Author Picks

FREE

Exploring PowerShell Automation

Chapters selected by
Richard Siddaway

manning

Exploring PowerShell Automation

Selections by Richard Siddaway

Manning Author Picks

For online information and ordering of these and other Manning books, please visit www.manning.com. The publisher offers discounts on these books when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

Manning Publications Co.
20 Baldwin Road Technical
PO Box 761
Shelter Island, NY 11964

Cover designer: Leslie Haimes

ISBN 9781617294525
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 - EBM - 21 20 19 18 17 16

contents

introduction

PowerShell is ten years old in November 2016. Over that decade a significant, and continually increasing, percentage of Windows administrators have learned that PowerShell enables them to be more productive. They've realised that PowerShell enables them to perform administrative tasks across a wide range of technologies from Microsoft and third party vendors. The time taken to develop PowerShell scripts is paid back multiple times– by automating repetitive tasks and reducing errors through the use of repeatable, reliable processes.

When PowerShell was first introduced there were only a small number of commands available. You had to learn to script if you wanted to perform any complex administration. Over time, particularly with the release of Windows 8 / Server 2012, the number of commands has greatly increased, to the point that all major components in Windows, or major Microsoft products, have PowerShell support available. The range of third party vendors with PowerShell support is staggering – VMWare, NetApp, IBM, Cisco and EMC to name a few. It's even possible to administer Linux machines using PowerShell through CIM and Desired State Configuration!

This ebook gives you an overview of using PowerShell to administer your environment. I've been involved in the production of all of these chapters either as an author or an editor, and I've chosen them specifically to represent the breadth of possibilities for administering your systems through PowerShell. The first two chapters provide an overview of PowerShell and PowerShell remoting. The remaining three chapters provide examples of using PowerShell to administer SQL Server, IIS and Active Directory – three components that'll be found in practically any Windows environment.

Enjoy!

Reading suggestions

If you're new to PowerShell this suggested list of books provides a good learning path. I'd suggest reading the following in this order:

- Learn PowerShell in a Month of Lunches
- PowerShell in Depth, second edition
- PowerShell Deep Dives

Once you've learned the PowerShell basics it's time to put that knowledge into practice. These books show you how to administer a number of common technologies using PowerShell:

- PowerShell in Practice
- PowerShell and WMI
- Learn Active Directory Management in a Month of Lunches
- Learn Windows IIS in a Month of Lunches
- Learn Windows Server in a Month of Lunches
- Learn Hyper-V in a Month of Lunches

If you want to learn how the PowerShell language works and why it works the way it does you need to read:

- PowerShell in Action, third edition

Introduction to PowerShell

I've included this chapter for two reasons. Firstly, the chapter is co-authored by Bruce Payette (one half of the team that designed the PowerShell language) who has been a leading developer on the PowerShell team since its inception. Who better to introduce you to PowerShell? Secondly, the chapter provides an excellent introduction to PowerShell fundamentals, including pipelines, formatting of output, and PowerShell's elastic syntax.

Chapter 1 from *Windows PowerShell in Action, Third Edition* by Bruce Payette and Richard Siddaway

Welcome to PowerShell

Vizzini: Inconceivable!
Inigo: You keep on using that word. I do not think it means what you think it means.

—William Goldman, *The Princess Bride*

This chapter covers

- Core concepts
- Aliases and elastic systems
- Parsing and PowerShell
- Pipelines
- Formatting and output

It may seem strange for us to start by welcoming you to PowerShell when PowerShell is nine years old (at the time of writing), is on its fifth version, and this is the third edition of this book. In reality the adoption of PowerShell is only now achieving significant momentum, meaning that to many users PowerShell is a new tech-

nology and the three versions of PowerShell subsequent to the book's second edition contain many new features. Welcome to PowerShell.

> **NOTE** This book's written using PowerShell 5.0. It'll be noted in the text where earlier versions are different, or work in a different manner. We'll also document when various features were introduced to PowerShell or significantly modified between versions.

Windows PowerShell is the command and scripting language from Microsoft built into all versions of Windows since Windows Server 2008. Although PowerShell is new and different (or has new features you haven't explored yet), it's been designed to make use of what you already know, making it easy to learn. It's also designed to allow you to learn a bit at a time.

Running PowerShell commands

You have two choices for running the examples provided in this book. First choice is to use the PowerShell console. This provides a command line interface – based on the same console used for cmd.exe. It is the tool of choice for interactive work.

The second choice is the PowerShell Integrated Scripting Environment (ISE). The ISE supplies an editing pane plus a combined output and interactive pane. The ISE is the tool of choice when developing scripts, functions, and other advanced functionality.

The examples in the book will be written in a way that allows pasting directly into either tool.

Other third party tools exist, such as those supplied by Sapien, but we'll only consider the native tools in this book.

Starting at the beginning, here's the traditional "Hello world" program in PowerShell:

```
'Hello world.'
```

But "Hello world" itself isn't interesting. Here's something a bit more complicated:

```
dir $env:windir\*.log | Select-String -List error |
Format-Table path,linenumber -AutoSize
```

Although this is more complex, you can probably still figure out what it does. It searches all the log files in the Windows directory, looking for the string "error", and then prints the full name of the matching file and the matching line number. "Useful, but not special," you might think, because you can easily do this using `cmd.exe` on Windows or bash on UNIX. What about the "big, really big" thing? Well, how about this example?

```
([xml] [System.Net.WebClient]::new().
    DownloadString('http://blogs.msdn.com/powershell/rss.aspx')).
        RSS.Channel.Item |
            Format-Table title,link
```

Now we're getting somewhere. This script downloads the RSS feed from the PowerShell team blog and then displays the title and a link for each blog entry. By the way, you weren't expected to figure out this example yet. If you did, you can move to the head of the class!

Finally, one last example:

```
using assembly  System.Windows.Forms
using namespace System.Windows.Forms
$form = [Form] @{
    Text = 'My First Form'
}
$button = [Button] @{
    Text = 'Push Me!'
    Dock = 'Fill'
}
$button.add_Click{
    $form.Close()
}
$form.Controls.Add($button)
$form.ShowDialog())
```

This script uses the Windows Forms library (WinForms) to build a graphical user interface (GUI) that has a single button displaying the text "Push Me!" The window this script creates is shown in figure 1.1.

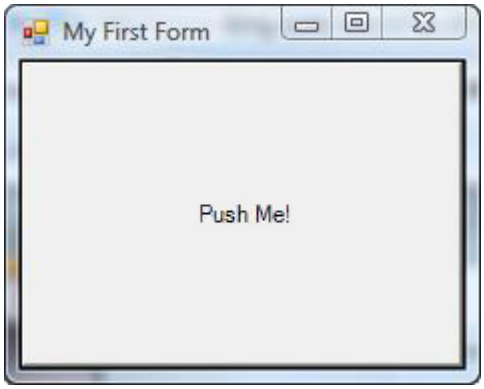


Figure 1.1 When you run the code from the example, this window will be displayed.

When you click the button, it closes the form and exits the script. With this you go from `"Hello world"` to a GUI application in less than two pages.

Now let's come back down to Earth for a minute. The intent of chapter one is to set the stage for understanding PowerShell—what it is, what it isn't, and, almost as important, why the PowerShell team made the decisions they made in designing the PowerShell language. Chapter one covers the goals of the project, along with some of the major issues the team faced in trying to achieve those goals. First, a philosophical digression: while under development, from 2002 until the first public release in 2006,

the codename for this project was Monad. The name Monad comes from *The Monadology* by Gottfried Wilhelm Leibniz, one of the inventors of calculus. Here's how Leibniz defined the Monad:

> *The Monad, of which we shall here speak, is nothing but a simple substance, which enters into compounds. By "simple" is meant "without parts."*
>
> —From *The Monadology* by Gottfried Wilhelm Leibniz (translated by Robert Latta)

In *The Monadology*, Leibniz described a world of irreducible components from which all things could be composed. This captures the spirit of the project: to create a toolkit of simple pieces that you compose to create complex solutions.

1.1 What is PowerShell?

More specifically, what is PowerShell, and what can you do with it? Ask a group of PowerShell users and you'll get different answers:

- PowerShell is a command-line shell
- PowerShell is a scripting environment
- PowerShell is an automation engine

These are all part of the answer. We prefer to say that PowerShell is a tool you can use to manage your Microsoft based machines and applications, that programs consistency into your management process. The tool is attractive to administrators and developers in that it can span the range of command line, simple and advanced scripts, to real programs.

> **NOTE** If you take this to mean that PowerShell is the ideal devops tool for the Microsoft platform then congratulations – you've got it in one.

PowerShell draws heavily from existing command-line shell and scripting languages, but the language, runtime and subsequent additions, such as PowerShell Workflows and Desired State Configuration, were designed from scratch to be an optimal environment for the modern Windows operating system.

Most people are introduced to PowerShell through its interactive aspects. Let's refine our definitions of shell and scripting.

1.1.1 Shells, command lines, and scripting languages

In the previous section, we called PowerShell a command-line shell. You may be asking, what's a shell? And how is that different from a command interpreter? What about scripting languages? If you can script in a shell language, doesn't that make it a scripting language? In answering these questions, let's start with shells.

Defining a shell can be tricky because pretty much everything at Microsoft has something called a *shell*. Windows Explorer is a shell. Visual Studio has a component called the shell. Heck, even the Xbox has something they call a shell.

Historically, the term *shell* describes the piece of software that sits over an operating system's core functionality. This core functionality is known as the *operating system kernel* (shell…kernel…get it?). A shell is the piece of software that lets you access the functionality provided by the operating system – for our purposes we're more interested in the traditional text-based environment where the user types a command and receives a response. Put another way, it is a shell is a command-line interpreter. The two terms can be used for the most part interchangeably.

SCRIPTING LANGUAGES VS. SHELLS

If this is the case, what is scripting and why are scripting languages not shells? To some extent, there's no difference. Many scripting languages have a mode in which they take commands from the user and then execute those commands to return results. This mode of operation is called a *Read-Evaluate-Print loop*, or REPL. In what way is a scripting language with a Read-Evaluate-Print loop not a shell? The difference is mainly in the user experience. A proper command-line shell is also a proper user interface. As such, a command line has to provide a number of features to make the user's experience pleasant and customizable, including aliases (shortcuts for hard-to-type commands), wildcard matching to avoid having to type out full names, and the ability to start other programs easily. Finally, command-line shells provide mechanisms for examining, editing, and re-executing previously typed commands. These mechanisms are called *command history*.

