v5 has been finalized as of your reading this), I suggest you look for DSC resources by querying the repositories:

```
Get-Package -Name x*
```

The *x* prefix is used to denote prerelease or e**X**perimental resource packages. Therefore, you use them in production at your own risk.

DSC Resource Waves

Aside from **OneGet** repos, your best bet for discovering useful DSC resources are the DSC Resource Kit "waves" that are regularly released by the Windows PowerShell team. Each wave brings new resources to the table that allow you greater administrative control over more and more products. Sometimes you'll find that a newer wave release includes updated resources that supersede previously released versions. (The *x* in *experimental* is taken very seriously by the PowerShell community.)

Sadly, the DSC resource kit waves aren't presented in a strictly linear fashion, which can make it tricky figuring out what's what. To help you along, I'll pass on the links for the nine wave announcements that are extant as of this writing:

- **DSC ResKit Wave 1**: http://bit.ly/1wAZpXb
- **DSC ResKit Wave 2**: http://bit.ly/1wAZr15
- **DSC ResKit Wave 3**: http://bit.ly/1wAZpX0
- **DSC ResKit Wave 4**: http://bit.ly/1wAZolQ
- **DSC ResKit Wave 5**: http://bit.ly/1wAZmuq
- **DSC ResKit Wave 6**: http://bit.ly/1wAZnOU
- **DSC ResKit Wave 7**: http://bit.ly/1wAZnhQ
- **DSC ResKit Wave 8**: http://bit.ly/1wAZieb
- **DSC ResKit Wave 9**: http://bit.ly/1wAZnyi

Again, you simply download the resources, unzip them into the proper directory, and run **Get-DSCResource** to verify that they show up. Recall also that you need to install the resources on all participating nodes.

Writing Your First Configuration Script

Okay, it's time to start building out our DSC infrastructure, the first step of which is authoring our configuration script. Remember that although target nodes can apply only one MOF file for a given configuration, you can apply multiple MOFs to a single host as long as you don't have conflicting configuration definitions. I'm sure that, over time, the Windows PowerShell team will make it easier for administrators to manage these manifold MOF manifests (alliteration alert).

I want you to understand before we get started that creating the MOF files via a PowerShell configuration script represents only one possibility for creating the MOFs. If, perchance, you understood MOF syntax, there's nothing stopping you from creating your own MOFs from scratch using only a text editor.

In other words, we should start to see MOF authoring tools emerge from independent software vendors (ISVs) and the community at large as we progress over time. Welcome to the world of vendor neutrality and community-driven software architectures.

More About MOF Files

Remember that the MOF is not a Microsoft proprietary format, but instead is a vendor-neutral data representation format developed by the Distributed Management Task Force, of which Microsoft is a member.

MOF is used to define both management objects in CIM/WMI, and is also closely related to Web-Based Enterprise Management (WBEM) protocols such as WS-Man.

Figure 18.5 shows you what a typical MOF file looks like. I can't stress enough that DSC is a potentially vendor-neutral technology, and the tool that you use to create the MOF doesn't have to be Windows PowerShell. I submit that we'll see graphical user interface (GUI) MOF creation utilities for DSC not too long in the future; perhaps these tools already exist by the time you're reading this book.

```
1  Configuration SampleConfig1
2  {
3      Node "dscclient01"
4      {
5          File CopyScript
6          {
7              Ensure = "Present"
8              Type = "Directory"
9              SourcePath = "\\dscserver01\scripts"
10             DestinationPath = "C:\scripts"
11         }
12
13     }
14 }
15 SampleConfig1
```

FIGURE 18.5
A MOF file can be created by using PowerShell, another utility or programming language, or from scratch. As long as the MOF uses legal syntax, the method by which you produce the file is irrelevant.

Spend a moment studying the configuration script code in Figure 18.6, and I'll walk you through each line. I strongly suggest you write your DSC configuration script in the Windows PowerShell integrated scripting environment (ISE) so that you can take advantage of IntelliSense and the easy script execution controls.