If scripting languages can be shells, can shells be scripting languages? The answer is, emphatically, yes. With each generation the UNIX shell languages have grown increasingly powerful. It's entirely possible to write substantial applications in a modern shell language, such as bash or zsh. Scripting languages characteristically have an advantage over shell languages, in that they provide mechanisms to help you develop larger scripts by letting you break a script into components, or *modules*. Scripting languages typically provide more sophisticated features for debugging your scripts. Next, scripting language runtimes are implemented in a way that makes their code execution more efficient, and scripts written in these languages execute more quickly than they'd in the corresponding shell script runtime. Finally, scripting language syntax is oriented more toward writing an application than toward interactively issuing commands.

In the end, there's no hard-and-fast distinction between a shell language and a scripting language. Because Power-Shell's goal is to be both a good scripting language and a good interactive shell, balancing the trade-offs between user-experience and script authoring was one of the major language design challenges.

MANAGING WINDOWS THROUGH OBJECTS

Another factor that drove the need for a new shell model is, as Windows acquired more and more subsystems and features, the number of issues we had to think about when managing a system increased dramatically. To help us deal with this increase in complexity, the manageable elements were factored into structured data objects. This

collection of *management objects* is known internally at Microsoft as the *Windows management surface.*

> **NOTE** Microsoft wasn't the only company that was running into issues caused by increased complexity. Most people in the industry were having this problem. This led to the Distributed Management Task Force (dmtf.org), an industry organization, creating a standard for management objects called the Common Information Model (CIM). Microsoft's implementation of this standard is called the *Windows Management Instrumentation* (WMI).

Although this factoring addressed overall complexity and worked well for graphical interfaces, it made it much harder to work with using a traditional text-based shell environment.

Windows is an API driven operating system compared to Unix and its derivatives, which are document (or text) driven. You can administer Unix by changing configuration files. In Windows you need to use the API which means accessing properties and using methods on the appropriate object.

Finally, as the power of the PC increased, Windows began to move off the desktop and into the corporate datacenter. In the corporate datacenter we had a large number of servers to manage, and the graphical point-and-click management approach didn't scale. All these elements combined to make it clear that Microsoft could no longer ignore the command line.

Now that you grasp the environmental forces that led to the creation of PowerShell—the need for command-line automation in a distributed object-based operating environment—let's look at the form the solution took.

1.2 PowerShell example code

We've said PowerShell is for solving problems that involve writing code. By now you're probably asking "Dude! Where's my code?" Enough talk, let's see some example code! First, we'll revisit the dir example. This time, instead of displaying the directory listing, you'll save it into a file using output redirection like in other shell environments. In the following example, you'll use dir to get information about a file named somefile.txt in the root of the C: drive. Using redirection, you direct the output into a new file, c:\foo.txt, and then use the type command to display what was saved. Here's what this looks like:

```
dir c:\somefile.txt > c:\foo.txt
type c:\foo.txt

        Directory: C:\


Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         11/07/2015     14:49          0 somefile.txt
```

As you can see, commands work more or less as you'd expect.

NOTE Okay, on your system choose any file that does exist and the example will work fine, though obviously the output will be different.

Let's go over some other things that should be familiar to you.

1.2.1 Navigation and basic operations

The PowerShell commands for working with the file system should be pretty familiar to most users. You navigate around the file system with the `cd` command. Files are copied with the `copy` or `cp` commands, moved with the `move` and `mv` commands, and removed with the `del` or `rm` commands. Why two of each command, you might ask? One set of names is familiar to `cmd.exe`/DOS users and the other is familiar to UNIX users. In practice they're *aliases* for the same command, designed to make it easy for people to get going with PowerShell. One thing to keep in mind is that, although the commands are similar they're not exactly the same as either of the other two systems. You can use the `Get-Help` command to get help about these commands. Here's the output of `Get-Help` for the `dir` command:

```
PS> Get-Help dir

NAME
    Get-ChildItem

SYNOPSIS
    Gets the items and child items in one or more specified locations.


SYNTAX
    Get-ChildItem [[-Path] <String[]>] [[-Filter] <String>] [-Exclude
     <String[]>] [-Force] [-Include <String[]>]
    [-Name] [-Recurse] [-UseTransaction [<SwitchParameter>]]
     [<CommonParameters>]

    Get-ChildItem [[-Filter] <String>] [-Exclude <String[]>] [-Force] [-
     Include <String[]>] [-Name] [-Recurse]
    -LiteralPath <String[]> [-UseTransaction [<SwitchParameter>]]
     [<CommonParameters>]


DESCRIPTION
    The Get-ChildItem cmdlet gets the items in one or more specified
     locations. If the item is a container, it gets
    the items inside the container, known as child items. You can use the
     Recurse parameter to get items in all child
    containers.

    A location can be a file system location, such as a directory, or a
     location exposed by a different Windows
    PowerShell provider, such as a registry hive or a certificate store.
```

```
RELATED LINKS
    Online Version: http://go.microsoft.com/fwlink/p/?linkid=290488
    Get-Alias
    Get-Item
    Get-Location
    Get-Process
    about_Providers

REMARKS
    To see the examples, type: "get-help Get-ChildItem -examples".
    For more information, type: "get-help Get-ChildItem -detailed".
    For technical information, type: "get-help Get-ChildItem -full".
    For online help, type: "get-help Get-ChildItem -online"
```

PowerShell help system

The PowerShell help subsystem contains information about all of the commands provided with the system and is a great way to explore what's available.

In PowerShell 3.0, and later, help files aren't installed by default. Help has become updatable and you need to install the latest versions yourself. See Get-Help about_Updatable_Help.

You can even use wildcard characters to search through the help topics (v2 and later). This is the simple text output. The PowerShell ISE also includes help in the richer Windows format and will even let you select an item and then press F1 to view the help for the item. Finally, by using the `-Online` option to `Get-Help`, you can view the help text for a command or topic using a web browser.

`Get-Help -Online` is the best way to get help because the online documentation is constantly being updated and corrected, whereas the local copies aren't.

1.2.2 Basic expressions and variables

In addition to running commands, PowerShell can evaluate expressions. In effect, it operates as a kind of calculator. Let's evaluate a simple expression:

```
2+2
4
```

Notice that as soon as you typed the expression, the result was calculated and displayed. It wasn't necessary to use any kind of print statement to display the result. It's important to remember that whenever an expression is evaluated, the result of the expression is output, not discarded. PowerShell supports most of the basic arithmetic operations you'd expect, including floating point.

You can save the output of an expression to a file by using the redirection operator:

```
(2+2)*3/7 > c:\foo.txt
type c:\foo.txt
1.71428571428571
```

Saving expressions into files is useful; saving them in variables is more useful:

```
$n = (2+2)*3
$n
12
$n / 7
1.71428571428571
```

Variables can also be used to store the output of commands:

```
$files = dir
$files[1]
    Directory: Microsoft.PowerShell.Core\FileSystem::C:\Document
    s and Settings\brucepay

    Directory: C:\Users\Richard\Documents


Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----         18/08/2014     20:11            Custom Office Templates
```

In this example, you extracted the second element of the collection of file information objects returned by the `dir` command. You were able to do this because you saved the output of the `dir` command as an array of objects in the `$files` variable.

> **NOTE** Collections in PowerShell start at 0, not 1. This is a characteristic we've inherited from .NET. This is why `$files[1]` is extracting the second element, not the first.

Given that PowerShell is all about objects, the basic operators need to work on more than numbers. Chapters three and four cover these features in detail.

1.2.3 Processing data

As you've seen in the preceding sections, you can run commands to get information, perform some basic operations on this information using the PowerShell operators, and then store the results in files and variables. Let's look at additional ways you can process this data. First you'll see how to sort objects and how to extract properties from those objects. Then we'll look at using the PowerShell flow-control statements to write scripts that use conditionals and loops to do more sophisticated processing.

SORTING OBJECTS

Sort the list of file information objects returned by `dir`. Because you're sorting objects, the command you'll use is `Sort-Object`. For convenience you'll use the shorter alias `sort` in these examples. Start by looking at the default output, which shows the files sorted by name:

```
cd c:\files
dir
    Directory: C:\files
```

```
    Directory: C:\files


Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---        21/01/2015     18:10          9 File 1.txt
-a---        11/07/2015     15:14      15986 File 2.txt
-a---        21/01/2015     18:10          9 File 3.txt
-a---        21/01/2015     18:10          9 File 4.txt
```

The output shows the basic properties on the file system objects, sorted by the name of the file. Let's sort by name in descending order:

```
dir | sort -Descending
            Directory: C:\files


Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---        21/01/2015     18:10          9 File 4.txt
-a---        21/01/2015     18:10          9 File 3.txt
-a---        11/07/2015     15:14      15986 File 2.txt
-a---        21/01/2015     18:10          9 File 1.txt
```

There you have it—files sorted by name in reverse order. Now let's sort by something other than the name of the file: file length.

NOTE In many examples in this book we'll be using aliases (shortcuts) rather than the full cmdlet name. This is for brevity and to ensure the code fits neatly in the page.

In PowerShell, when you use the `Sort-Object` cmdlet, you don't have to tell it to sort numerically—it already knows the type of the field, and you can specify the sort key by property name instead of a numeric field offset. The result looks like this:

```
dir | sort -Property length
        Directory: C:\files


Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---        21/01/2015     18:10          9 File 3.txt
-a---        21/01/2015     18:10          9 File 4.txt
-a---        21/01/2015     18:10          9 File 1.txt
-a---        11/07/2015     15:14      15986 File 2.txt
```

This illustrates what working with pipelines of objects gives you:

- You have the ability to access data elements by name instead of using substring indexes or field numbers.
- By having the original type of the element preserved, operations execute correctly without you having to provide additional information.

Now let's look at some other things you can do with objects.

SELECTING PROPERTIES FROM AN OBJECT

In this section, we'll introduce another cmdlet for working with objects: `Select-Object`. This cmdlet allows you to select a subrange of the objects piped into it and to specify a subset of the properties on those objects.

Say you want to get the largest file in a directory and put it into a variable:

```
$a = dir | sort -Property length -Descending |
Select-Object -First 1
$a
        Directory: C:\files


Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         11/07/2015     15:14      15986 File 2.txt
```

> **NOTE** You'll notice the secondary prompt >> when you copy the previous example into a PowerShell console. The first line of the command ended in a pipe symbol. The PowerShell interpreter noticed this, saw that the command was incomplete, and prompted for additional text to complete the command. Once the command is complete, you type a second blank line to send the command to the interpreter. If you want to cancel the command, you can press Ctrl-C at any time to return to the normal prompt.

Now say you want only the name of the directory containing the file and not all the other properties of the object. You can also do this with `Select-Object`. As with the `Sort-Object` cmdlet, `Select-Object` takes a `-Property` parameter (you'll see this frequently in the PowerShell environment—commands are consistent in their use of parameters):

```
$a = dir | sort -Property length -Descending |
Select-Object -First 1 -Property directory
$a

Directory
---------
C:\files
```

You now have an object with a single property.

PROCESSING WITH THE FOREACH-OBJECT CMDLET

The final simplification is to get the value itself. We'll introduce a new cmdlet that lets you do arbitrary processing on each object in a pipeline. The `ForEach-Object` cmdlet executes a block of statements for each object in the pipeline. You can get an arbitrary property out of an object and then do arbitrary processing on that information using the `ForEach-Object` command. Here's an example that adds up the lengths of all the objects in a directory:

```
$total = 0
dir | ForEach-Object {$total += $_.length }
$total
16013
```

In this example you initialize the variable `$total` to 0, and then add to it the length of each file returned by the `dir` command and finally display the total (you'll get a different total on your system).

PROCESSING OTHER KINDS OF DATA

One of the great strengths of the PowerShell approach is that once you learn a pattern for solving a problem, you can use this same pattern over and over again. For example, say you want to find the largest three files in a directory. The command line might look like this:

```
dir | sort -Descending length | select -First 3
```

Here, the `dir` command retrieved the list of file information objects, sorted them in descending order by length, and then selected the first three results to get the three largest files.

Now let's tackle a different problem. You want to find the three processes on the system with the largest working set size. Here's what this command line looks like:

```
Get-Process | sort -Descending ws | select -First 3
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
   1337    1916   235360     287852  1048    63.23   2440 WWAHost
    962      55    94460     176008   692   340.25   6632 WINWORD
    635      40   136040     140088   783     6.42   2564 powershell
```

This time you run `Get-Process` to get data about the processes on this computer, and sort on the working set instead of the file size. Otherwise, the pattern is identical to the previous example. This command pattern can be applied over and over again.

> **NOTE** Because of this ability to apply a command pattern over and over, most of the examples in this book are deliberately generic. The intent is to highlight the pattern of the solution rather than show a specific example. Once you understand the basic patterns, you can effectively adapt them to solve a multitude of other problems.

1.2.4 Flow-control statements

Pipelines are great, but sometimes you need more control over the flow of your script. PowerShell has the usual script flow-control statements found in most programming languages. These include the basic `if` statements, a powerful `switch` statement, and various loops like a `while` loop, `for` and `foreach` loops, and so on. Here's an example showing use of the `while` and if statements:

```
$i=0
while ($i++ -lt 10) { if ($i % 2) {"$i is odd"}}
1 is odd
3 is odd
5 is odd
7 is odd
9 is odd
```

This example uses the `while` loop to count through a range of numbers, printing out only the odd numbers. In the body of the `while` loop is an `if` statement that tests to see whether the current number is odd, and then writes out a message if it is. You can do the same thing using the `foreach` statement and the range operator (`..`), but much more succinctly:

```
foreach ($i in 1..10) { if ($i % 2) {"$i is odd"}}
```

The `foreach` statement iterates over a collection of objects, and the range operator is a way to generate a sequence of numbers. The two combine to make looping over a sequence of numbers clean.

Because the range operator generates a sequence of numbers, and numbers are objects like everything else in PowerShell, you can implement this using pipelines and the `ForEach-Object` cmdlet:

```
1..10 | foreach { if ($_ % 2) {"$_ is odd"}}
```

These examples only scratch the surface of what you can do with the PowerShell flow-control statements (wait until you see the `switch` statement!). The complete set of control structures is covered in detail in chapter 5 with lots of examples.

1.2.5 Scripts and functions

What good is a scripting language if you can't package commands into scripts? PowerShell lets you do this by putting your commands into a text file with a .ps1 extension and then running that command. You can even have parameters in your scripts. Put the following text into a file called hello.ps1:

```
param($name = 'bub')
"Hello $name, how are you?"
```

Notice that the `param` keyword is used to define a parameter called `$name`. The parameter is given a default value of `'bub'`. Now you can run this script from the PowerShell prompt by typing the name as `.\hello`. You need the `.\` to tell PowerShell to get the command from the current directory.

> **NOTE** Before you can run scripts on a machine in the default configuration, you'll have to change the PowerShell execution policy to allow scripts to run. See `Get-Help –Online for_` detailed instructions `on execution_policies`. The default settings change between Windows versions, so be careful to check the execution policy setting.

The first time you run this script, you won't specify any arguments:

```
.\hello
Hello bub, how are you?
```

You see that the default value was used in the response. Run it again, but this time specify an argument:

```
.\hello Bruce
Hello Bruce, how are you?
```

Now the argument is in the output instead of the default value. Sometimes you want to have subroutines in your code. PowerShell addresses this need through functions. Let's turn the hello script into a function. Here's what it looks like:

```
function hello {
param($name = "bub")
"Hello $name, how are you"
}
```

The body of the function is exactly the same as the script. The only thing added is the `function` keyword, the name of the function, and braces around the body of the function. Now run it, first with no arguments as you did with the script:

```
hello
Hello bub, how are you
```

and then with an argument:

```
hello Bruce
Hello Bruce, how are you
```

Obviously the function operates in the same way as the script, except that PowerShell didn't have to load it from a disk file, making it a bit faster to call. Scripts and functions are covered in detail in chapter six.

1.2.6 Remote administration

In the previous sections, you've seen the kinds of things you can do with PowerShell on a single computer, but the computing industry has long since moved beyond a one-computer world. Being able to manage groups of computers, without having to physically visit each one, is critical in the modern cloud-orientated IT world where your server may easily be on another continent. To address this, PowerShell has built-in remote execution capabilities (remoting) and an execution model that ensures that if a command works locally it should also work remotely.

> **NOTE** Remoting was introduced in PowerShell 2.0. It isn't available in PowerShell 1.0

The core of PowerShell remoting is the `Invoke-Command` command (aliased to `icm`). This command allows you to invoke a block of PowerShell script on the current computer, on a remote computer, or on a thousand remote computers. Let's see some of this in action. Microsoft release patches for Windows on a regular basis. Some of those patches are critical, in that they resolve security related issues, and as an administrator you need to be able to test if the patch has been applied to the machines for which you are responsible. Checking a single machine is relatively easy – you can use the Windows update option in Control panel and view the installed updates as shown in figure 1.2

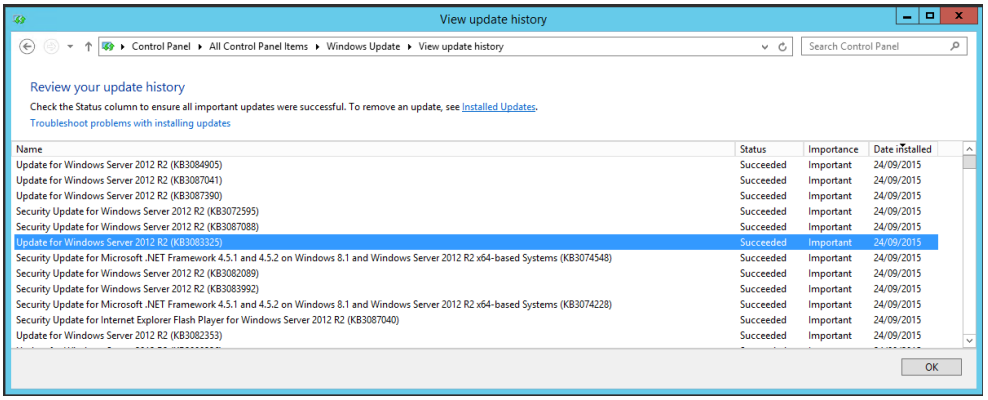


Figure 1.2 Viewing the installed updates on the local machine.

Alternatively, you can use the `Get-Hotfix` cmdlet:

```
 Get-HotFix -Id KB3083325

Source    Description HotFixID  InstalledBy          InstalledOn
------    ----------- --------  -----------          -----------
SERVER02 Update       KB3083325 NT AUTHORITY\SYSTEM
```

This shows you that the hotfix is installed on the local machine. But what about all of your other machines? Connecting to each one individually and using the control panel or running the `Get-Hotfix` cmdlet is tedious. You need a method of running the cmdlet on remote machines and having the results returned to your local machine.

Invoke-Command is used to wrap the previous command:

```
Invoke-Command -ScriptBlock {Get-HotFix -Id KB3083325} `
-ComputerName W12R2SCDC01

Source       Description HotFixID  InstalledBy          InstalledOn
------       ----------- --------  -----------          -----------
W12R2SCDC01 Update       KB3083325 NT AUTHORITY\SYSTEM 9/24/2015
```

NOTE Get-Hotfix has a –ComputerName parameter, and, like many cmdlets, is capable of working directly with remote machines. Cmdlet based remoting often uses protocols other than WSMAN. Using Invoke-Command, as in a PowerShell remoting session, is more efficient, as you'll see in chapter eleven.