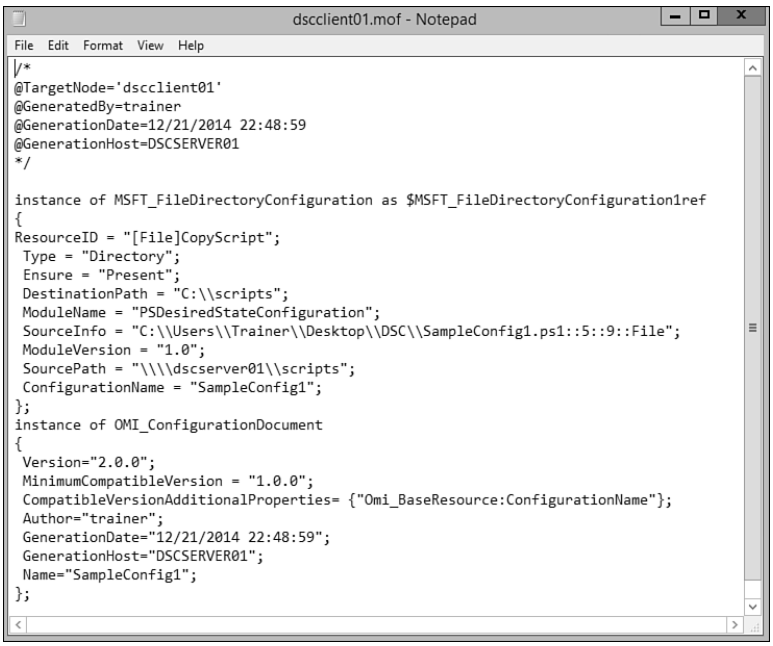


FIGURE 18.6
This DSC configuration script will produce one MOF file that is named after the specified target node.

- **Line 1**: We use the **Configuration** keyword in our script to denote a DSC configuration file. The configuration name is arbitrary.
- **Line 2**: The **Configuration** element is enclosed in top-level curly braces.
- **Line 3**: The **Node** keyword specifies the target node. In this first example, we're hard-coding the name of a Windows Server 2012 R2 host named dscclient01. In a later example, we'll parameterize this element with a variable so we can use one config script to target multiple nodes.
- **Line 4**: Indenting curly braces is optional, but an excellent practice to minimize the chance of our forgetting to close a script block and generate a runtime script failure.
- **Line 5**: The "meat and potatoes" of the configuration script are these subblocks. Here we specify the File DSC resource type, passing in an arbitrary name.

- **Line 6**: Another indented curly brace, this time enclosing the File resource script block.
- **Line 7**: Each DSC resource contains a number of named parameters. Like anything else in PowerShell, read the resource's online documentation to learn the acceptable values for each parameter. The **Ensure="Present"** line is ubiquitous in DSC configuration scripts, in my experience. This ensures that the policy is enforced.
- **Lines 8-10**: Here we plug in the details for our File DSC resource declarations. What we're doing is copying the scripts shared folder on my deployment server to a local path on the target node. As of this writing, I needed to add the source and destination node computer accounts to the shared folder's discretionary access control list (DACL) to make a UNC path work.
- **Lines 11–14**: Here we close up all the script blocks.
- **Line 15**. This is an optional line in which we call the configuration. That way we actually execute the Configuration block when we run the script in the Windows PowerShell ISE.

When you're ready, run the entire configuration script. If all goes well, you'll see output that is similar to this:

```
PS C:\Users\Trainer> C:\Users\Trainer\Desktop\DSC\SampleConfig1.ps1

    Directory: C:\Users\Trainer\SampleConfig1

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----              12/22/2014   8:23 AM  1736 dscclient01.mof
```

You'll note that Windows PowerShell does a couple things when you run the DSC configuration script:

- Creates a directory in the root of the C: drive with the same name as the .ps1 script file
- Creates one MOF for each node referenced in the script; the MOF files are named with the node's hostname

Customizing the Local Configuration Manager

Earlier in this hour, I told you that all DSC-enabled Windows nodes have a client component called the Local Configuration Manager (LCM) that is installed as part of WMF v4.