You have many machines that need testing. Typing in the computer names one at a time is still too tedious. You can create a list of computers – either from a text file or in your code, and test them all:

```
 $computers = 'W12R2SCDC01', 'W12R2SUS'
Invoke-Command -ScriptBlock {Get-HotFix -Id KB3083325} `
-ComputerName $computers
```

```
Source      Description HotFixID  InstalledBy         InstalledOn
------      ----------- --------  -----------         -----------
W12R2SCDC01 Update      KB3083325 NT AUTHORITY\SYSTEM 9/24/2015
W12R2SUS    Update      KB3083325 NT AUTHORITY\SYSTEM 9/24/2015
```

What happens if a machine doesn't have the hotfix installed:

```
Invoke-Command -ScriptBlock {Get-HotFix -Id KB3080042} `
-ComputerName $computers
Cannot find the requested hotfix on the 'localhost' computer. Verify the
     input and run the command again.
    + CategoryInfo          : ObjectNotFound: (:) [Get-HotFix],
     ArgumentException
    + FullyQualifiedErrorId :
     GetHotFixNoEntriesFound,Microsoft.PowerShell.Commands.GetHotFixCommand
    + PSComputerName        : W12R2SCDC01


Source   Description HotFixID  InstalledBy         InstalledOn
------   ----------- --------  -----------         -----------
W12R2SUS Update      KB3080042 NT AUTHORITY\SYSTEM 9/24/2015
```

An error is generated on a computer that doesn't have the patch installed, and results appear on the computers that do.

> **NOTE** In a production script you'd put error handling in place to catch the error and report that the patch wasn't installed. This will be covered in chapter fourteen.

The `Invoke-Command` command is the way to programmatically execute PowerShell commands on a remote machine. When you want to connect to a machine to interact with it on a one-to-one basis, you use the `Enter-PSSession` command. This command allows you to start an interactive one-to-one session with a remote computer. Running `Enter-PSSession` looks like this:

```
Enter-PSSession -ComputerName W12R2SUS
[W12R2SUS]: > Get-HotFix -Id KB3080042

Source        Description      HotFixID      InstalledBy
------        -----------      --------      -----------
W12R2SUS      Update           KB3080042     NT AUTHORITY\SYSTEM


[W12R2SUS]: > Get-Date

28 September 2015 15:57:53


[W12R2SUS]: > Exit-PSSession
```

As shown here, when you connect to the remote computer, your prompt changes to indicate that you're working remotely. Otherwise, once connected, you can interact with the remote computer the same way you'd a local machine. When you're done,

exit the remote session with the `Exit-PSSession` command, which returns you to the local session. This brief introduction covers some powerful techniques, but we've only begun to cover all the things remoting lets you do.

At this point, we'll end our "Cook's tour" of PowerShell. We've only breezed over the features and capabilities of the environment. Many other areas of PowerShell aren't covered here. In upcoming chapters, we'll explore each of the elements discussed here in detail and a whole lot more.

1.3 Core concepts

The core PowerShell language is based on the mature IEEE standard POSIX 1003.2 grammar for the Korn shell, which has a long history as a successful basis for modern shells like bash and zsh. The language design team (Jim Truher and Bruce Payette) deviated from this standard where necessary to address the specific needs of an object-based shell and to make it easier to write sophisticated scripts.

PowerShell syntax is aligned with C#. The major value this brings is that PowerShell code can be migrated to C#, when necessary for performance improvements, and, more importantly, C# examples can be easily converted to PowerShell — the more examples you have in a language, the better off you are.

1.3.1 Command concepts and terminology

Much of the terminology used in PowerShell will be familiar if you've used other shells in the Linux or Windows world. Because PowerShell is a new kind of shell, there are a number of terms that are different and a few new terms to learn. In this section, we'll go over the PowerShell-specific concepts and terminology for command types and command syntax.

1.3.2 Commands and cmdlets

Commands are the fundamental part of any shell language; they're what you type to get things done. A simple command looks like this:

```
command –parameter1 –parameter2 argument1 argument2
```

A more detailed illustration of the anatomy of this command is shown in figure 1.3. This figure calls out all the individual elements of the command.
All commands are broken down into the command name, the parameters specified to the command, and the arguments to those parameters. You can think of a parameter as the receiver of a piece of information and the argument as the information itself.

> **NOTE** The distinction between *parameter* and *argument* may seem a bit strange from a programmer's perspective. If you're used to languages such as Python and Visual Basic, which allow for keyword parameters, PowerShell parameters correspond to the keywords, and arguments correspond to the values.

The first element in the command is the name of the command to be executed. The PowerShell interpreter looks at this name and determines which command to run,

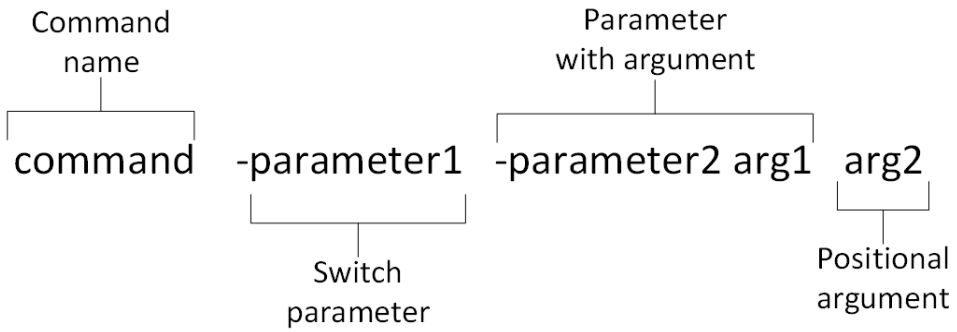


Figure 1.3 The anatomy of a basic command. It begins with the name of the command, followed by parameters. These may be switch parameters that take no arguments, regular parameters that do take arguments, or positional parameters, where the matching parameter is inferred by the argument's position on the command line.

and also which *kind* of command to run. In PowerShell there are a number of categories of commands: cmdlets, shell function commands, script commands, workflow commands, and native Windows commands. Following the command name come zero or more parameters and/or arguments. A parameter starts with a dash, followed by the name of the parameter. An argument, on the other hand, is the value that'll be associated with, or *bound to*, a specific parameter. Let's look at an example:

```
PS (1) > Write-Output -InputObject Hello
Hello
```

In this example, the command is `Write-Output`, the parameter is `-InputObject`, and the argument is `Hello`.

What about the positional parameters? When a PowerShell command is created, the author of that command specifies information that allows PowerShell to determine which parameter to bind an argument to, even if the parameter name itself is missing. For example, the `Write-Output` command has been defined such that the first parameter is `-InputObject`. This lets you write

```
PS (2) > Write-Output Hello
Hello
```

The piece of the PowerShell interpreter that figures all of this out is called the *parameter binder*. The parameter binder is smart—it doesn't require that you specify the full name of a parameter as long as you specify enough for it to uniquely distinguish what you mean.

> **NOTE** PowerShell isn't case sensitive but we use the correct casing on commands and parameters to aid reading. It's also a good practice when scripting, as it's easier to understand the code when you revisit it many months later.

What else does the parameter binder do? It's in charge of determining how to match the types of arguments to the types of parameters. Remember that PowerShell is an object-based shell. Everything in PowerShell has a type. PowerShell uses a fairly complex type-conversion system to correctly put things together. When you type a command at the command line, you're typing strings. What happens if the command requires a different type of object? The parameter binder uses the type converter to try to convert that string into the correct type for the parameter. If you use a value that can't be converted to the correct type you get an error message explaining that the type conversion failed. We discuss this in more detail in chapter two, when we talk about types.

What happens if the argument you want to pass to the command starts with a dash? This is where the quotes come in. Let's use `Write-Output` to print out the string "-InputObject":

```
PS (1) > Write-Output -InputObject "-InputObject"
-InputObject
```

And it works as desired. Alternatively, you could type this:

```
PS (2) > Write-Output "-InputObject"
-InputObject
```

The quotes keep the parameter binder from treating the quoted string as a parameter.

Another, less frequently used way of doing this is by using the special "end-of-parameters" parameter, which is two hyphens back to back (`--`). Everything after this sequence will be treated as an argument, even if it looks like a parameter. For example, using `--` you can also write out the string `-InputObject` without using quotes:

```
PS (3) > Write-Output -- -InputObject
-InputObject
```

This is a convention standardized in the POSIX Shell and Utilities specification.

The final element of the basic command pattern is the *switch parameter*. These are parameters that don't require an argument. They're usually either present or absent (obviously they can't be positional). A good example of this is the `-Recurse` parameter on the `dir` command. This switch tells the `dir` command to display files from a specified directory as well as all its subdirectories:

```
PS (1) > dir -Recurse -Filter c*d.exe c:\windows
        Directory: C:\windows\system32


Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         29/10/2014     00:37     141824 CloudStorageWizard.exe
-a---         29/10/2014     01:28     357376 cmd.exe
```

As you can see, the `-Recurse` switch takes no arguments.

NOTE Although it's almost always the case that switch parameters don't take arguments, it's possible to specify arguments to them. We'll save discussion of when and why you might do this for chapter seven, which focuses on scripts (shell functions and scripts are the only time you need this particular feature, and we'll keep you in suspense for the time being).

Now that we've covered the basic anatomy of the command line, let's go over the types of commands that PowerShell supports.

1.3.3 Command categories

As we mentioned earlier, there are four categories of commands in PowerShell: cmdlets, functions, scripts, and native Win32 executables.

CMDLETS

The first category of command is a cmdlet (pronounced "command-let"). *Cmdlet* is a term that's specific to the PowerShell environment. A cmdlet is implemented by a .NET class that derives from the `Cmdlet` base class in the PowerShell Software Developers Kit (SDK).

NOTE Building cmdlets is a developer task and requires the PowerShell SDK. This SDK is freely available for download from Microsoft and includes extensive documentation along with many code samples. Our goal is to coach you to effectively use and script in the PowerShell environment, and we're not going to do much more than mention the SDK in this book.

This category of command is compiled into a dynamic link library (DLL) and then loaded into the PowerShell process, usually when the shell starts up. Because the compiled code is loaded into the process, it's the most efficient category of command to execute.

Cmdlets always have names of the form `Verb-Noun`, where the verb specifies the action and the noun specifies the object on which to operate. In traditional shells, cmdlets correspond most closely to what's usually called a *built-in command*. In PowerShell, though, anybody can add a cmdlet to the runtime, and there isn't any special class of built-in commands.

FUNCTIONS

The next type of command is a *function*. This is a named piece of PowerShell script code that lives in memory as the interpreter is running, and is discarded on exit. Functions consist of user-defined code that's parsed when defined. This parsed representation is preserved in order that it doesn't have to be reparsed every time it's used.

Functions in PowerShell version 1 could have named parameters like cmdlets but were otherwise fairly limited. In version 2, and later, this was fixed, and scripts and functions now have the full parameter specification capabilities of cmdlets. The same basic structure is followed for both types of commands. Functions and cmdlets have the same streaming behavior.

PowerShell workflows were introduced in PowerShell 3.0. Their syntax is similar to that of a function. When the workflow is first loaded in memory a PowerShell function is created that can be viewed through the function: PowerShell drive. Workflows are covered in chapter twelve.

SCRIPTS

A *script command* is a piece of PowerShell code that lives in a text file with a .ps1 extension. These script files are loaded and parsed every time they're run, making them somewhat slower than functions to start (although once started, they run at the same speed). In terms of parameter capabilities, shell function commands and script commands are identical.

NATIVE COMMANDS (APPLICATIONS)

The last type of command is called a *native command*. These are external programs (typically executables) that can be executed by the operating system. Because running a native command involves creating a whole new process for the command, native commands are the slowest of the command types. Also, native commands do their own parameter processing and don't necessarily match the syntax of the other types of commands.

Native commands cover anything that can be run on a Windows computer, and you get a wide variety of behaviors. One of the biggest issues is when PowerShell waits for a command to finish but it keeps on going. For example, say you're starting a text document at the command line:

```
PS (1) > .\foo.txt
PS (2) >
```

You get the prompt back more or less immediately, and your default text editor will pop up (probably `notepad.exe` because that's the default). The program to launch is determined by the file associations that are defined as part of the Windows -environment.

> **NOTE** In PowerShell, unlike in `cmd.exe`, you have to prefix a command with `./` or `.\` if you want to run it out of the current directory. This is part of PowerShell's "Secure by Design" philosophy. This particular security feature was adopted to prevent Trojan horse attacks where the user is lured into a directory and then told to run an innocuous command such as `notepad.exe`. Instead of running the system `notepad.exe`, they end up running a hostile program that the attacker has placed in that directory and named `notepad.exe`.

What if you specify the editor explicitly?

```
PS (2) > notepad foo.txt
PS (3) >
```

The same thing happens—the command returns immediately. What if you run the command in the middle of a pipeline?

```
PS (3) > notepad foo.txt | sort-object
<exit notepad>
PS (4) >
```

This time PowerShell waits for the command to exit before giving you back the prompt. This can be handy when you want to insert something such as a graphical form editor in the middle of a script to do some processing. This is also the easiest way to make PowerShell wait for a process to exit. As you can see, the behavior of native commands depends on the type of native command, as well as where it appears in the pipeline.

A useful thing to remember is that the PowerShell interpreter itself is a native command: `powershell.exe`. This means you can call PowerShell from within PowerShell. When you do this, a second PowerShell process is created. In practice there's nothing unusual about this—that's how all shells work. PowerShell doesn't have to do it often, making it much faster than conventional shell languages.

The ability to run a child PowerShell process is particularly useful if you want to have isolation in portions of your script. A separate process means that the child script can't impact the caller's environment. This feature is useful enough that PowerShell has special handling for this case, allowing you to embed the script to run inline. If you want to run a fragment of script in a child process, you can do this by passing the block of script to the child process delimited by braces. Here's an example:

```
PS (1) > powershell { Get-Process *ss } | Format-Table name, handles

Name                                                    Handles
----                                                    -------
csrss                                                      1077
lsass                                                      1272
smss                                                         28
```

Two things should be noted in this example; the script code in the braces can be any PowerShell code, and it'll be passed through to the new PowerShell process. The special handling takes care of encoding the script in such a way that it's passed properly to the child process. The other thing to note is that, when PowerShell is executed this way, the output of the process is *serialized objects*—the basic structure of the output is preserved—and can be passed into other commands. We'll look at this serialization in detail when we cover *remoting*—the ability to run PowerShell scripts on a remote computer—in chapter twelve.

DESIRED STATE CONFIGURATION

Desired State Configuration (DSC) is a management platform in Windows PowerShell. It enables the deployment and management of configuration data for software services and the environment on which these services run. A configuration is created using PowerShell like syntax. The configuration is used to create a MOF (Managed Object Format) file which is passed to the remote machine on which the configuration will be applied. DSC is covered in chapter eighteen.

Now that we've covered the PowerShell command types, let's get back to looking at the PowerShell syntax. Notice that a lot of what we've examined this far is a bit verbose. This makes it easy to read, which is great for script maintenance, but it looks like it'd be a pain to type on the command line. PowerShell addresses these two conflicting goals—readability and writeability—with the concept of *elastic syntax*. Elastic syntax

allows you to expand and collapse how much you need to type to suit your purpose. We'll see how this works in the next section.

1.3.4 Aliases and elastic syntax

We haven't talked about aliases yet or how they're used to achieve an elastic syntax in PowerShell. Because this concept is important in the PowerShell environment, we need to spend some time on it.

The cmdlet `Verb-Noun` syntax, while regular, is, as we noted, also verbose. You may have noticed that in most of the examples we're using commands such as `dir` and `type`. The trick behind all this is aliases. The `dir` command is `Get-ChildItem`, and the `type` command is `Get-Content`. You can see this by using the `Get-Command` command:

```
PS (1) > Get-Command dir
CommandType Name                  ModuleName
----------- ----                  ----------
Alias       dir -> Get-ChildItem
```

This tells you that the command is an alias for `Get-ChildItem`. To get information about the `Get-ChildItem` command, you then do this

```
PS (2) > Get-Command Get-ChildItem
CommandType Name          ModuleName
----------- ----          ----------
Cmdlet      Get-ChildItem Microsoft.PowerShell.Management
```

To see all the information, pipe the output of `Get-Command` into `fl`. This shows you the full detailed information about this cmdlet. But wait—what's the `fl` command? Again you can use `Get-Command` to find out:

```
PS (4) > Get-Command fl
CommandType Name              ModuleName
----------- ----              ----------
Alias       fl -> Format-List
```

PowerShell comes with a large set of predefined aliases. Two basic categories of aliases exist—*transitional aliases* and *convenience aliases*. By transitional aliases, we mean a set of aliases that map PowerShell commands to commands that people are accustomed to using in other shells, specifically `cmd.exe` and the UNIX shells. For the `cmd.exe` user, PowerShell defines `dir`, `type`, `copy`, and so on. For the UNIX user, PowerShell defines `ls`, `cat`, `cp`, and so forth. These aliases allow a basic level of functionality for new users right away.

The other set of aliases are the convenience aliases. These aliases are derived from the names of the cmdlets they map to. `Get-Command` becomes `gcm`, `Get-ChildItem` becomes `gci`, `Invoke-Item` becomes `ii`, and so on. For a list of the defined aliases, type `Get-Alias` at the command line. You can use the `Set-Alias` command (whose alias is `sal`, by the way) to define your own aliases – many experienced PowerShell users create a set of one letter aliases to cover the cmdlets they most often use at the command prompt.

NOTE Aliases in PowerShell are limited to aliasing the command name only. Unlike in other systems such as ksh, bash, and zsh, PowerShell aliases can't include parameters. If you need to do something more sophisticated than simple command-name translations, you'll have to use shell functions or scripts.

This is all well and good, but what does it have to do with elastics? Glad you asked! The idea is that PowerShell can be terse when needed and descriptive when appropriate. The syntax is concise for simple cases and can be stretched like an elastic band for larger problems. This is important in a language that's both a command-line tool and a scripting language. Many "scripts" that you'll write in PowerShell will be no more than a few lines long. There'll be a string of commands that you'll type on the command line and then never use again. To be effective in this environment, the syntax needs to be concise. This is where aliases like `fl` come in—they allow you to write concise command lines. When you're scripting, though, it's best to use the long name of the command. Sooner or later, you'll have to read the script you wrote (or—worse—someone else will). Would you rather read something that looks like this

```
gcm|?{$_.parametersets.Count -gt 3}|fl name
```

or this?

```
Get-Command |
  Where-Object {$_.parametersets.count -gt 3} |
    Format-List name
```

We'd certainly rather read the latter. (As always, we'll cover the details of these examples later in the book.)

There's a second type of alias used in PowerShell: *parameter aliases*. Unlike command aliases, which can be created by end users, parameter aliases are created by the author of a cmdlet, script, or function. (You'll see how to do this when we look at advanced function creation in chapter seven.)

A parameter alias is a shorter name for a parameter. Wait a second, earlier we said that you needed enough of the parameter name to distinguish it from other command parameters. Isn't this enough for convenience and elasticity? Why do you need parameter aliases? The reason you need these aliases has to do with *script versioning*. The easiest way to understand versioning is to look at an example.

Say you have a script that calls a cmdlet `Process-Message`. This cmdlet has a parameter `-Reply`. You write your script specifying

```
Process-Message -Re
```

Run the script, and it works fine. A few months later, you install an enhanced version of the `Process-Message` command. This new version introduces a new parameter: `-receive`. only specifying `-Re` is no longer sufficient. If you run the old script with the new cmdlet, it'll fail with an ambiguous parameter message; the script is broken.

How do you fix this with parameter aliases? The first thing to know is that PowerShell always picks the parameter that exactly matches a parameter name or alias over a

partial match. By providing parameter aliases, you can achieve pithiness without also making scripts subject to versioning issues. We recommend always using the full parameter name for production scripts or scripts you want to share. Readability is always more important in that scenario.

Now that we've covered the core concepts of how commands are processed, let's step back a bit and look at PowerShell language processing overall. PowerShell has a small number of important syntactic rules you should learn. When you understand these rules, your ability to read, write, and debug PowerShell scripts will increase tremendously.

1.4 Parsing the PowerShell language

In this section, we'll cover the details of how PowerShell scripts are parsed. Before the PowerShell interpreter can execute the commands you type, it first has to parse the command text and turn it into something the computer can execute, as shown in figure 1.4.