We can (and probably should) push a separate configuration script to our target nodes to customize the deployment parameters. First, let's run **Get-DscLocalConfigurationManager** to see what's what on my dscserver01 machine:

```
PS C:\> Get-DscLocalConfigurationManager

ActionAfterReboot              : ContinueConfiguration
AllowModuleOverWrite           : False
CertificateID                  :
ConfigurationDownloadManagers  : {}
ConfigurationID                :
ConfigurationMode              : ApplyAndMonitor
ConfigurationModeFrequencyMins : 15
Credential                     :
DebugMode                      : False
DownloadManagerCustomData      :
DownloadManagerName            :
LCMCompatibleVersions          : {1.0, 2.0}
LCMState                       : Ready
LCMVersion                     : 2.0
MaxPendingConfigRetryCount     :
StatusRetentionTimeInDays      : 7
PartialConfigurations          : {}
RebootNodeIfNeeded             : False
RefreshFrequencyMins           : 30
RefreshMode                    : PUSH
ReportManagers                 : {}
ResourceModuleManagers         : {}
PSComputerName                 :
```

Some of the LCM parameters are more important than others. The **ConfigurationMode** parameter tells the node what to do in terms of how it applies and refreshes DSC configurations. The options here are as follows:

- **Apply**: Applies the configuration once and then doesn't check for an update or refresh again (one-time application, in other words).
- **ApplyAndMonitor**: Applies the configuration and continues to validate that the node is in compliance with the policy. If configuration drift occurs, the node does nothing.
- **ApplyAndAutoCorrect**: Applies the configuration, periodically checks for compliance, and reapplies the configuration if something changes within the scope of active configurations.

Note also the **RefreshFrequencyMins** parameter. In push mode, the node checks for DSC compliance every 30 minutes. This may be far too frequent for your business needs, so let's deploy a new set of LCM settings to our localhost and dscclient01 nodes.

Once again, take a look at our script shown in Figure 18.7, and I'll walk you through selected parts:

```
1  Configuration SetupLCM
2 {
3     param
4     (
5        [string[]]$NodeName = $env:computername
6     )
7
8     LocalConfigurationManager
9     {
10        ConfigurationModeFrequencyMins = 240
11        ConfigurationMode = 'ApplyAndAutoCorrect'
12        RebootNodeIfNeeded = 'True'
13        RefreshMode = 'PUSH'
14    }
15 }
16
17 SetupLCM -NodeName ("dscserver01","dscclient01")
18
```

FIGURE 18.7
By deploying an LCM configuration, we can take fine-grained control over how DSC policies are evaluated and applied by target nodes.

- **Lines 3–5**: These lines create an input parameter for our LCM script. Note that the **$NodeName** parameter is defined as a string array, [], which makes it a snap to target multiple nodes without having to repeat code blocks in the script file.
- **Line 10**: Here we specify a 4-hour refresh interval for the LCM policy refresh mode.
- **Line 17**: Again for convenience, we run the configuration in-line with the code, specifying two target nodes by hostname. Of course, you can import the script into your runspace by using dot sourcing. However, I like the convenience of calling the function directly in the script file. Your mileage may vary, as I've said in this book about a hundred times before.

▶ An exhaustive discussion of how to import scripts into your runspace using dot sourcing is included in Hour 19, "Introduction to Windows PowerShell Scripting."

When you run the LCM config script, you wind up with a single "meta" MOF regardless of how many nodes you target in the script. Likewise, we use a different cmdlet to apply an LCM script: **Set-DscLocalConfigurationManager**. The **-Path** parameter points to the directory that contains our LCM script:

```
Set-DscLocalConfigurationManager -Path "C:\SetupLCM"
```

Let's do a Try It Yourself exercise so that you can shore up your Windows PowerShell skills and see how DSC works with your own eyes.

TRY IT YOURSELF ▼

Creating and Pushing a DSC Configuration

In this Try It Yourself exercise, you'll apply much of the PowerShell skills you've accrued throughout the book to apply a specific configuration to a target node.

Specifically, you'll configure a Windows Server 2012 R2 member server named dscclient01 to keep the Internet Information Services (IIS) web server installed and the default website stopped.

We'll start by using **OneGet** to download and install the **xWebAdministration** custom DSC resource module. Next we'll author our configuration script and push it to a target node. Finally, we'll verify that the configuration "took" by intentionally producing configuration drift and testing autocorrection. If you don't have WMF v5 installed on your nodes, go with v4 and simply download the **xWebAdministration** DSC resource package from TechNet (http://bit.ly/13VAcvz).