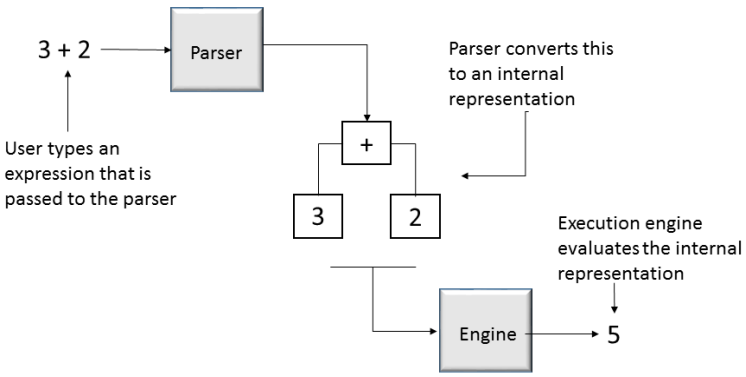


Figure 1.4 The flow of processing in the PowerShell interpreter, where an expression is transformed and then executed to produce a result

More formally, parsing is the process of turning human-readable source code into a form the computer understands. A piece of script text is broken up into tokens by the *tokenizer* (or *lexical analyzer*, if you want to be more technical). A token is a particular type of symbol in the programming language, such as a number, a keyword, or a variable. Once the raw text has been broken into a stream of tokens, these tokens are processed into structures in the language through syntactic analysis.

In syntactic analysis, the stream of tokens is processed according to the grammatical rules of the language. In normal programming languages, this process is straightforward—a token always has the same meaning. A sequence of digits is always a number; an expression is always an expression, and so on. For example, the sequence

```
3+2
```

would always be an addition expression, and "Hello world" would always be a constant string. Unfortunately, this isn't the case in shell languages. Sometimes you can't tell what a token is except through its context.

> **NOTE** More information on this and the inner workings of PowerShell is available in the PowerShell language specification at http://www.microsoft.com/en-us/download/details.aspx?id=36389. The specification is currently only available up to version 3.0 of the PowerShell language.

In the next section, we go into more detail on why this is, and how the PowerShell interpreter parses a script.

1.4.1 How PowerShell parses

For PowerShell to be successful as a shell, it can't require that everything be quoted. PowerShell would fail if it required people to continually type

```
cd ".."
```

or

```
copy "foo.txt" "bar.txt"
```

On the other hand, people have a strong idea of how expressions should work:

```
2
```

This is the number 2, not a string "2". Consequently, PowerShell has some rather complicated parsing rules. The next three sections will cover these rules. We'll discuss how quoting is handled, the two major parsing modes, and the special rules for newlines and statement termination.

1.4.2 Quoting

Quoting is the mechanism used to turn a token that has special meaning to the PowerShell interpreter into a simple string value. For example, the `Write-Output` cmdlet has a parameter `-InputObject`. But what if you want to use the string "-InputObject" as an argument? To do this, you have to quote it; you surround it with single or double quotes. The result looks like this:

```
PS (2) > Write-Output '-InputObject'
-inputobject
```

If you hadn't put the argument in quotes an error message is produced indicating that an argument to the parameter `-InputObject` is required.

PowerShell supports several forms of quoting, each with somewhat different meanings (or semantics). Putting single quotes around an entire sequence of characters causes them to be treated like a single string. This is how you deal with file paths that have spaces in them for example. If you want to change to a directory whose path contains spaces, you type this:

```
PS (4) > cd 'c:\program files'
PS (5) > pwd
Path
----
C:\Program Files
```

When you don't use the quotes, you receive an error complaining about an unexpected parameter in the command because `"c:\program"` and `"files"` are treated as two separate tokens.

> **NOTE** Notice that the error message reports the name of the cmdlet, not the alias that was used. This way you know what is being executed. The position message shows you the text that was entered in order that you can see an alias was used.

One problem with using matching quotes as we did in the previous examples is that you have to remember to start the token with an opening quote. This raises an issue when you want to quote a single character. You can use the backquote (`) character to do this (the backquote is usually the upper-leftmost key, below Esc):

```
PS (6) > cd c:\program` files
PS (7) > pwd
Path
----
C:\Program Files
```

The backquote, or *backtick*, as it tends to be called, has other uses that we'll explore later in this section. Now let's look at the other form of matching quote: double quotes. You'd think it works pretty much like the example with single quotes; what's the difference? In double quotes, variables are expanded. If the string contains a variable reference starting with a `$`, it'll be replaced by the string representation of the value stored in the variable. Let's look at an example. First assign the string "files" to the variable `$v`:

```
PS (10) > $v = 'files'
```

Now reference that variable in a string with double quotes:

```
PS (11) > cd "c:\program $v"
PS (12) > pwd

Path
----
C:\Program Files
```

The `cd` succeeded and the current directory was set as you expected.

> **NOTE** Variable expansion only occurs with double quotes. A common beginner error is to use single quotes and expect variable expansion to work.

What if you want to show what the value of `$v` is? To do this, you need to have expansion in one place but not in the other. This is one of those other uses we had for the

backtick. It can be used to quote or escape the dollar sign in a double-quoted string to suppress expansion. Let's try it:

```
PS (16) > Write-Output "`$v is $v"
$v is files
```

Here's one final tweak to this example—if `$v` contained spaces, you'd want to make clear what part of the output was the value. Because single quotes can contain double quotes and double quotes can contain single quotes, this is straightforward:

```
PS (17) > Write-Output "`$v is '$v'"
$v is 'files'
PS (18) >
```

Now, suppose you want to display the value of `$v` on another line instead of in quotes. Here's another situation where you can use the backtick as an escape character. The sequence `` `n `` in a double-quoted string will be replaced by a newline character. You can write the example with the value of `$v` on a separate line as follows:

```
PS (19) > "The value of `$v is:`n$v"
The value of $v is:
Files
```

The list special characters that can be generated using backtick (also called *escape*) sequences can be found using `Get-Help about_Escape_Characters`. Note that escape sequence processing, like variable expansion, is only done in double-quoted strings. In single-quoted strings, what you see is what you get. This is particularly important when writing a string to pass to a subsystem that does additional levels of quote processing.

1.4.3 Expression-mode and command-mode parsing

As mentioned earlier, because PowerShell is a shell, it has to deal with some parsing issues not found in other languages. PowerShell simplifies parsing considerably, trimming the number of modes down to two: expression mode and command mode.

In expression mode, the parsing is conventional: strings must be quoted, numbers are always numbers, and so on. In command mode, numbers are treated as numbers but all other arguments are treated as strings unless they start with `$`, `@`, `'`, `"`, or `(`. When an argument begins with one of these special characters, the rest of the argument is parsed as a value expression. (There's also special treatment for leading variable references in a string, which we'll discuss later.) Table 1.1 shows some examples that illustrate how items are parsed in each mode.

Table 1.1 Parsing mode examples

Example command line	Parsing mode and explanation
`2+2`	Expression mode; results in 4.
`Write-Output 2+2`	Command mode; results in 2+2.
`$a=2+2`	Expression mode; the variable `$a` is assigned the value 4.

Table 1.1 Parsing mode examples *(continued)*

Example command line	Parsing mode and explanation
`Write-Output (2+2)`	Expression mode; because of the parentheses, 2+2 is evaluated as an expression producing 4. This result is then passed as an argument to the `Write-Output` cmdlet.
`Write-Output $a`	Expression mode; produces 4. This is ambiguous—evaluating it in either mode produces the same result. The next example shows why the default is expression mode if the argument starts with a variable.
`Write-Output $a.Equals(4)`	Expression mode; `$a.Equals(4)` evaluates to true and `Write-Output` writes the Boolean value True. This is why a variable is evaluated in expression mode by default. You want simple method and property expressions to work without parentheses.
`Write-Output $a/foo.txt`	Command mode; `$a/foo.txt` expands to `4/foo.txt`. This is the opposite of the previous example. Here you want it to be evaluated as a string in command mode. The interpreter first parses in expression mode and sees that it's not a valid property expression, and it backs up and rescans the argument in command mode. As a result, it's treated as an expandable string.

Notice that in the `Write-Output (2+2)` case, the open parenthesis causes the interpreter to enter a new level of interpretation where the parsing mode is once again established by the first token. This means the sequence `2+2` is parsed in expression mode, not command mode, and the result of the expression (4) is emitted. Also, the last example in the table illustrates the exception mentioned previously for a leading variable reference in a string. A variable itself is treated as an expression, but a variable followed by arbitrary text is treated as though the whole thing were in double quotes. This allows you to write

```
cd $HOME/scripts
```

instead of

```
cd "$HOME/scripts"
```

As mentioned earlier, quoted and unquoted strings are recognized as different tokens by the parser. This is why

```
Invoke-MyCmdlet -Parm arg
```

treats `-Parm` as a parameter and

```
Invoke-MyCmdlet "-Parm" arg
```

treats `"-Parm"` as an argument. There's an additional wrinkle in the parameter binding. If an unquoted parameter like `-NotAparameter` isn't a parameter on `Invoke-MyCmdlet`, it'll be treated as an argument. This lets you say

```
Write-Host  -this -is -a parameter
```

without requiring quoting.

This finishes our coverage of the basics of parsing modes, quoting, and commands. Commands can take arbitrary lists of arguments, and knowing when the statement ends is important. We'll cover this in the next section.

1.4.4 Statement termination

In PowerShell, there are two statement terminator characters: the semicolon (;) and (sometimes) the newline. Why is a newline a statement separator only *sometimes*? The rule is that if the previous text is a syntactically complete statement, a newline is considered to be a statement termination. If it isn't complete, the newline is treated like any other whitespace. This is how the interpreter can determine when a command or expression crosses multiple lines. For example, in the following

```
PS (1) > 2 +
>> 2
>>
4
PS (2) >
```

the sequence `2 +` is incomplete, and the interpreter prompts you to enter more text. (This is indicated by the nest prompt characters, >>.) On the other hand, in the next sequence

```
PS (2) > 2
2
PS (3) > + 2
2
PS (4) >
```

the number `2` by itself is a complete expression, and the interpreter goes ahead and evaluates it. Likewise, `+ 2` is a complete expression and is also evaluated (`+` in this case is treated as the unary plus operator). From this, you can see that if the newline comes after the `+` operator, the interpreter will treat the two lines as a single expression. If the newline comes before the `+` operator, it'll treat the two lines as two individual expressions.

Most of the time, this mechanism works the way you expect, but sometimes you can receive some unanticipated results. Take a look at the following example:

```
PS (22) > $b = ( 2
>> + 2 )
>>
Missing closing ')' in expression.
At line:2 char:1
+ + <<<<  2 )
PS (23) >
```

This was a question raised by one of the PowerShell beta testers. They were surprised by this result and thought there was something wrong with the interpreter, but in fact, this isn't a bug. Here's what's happening.

Consider the following text:

```
> $b = (2 +
> 2)
```

It's parsed as `$b = (2 + 2)` because a trailing `+` operator is only valid as part of a binary operator expression. The sequence `$b = ( 2 +` can't be a syntactically complete statement, and the newline is treated as whitespace. On the other hand, consider the text

```
> $b = (2
> + 2)
```

In this case, `2` is a syntactically complete statement, and the newline is now treated as a line terminator. In effect, the sequence is parsed like `$b = (2 ; + 2 );` two complete statements. Because the syntax for a parenthetical expression is

```
( <expr> )
```

you get a syntax error—the interpreter is looking for a closing parenthesis as soon as it has a complete expression. Contrast this with using a *subexpression* instead of the parentheses alone:

```
>> $b = $(
>> 2
>> +2
>> )
>>
PS (24) > $b
2
2
```

Here the expression is valid because the syntax for subexpressions is

```
$( <statementList> )
```

How do you extend a line that isn't extensible by itself? This is another situation where you can use the backtick escape character. If the last character in the line is a backtick, then the newline will be treated as a simple breaking space instead of a newline:

```
PS (1) > Write-Output `
>> -inputobject `
>> "Hello world"
>>
Hello world
PS (2) >
```

Finally, one thing that surprises some people is that strings aren't terminated by a newline character. Strings can carry over multiple lines until a matching, closing quote is encountered:

```
PS (1) > Write-Output "Hello
>> there
>> how are
>> you?"
```

```
>>
Hello
there
how are
you?
PS (2) >
```

In this example, you see a string that extended across multiple lines. When that string was displayed, the newlines were preserved in the string.

The handling of end-of-line characters in PowerShell is another of the trade-offs that kept PowerShell useful as a shell. Although the handling of end-of-line characters is a bit strange compared to non-shell languages, the overall result is easy for most people to get used to.

1.4.5 Comment syntax in PowerShell

Every computer language has some mechanism for annotating code with expository comments. Like many other shells and scripting languages, PowerShell comments begin with a number sign (`#`) symbol and continue to the end of the line. The `#` character must be at the beginning of a token for it to start a comment. Here's an example that illustrates what this means:

```
PS (1) > echo hi#there
hi#there
```

In this example, the number sign is in the middle of the token `hi#there` and isn't treated as the starting of a comment. In the next example, there's a space before the number sign:

```
PS (2) > echo hi #there
hi
```

Now the `#` is treated as starting a comment and the following text isn't displayed. It can be preceded by characters other than a space and still start a comment. It can be preceded by any statement-terminating or expression-terminating character like a bracket, brace, or semicolon, as shown in the next couple of examples:

```
PS (3) > (echo hi)#there
hi
PS (4) > echo hi;#there
hi
```

In both of these examples, the `#` symbol indicates the start of a comment.

Finally, you need to take into account whether you're in expression mode or command mode. In command mode, as shown in the next example, the `+` symbol is included in the token `hi+#there`:

```
PS (5) > echo hi+#there
hi+#there
```

In expression mode it's parsed as its own token. Now the `#` indicates the start of a comment, and the overall expression results in an error:

```
PS (6) > "hi"+#there
You must provide a value expression on the right-hand side of the '+'
operator.
At line:1 char:6
+ "hi"+ <<<< #there
```

The # symbol is also allowed in function names:

```
PS (3) > function hi#there { "Hi there" }
PS (4) > hi#there
Hi there
```

The reason for allowing the # in the middle of tokens was to make it easy to accommodate path providers that used # as part of their path names. People conventionally include a space before the beginning of a comment, and this doesn't appear to cause any difficulties.

MULTILINE COMMENTS

In PowerShell version 2, *multiline* comments were introduced, primarily to allow you to embed inline help text in scripts and functions. A multiline comment begins with <# and ends with #>. Here's an example:

```
<#
 This is a comment
    that spans
 multiple lines
#>
```

This type of comment need not span multiple lines;you can use this notation to add a comment preceding some code:

```
PS {2) > <# a comment #> "Some code"
Some code
PS {3) >
```

In this example, the line is parsed, the comment is read and ignored, and the code after the comment is executed.

One of the things this type of comment allows you to do is easily embed chunks of preformatted text in functions and scripts. The PowerShell help system takes advantage of this feature to allow functions and scripts to contain *inline documentation* in the form of special comments. These comments are automatically extracted by the help system to generate documentation for the function or script. You'll learn how the comments are used by the help subsystem in chapter seven.

Now that you have a good understanding of the basic PowerShell syntax, let's look at how your commands are executed by the PowerShell execution engine. We'll start with the pipeline.

1.5 How the pipeline works

A pipeline is a series of commands separated by the pipe operator (|), as shown in figure 1.5. In some ways, the term *production line* better describes pipelines in PowerShell. Each

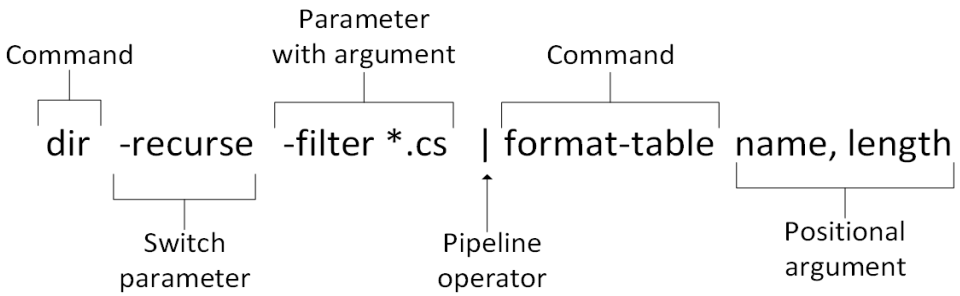


Figure 1.5 Anatomy of a pipeline

command in the pipeline receives an object from the previous command, performs some operation on it, and then passes it along to the next command in the pipeline.

> **NOTE** This, by the way, is the great PowerShell Heresy. All previous shells passed strings only through the pipeline. Many people had difficulty with the notion of doing anything else. Like the character in *The Princess Bride*, they'd cry "Inconceivable!" And we'd respond, "I do not think that word means what you think it means."

All of the command categories take parameters and arguments. In the following example

```
Get-ChildItem -Filter *.dll -Path c:\windows -Recurse
```

`-Filter` is a parameter that takes one argument, `*.dll`. The string "c:\windows" is the argument to the positional parameter `-Path`.

Next we'll discuss the signature characteristic of pipelines—streaming behavior.

1.5.1 Pipelines and streaming behavior

Streaming behavior occurs when objects are processed one at a time in a pipeline. This is one of the characteristic behaviors of shell languages. In stream processing, objects are output from the pipeline as soon as they become available. In more traditional programming environments the results are returned only when the entire result set has been generated—the first result and the last result are returned at the same time. In a pipelined shell, the first result is returned as soon as it's available and subsequent results return as they also become available. This flow is illustrated in figure 1.6.

At the top of figure 1.5 you see a PowerShell command pipeline containing four commands. This command pipeline is passed to the PowerShell parser, which figures out what the commands are, what the arguments and parameters are, and how they should be bound for each command. When the parsing is complete, the pipeline processor begins to sequence the commands. First it runs the `begin` clause of each of the commands, once, in sequence from first to last. After all the `begin` clauses have been run, it runs the `process` clause in the first command. If the command generates one or more objects, the pipeline processor passes these objects, one at a time, to the sec-

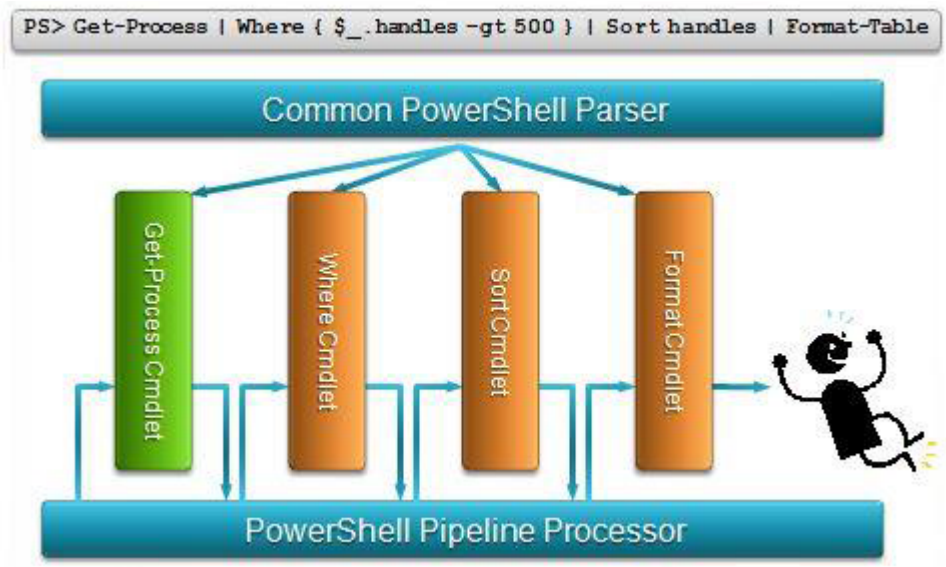


Figure 1.6 How objects flow through a pipeline one at a time. A common parser constructs each of the command objects and then starts the pipeline processor, stepping each object through all stages of the pipeline.

ond command. If the second command also emits an object, this object is passed to the third command, and so on.