1. On your DSC authoring server (dscserver01 in my case), fire up an elevated PowerShell v5 session and install the **xWebAdministration** package:

```
Find-Package -Name "xWebAdministration" | Install-Package -Verbose
```

   Remember that you need to run this command on all DSC nodes, which means both my authoring server as well as my dscclient01.company.pri target node.

   You'll also want to verify that the DSC resource has been installed in the proper location on disk:

```
PS C:\Program Files\WindowsPowerShell\Modules> dir

    Directory: C:\Program Files\WindowsPowerShell\Modules

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----       12/22/2014  12:49 PM                xWebAdministration
```

2. Now open an elevated ISE instance and create a new .ps1 script file named **WebServerConfig.ps1**. Check out Figure 18.8, and I'll walk you through the most important code lines, as has become my habit:

```
1  configuration SetupIIS
2  {
3      param
4      (
5          [string[]]$NodeName = 'localhost'
6      )
7
8      Import-DscResource -Module xWebAdministration
9
10     Node $NodeName
11     {
12         # Install the IIS server role
13         WindowsFeature IIS
14         {
15             Ensure          = "Present"
16             Name            = "Web-Server"
17         }
18         # Stop the Default Web Site website
19         xWebsite DefaultSite
20         {
21             Ensure          = "Present"
22             Name            = "Default Web Site"
23             State           = "Stopped"
24             PhysicalPath    = "C:\inetpub\wwwroot"
25             DependsOn       = "[WindowsFeature]IIS"
26         }
27     }
28 }
29 SetupIIS -NodeName ("dscserver01", "dscclient01")
```

FIGURE 18.8
Our DSC configuration script ensures that IIS is installed and that the Default Web Site website is stopped.

Line 5: Once again, we parameterize the **Node** name to make the script more flexible.

Line 8: This is a bit of "smoke and mirrors." We need to run **Import-DscResource** to load our custom DSC resource into the runspace. However, this is a "dynamic keyword" and is not an honest-to-goodness PowerShell cmdlet.

Lines 13–17: Here we invoke the built-in **WindowsFeature** resource to ensure that IIS is installed on the target node.

Lines 19–25: Now we call our **xWebSite** custom DSC resource to stop the default website.

Line 25: The **DependsOn** property is helpful when a configuration setting will work only if another one is active. Logically, then, we understand that we can stop the default website only if there exists an IIS web server to begin with.

Line 29: Modify the call to the configuration to target your own machines.

3. Run your WebServerConfig.ps1 script by pressing **F5** in the integrated scripting environment (ISE) and verify that PowerShell created two MOF files in a separate directory named after the script file.

4. When you're ready, unleash the proverbial hounds and apply the new configuration by running **Start-DscConfiguration**:

```
Start-DscConfiguration -Path "C:\SetupIIS"
```

5. Because PowerShell runs DSC configuration pushes as background jobs, we can use traditional syntax to check on job status:

```
PS C:\> Receive-Job -id 7 -Keep

PSComputerName
--------------
dscclient01
dscserver01
```

   Cool. No errors on my end. How has everything gone in your neck of the woods?

6. Connect to one of your target modes and see whether you can start the IIS Manager. If so, is the default website stopped? Figure 18.9 demonstrates my dscclient01 machine's compliance.

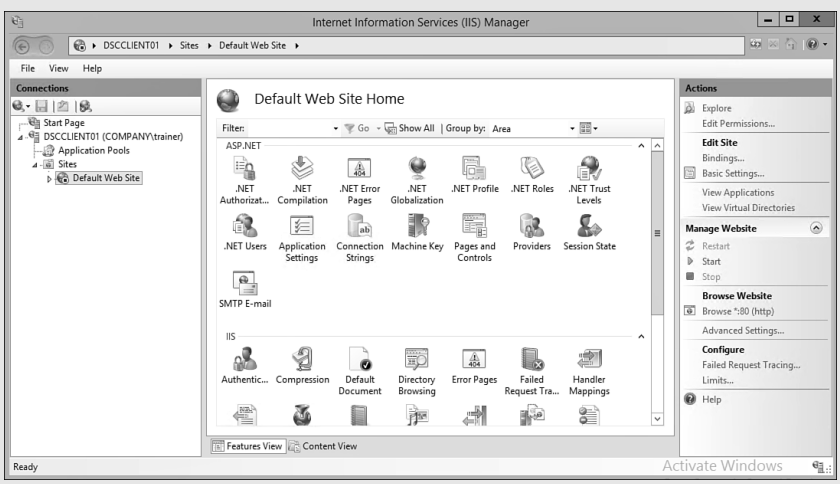


FIGURE 18.9
I'm just going to go ahead and say it: DSC rocks! Here we see the configuration applied to my dscclient01 member server.

7. Use Windows PowerShell on one of your target nodes to start the default website. Of course, this will produce configuration drift. (If you haven't configured your LCM to perform autocorrection, go back and do that now.)

```
Start-Website -Name "Default Web Site"
```

8. If you want, you can simply wait for the next DSC LCM refresh interval to test whether your server turned the default website back off. Alternatively, and perhaps more conveniently, we can force a manual update:

```
Update-DscConfiguration
```

   The update will once again exist as a configuration background job. Note also that you can try **Get-DscConfigurationStatus** to review a node's current relationship to DSC.

9. Surprise! You should find that the **Update-DscConfiguration** job fails:

```
PS C:\> Receive-Job -Name Job4 -Keep
No attempt was made to get a configuration from the pull server because LCM
RefreshMode is currently set to Push.
    + CategoryInfo          : NotSpecified:
(root/Microsoft/...gurationManager:String) [], CimException
    + FullyQualifiedErrorId : MI RESULT 1
    + PSComputerName        : localhost
```

   Here's the deal: DSC push mode is great for test/demo situations because it's easy to set up. However, we'll have to run **Start-DscConfiguration** again from the authoring computer to refresh this policy. It's only in a pull server scenario that nodes have the ability to refresh their policies. This makes sense because in a client refresh, we need some server from which the client can check to verify it has the correct policies applied.

A Word on DSC Push Configuration

Due to space constraints in this book, I'll simply give you the barebones, "need-to-know" information regarding setting up a DSC pull server. Let's do that in a stepwise fashion, covering the highest-level steps:

1. Download and install the **xPSDesiredStateConfiguration** custom DSC resource from the TechNet Script Center or by using **OneGet**.
2. Create and deploy your pull server configuration script. The recipe I recommend for your config script comes to us courtesy of the Windows PowerShell team directly: http://bit.ly/1ARl7pc.
3. Create and deploy an LCM configuration script. You can find an excellent example at Pwrshell.net (http://bit.ly/1ARmlAJ).

These settings are important because we change the configuration mode from push to pull and we specify the URL of the pull server's web service. We also specify how long the client waits before updating its DSC policies.

The communication between the node and the web service occurs over HTTP or HTTPS, depending on your authentication requirements. That's an important point, actually; you want to do what you can to ensure that your nodes are pulling configuration from legitimate DSC pull servers. It would be a very bad day indeed if a malicious individual stood up a bogus pull server and borked up your DSC client nodes in the absence of Secure Sockets Layer / Transport Layer Security (SSL/TLS) server authentication.

Summary

This was an awesome hour of training, wasn't it? I hope you're as stoked about DSC as I am. I don't know about you, but I can't stand manually (re)configuring servers. Declaring how a server "should" look and letting DSC take care of maintaining compliance to that configuration is plain old awesome.

In Hour 19 we'll stay within the ISE because it's finally time for us to take charge of Windows PowerShell scripting. (And here you never thought you'd be a programmer.)

Q&A

Q. **I know that many PowerShell cmdlets have a -WhatIf flag that allows you to test a cmdlet before it runs. Is there a command or parameter we can use to verify that DSC is functioning on a node?**

A. Yes, indeed. You'll want to run **Test-DscConfiguration -Verbose** to instruct Windows PowerShell to process all of its DSC scripts; the output returns True if all tests pass.

Q. **Will you please look at my script? I ran the following two lines of code:**

```
PS C:> .\LCMConfig.ps1
PS C:> SetupLCM -NodeName "dscclient01"
```

and got a bunch of red error text, as shown in Figure 18.10. What's the problem?

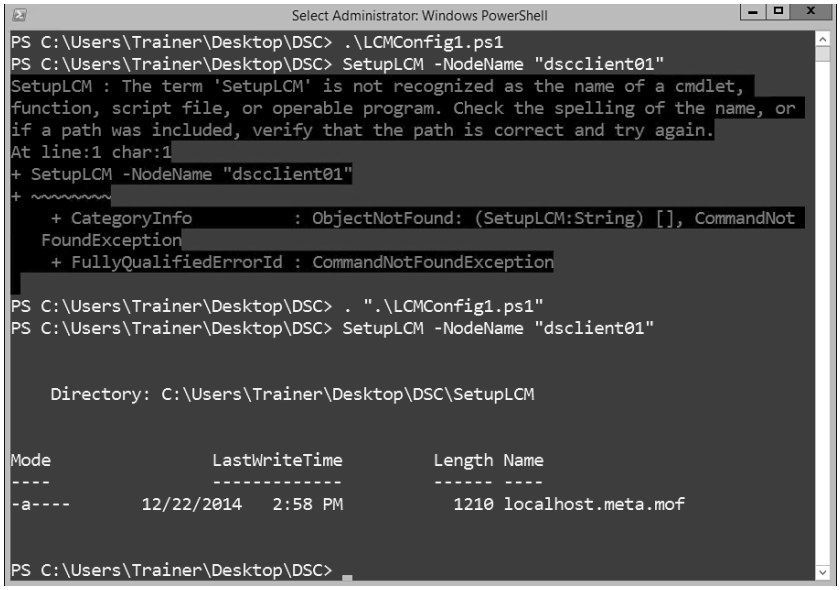


FIGURE 18.10
You need to understand the implications of "dot sourcing" your Windows PowerShell scripts.

A. The dot-slash (.\) notation simply tells PowerShell to run the present command from the current working directory and nothing more. This means that running a PowerShell script in this way runs the code contained inside the script but removes any functions, variables, and so forth from the session immediately thereafter.

Dot sourcing occurs when you type a period (.) and then type a partial or full path to a PowerShell script. (Don't forget to put quotes around the script path, including the dot slash.) The key difference with dot sourcing is that any objects defined inside the script persist in the user's current runspace. This allows us to run the DSC configuration script manually as was depicted in Figure 18.10.

Q. What is the suggested configuration refresh frequency for DSC?

A. How you configure your nodes' LCM component, and particularly the **ConfigurationFrequencyMins** property value, depends entirely on how much tolerance for configuration drift you have.

DSC network traffic is relatively low compared to, say, Group Policy. However, many Windows administrators are cool with setting refresh to 48 hours because the likelihood that a server will fall out of compliance may not be particularly high.

Workshop

Create a DSC configuration file that performs the following two tasks:

- Ensures that the Shutdown Event Tracker is enabled
- Ensures that the Google Chrome web browser is installed

The only hint I'll provide is that you need both the Registry built-in DSC resource and the xChrome custom resource to complete the configuration.

Quiz

1. You'd like to see what options are available for the **WindowsFeature** DSC resource. Which of the following commands accomplishes that goal?
   a. Get-Package
   b. Get-DscResource
   c. Get-Job
   d. Get-DscLocalConfigurationManager
2. The **UpdateDscConfiguration** cmdlet can be used only in DSC push scenarios.
   a. True
   b. False
3. A DSC authoring server running WMF 4.0 can push a configuration to a server running WMF 5.0.
   a. True
   b. False

Answers

1. The correct answer is B. Here is a run of the **WindowsFeature** DSC resource properties from my Windows Server 2012 R2 domain controller:

```
PS C:\> (Get-DscResource -Name WindowsFeature).Properties

Name                        PropertyType             IsMandatory Values
----                        ------------             ----------- ------
Name                        [string]                 True {}
Credential                  [PSCredential]           False {}
DependsOn                   [string[]]               False {}
Ensure                      [string]                 False {Absent, Present}
IncludeAllSubFeature        [bool]                   False {}
LogPath                     [string]                 False {}
Source                      [string]                 False {}
```

   Remember that you can run **Get-Command -module PSDesiredStateConfiguration** to retrieve a list of all DSC commands.

2. The correct answer is B. **Update-DscConfiguration** refreshes the target node only when a DSC pull server is online and available.

3. The correct answer is A. Remember that backward compatibility is a priority for the Windows PowerShell team. To use DSC, all your nodes must have the DSC bits available, which means that your Powershell version is 4 or 5 and your host operating system is Windows Server 2012/R2 or Windows 8/8.1.

This page intentionally left blank

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z