When processing reaches the end of the pipeline, any objects emitted are passed back to the PowerShell host. The host is then responsible for any further processing.

This aspect of streaming is important in an interactive shell environment, because you want to see objects as soon as they're available. The next example shows a simple pipeline that traverses through C:\Windows looking for all of the DLLs whose names start with the word "system":

```
dir -recurse -filter *.dll | where Name -match "system.*dll"

            Directory: C:\windows\assembly\GAC_32\System.EnterpriseServices\
[CA]2.0.0.0__b03f5f7f11d50a3a


Mode         LastWriteTime Length Name
----         ------------- ------ ----
-a--- 27/05/2014     05:39 113664 System.EnterpriseServices.Wrapper.dll


    Directory: C:\windows\assembly\GAC_32\System.Printing\
[CA]3.0.0.0__31bf3856ad364e35


Mode         LastWriteTime Length Name
----         ------------- ------ ----
-a--- 03/08/2013     05:41 372736 System.Printing.dll
```

With streaming behavior, as soon as the first file is found, it's displayed. Without streaming, you'd have to wait until the entire directory structure has been searched before you'd see any results.

In most shell environments streaming is accomplished by using separate processes for each element in the pipeline. In PowerShell, which only uses a single process (and a single thread as well), streaming is accomplished by splitting cmdlets into three clauses: `BeginProcessing`, `ProcessRecord`, and `EndProcessing`. In a pipeline, the `BeginProcessing` clause is run for all cmdlets in the pipeline. Then the `ProcessRecord` clause is run for the first cmdlet. If this clause produces an object, that object is passed to the `ProcessRecord` clause of the next cmdlet in the pipeline, and so on. Finally, the `EndProcessing` clauses are all run. (We cover this sequencing again in more detail in chapter five, which is about scripts and functions, because they can also have these clauses.)

1.5.2 Parameters and parameter binding

Now let's talk about more of the details involved in binding parameters for commands. *Parameter binding* is the process in which values are bound to the parameters on a command. These values can come from either the command line or the pipeline. Here's an example of a parameter argument being bound from the command line:

```
PS (1) > Write-Output 123
123
```

And here's the same example where the parameter is taken from the input object stream:

```
PS (2) > 123 | Write-Output
123
```

The binding process is controlled by declaration information on the command itself. Parameters can have the following characteristics: they are either mandatory or optional, they have a type to which the formal argument must be convertible, and they can have attributes that allow the parameters to be bound from the pipeline. Table 1.2 describes the actual steps in the binding process.

Table 1.2 Steps in the parameter binding process

Binding step	Description
1. Bind all named parameters.	Find all unquoted tokens on the command line that start with a dash. If the token ends with a colon, an argument is required. If there's no colon, look at the type of the parameter and see if an argument is required. Convert the type of actual argument to the type required by the parameter, and bind the parameter.
2. Bind all positional parameters.	If there are any arguments on the command line that haven't been used, look for unbound parameters that take positional parameters and try to bind them.

Table 1.2 Steps in the parameter binding process *(continued)*

Binding step	Description
3. Bind from the pipeline by value with exact match.	If the command isn't the first command in the pipeline and there are still unbound parameters that take pipeline input, try to bind to a parameter that matches the type exactly.
4. If not bound, then bind from the pipe by value with conversion.	If the previous step failed, try to bind using a type conversion.
5. If not bound, then bind from the pipeline by name with exact match.	If the previous step failed, look for a property on the input object that matches the name of the parameter. If the types exactly match, bind the parameter.
6. If not bound, then bind from the pipeline by name with conversion.	If the input object has a property whose name matches the name of a parameter, and the type of the property is convertible to the type of the parameter, bind the parameter.

As you can see, this binding process is quite involved. In practice, the parameter binder almost always does what you want—that's why a sophisticated algorithm is used. Sometimes you'll need to understand the binding algorithm to get a particular behavior. PowerShell has built-in facilities for debugging the parameter-binding process that can be accessed through the `Trace-Command` cmdlet. Here's an example showing how to use this cmdlet:

```
Trace-Command -Name ParameterBinding  -Option All `
-Expression { 123 | Write-Output } -PSHost
```

In this example, you're tracing the expression in the braces—that's the expression:

```
123 | Write-Output
```

This expression pipes the number 123 to the cmdlet `Write-Output`. The `Write-Output` cmdlet takes a single mandatory parameter `-InputObject`, which allows pipeline input by value. The tracing output is long but fairly self-explanatory, and we haven't included it here. This is something you should experiment with to see how it can help you figure out what's going on in the parameter-binding process.

And now for the final topic in this chapter: formatting and output. The formatting and output subsystem provides the magic that lets PowerShell figure out how to display the output of the commands you type.

1.6 Formatting and output

One of the issues people new to PowerShell face is the formatting system. As a general rule we run commands and depend on the system to figure out how to display the results. We'll use commands such as `Format-Table` and `Format-List` to give general guidance on the shape of the display, but no specific details. Let's dig in now and see how this all works.

PowerShell is a type-based system. Types are used to determine how things are displayed, but normal objects don't usually know how to display themselves. PowerShell deals with this by including formatting information for various types of objects as part of the extended type system. This extended type system allows PowerShell to add new behaviors to existing .NET objects or extend the formatting system to cope with new types you have created. The default formatting database is stored in the PowerShell install directory, which you can get to by using the `$PSHOME` shell variable. Here's a list of the files that were included as of this writing:

```
PS (1) > dir $PSHOME/*format* | Format-Table name

Name
----
Certificate.format.ps1xml
Diagnostics.Format.ps1xml
DotNetTypes.format.ps1xml
Event.Format.ps1xml
FileSystem.format.ps1xml
Help.format.ps1xml
HelpV3.format.ps1xml
PowerShellCore.format.ps1xml
PowerShellTrace.format.ps1xml
Registry.format.ps1xml
WSMan.Format.ps1xml
```

The naming convention helps users figure out the purpose of files. (The others should become clear after reading the rest of this book.) These files are XML documents that contain descriptions of how each type of object should be displayed.

TIP These files are digitally signed by Microsoft. Do NOT alter them under any circumstances. You'll break things if you do.

These descriptions are fairly complex and somewhat difficult to write. It's possible for end users to add their own type descriptions, but that's beyond the scope of this chapter. The important thing to understand is how the formatting and outputting commands work together.

1.6.1 The formatting cmdlets

Display of information is controlled by the type of the objects being displayed, but the user can choose the "shape" of the display by using the `Format-*` commands:

```
PS (5) > Get-Command Format-* | Format-Table name

Name
----
Format-Custom
Format-List
Format-Table
Format-Wide
```

By *shape*, we mean things such as a table or a list. Here's how they work. The `Format-Table` cmdlet formats output as a series of columns displayed across your screen:

```
PS (1) > Get-Item c:\ | Format-Table

        Directory:

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d--hs         30/03/2015     20:04            C:\
```

PowerShell 5.0 automatically derives the on–screen positioning from the first few objects through the pipeline – effectively an automatic `-Autosize` parameter. This change was introduced because `-Autosize` is a blocking parameter that caused huge amounts of data to be stored in memory until all objects were available.

Format-Table autosize parameter

In PowerShell 1.0 through 4.0 `Format-Table` tries to use the maximum width of the display and guesses at how wide a particular field should be. This allows you to start seeing data as quickly as possible (streaming behavior) but doesn't always produce optimal results. You can achieve a better display by using the `-AutoSize` switch, but this requires the formatter to process every element before displaying any of them, and this prevents streaming. PowerShell has to do this to figure out the best width to use for each field. The result in this example looks like this:

```
PS (3) > Get-Item c:\ | Format-Table -AutoSize

        Directory:

Mode         LastWriteTime Length Name
----         ------------- ------ ----
d--hs 30/03/2015     20:04        C:\
```

In practice, the default layout when streaming is good and you don't need to use `-Autosize`, but sometimes it can help make things more readable.

The `Format-List` command displays the elements of the objects as a list, one after the other:

```
PS (2) > Get-Item c:\ | Format-List

        Directory:

Name           : C:\
CreationTime   : 22/08/2013 14:31:02
LastWriteTime  : 30/03/2015 20:04:20
LastAccessTime : 30/03/2015 20:04:20
```

If there's more than one object to display, they'll appear as a series of lists. This is usually the best way to display a large collection of fields that won't fit well across the screen.

The `Format-Wide` cmdlet is used when you want to display a single object property in a concise way. It'll treat the screen as a series of columns for displaying the same information.

```
PS (1) > Get-Process -Name s* | Format-Wide -Column 8 id

1372     640     516     1328    400     532     560     828
876      984     1060    1124    4
```

In this example, you want to display the process IDs of all processes whose names start with "s" in eight columns. This formatter allows for dense display of information.

The final formatter is `Format-Custom`, which displays objects while preserving the basic structure of the object. Because most objects have a structure that contains other objects, which in turn contain other objects, this can produce extremely verbose output. Here's a small part of the output from the `Get-Item` cmdlet, displayed using `Format-Custom`:

```
PS (10) > Get-Item c:\ | Format-Custom -Depth 1
v
class DirectoryInfo
{
  PSPath = Microsoft.PowerShell.Core\FileSystem::C:\
  PSParentPath =
  PSChildName = C:\
  PSDrive =
    class PSDriveInfo
    {
      CurrentLocation =
      Name = C
      Provider = Microsoft.PowerShell.Core\FileSystem
      Root = C:\
      Description = C_Drive
      Credential = System.Management.Automation.PSCredential
    }
```

The full output is considerably longer, and notice that we've told it to stop walking the object structure at a depth of 1. You can imagine how verbose this output can be! Why have this cmdlet? Mostly because it's a useful debugging tool, either when you're creating your own objects or for exploring the existing objects in the .NET class libraries.

1.6.2 The outputter cmdlets

Now that you know how to format something, how do you output it? You don't have to worry because, by default, things are automatically sent to (can you guess?) `Out-Default`.

Note that the following three examples do exactly the same thing:

```
dir | Out-Default
dir | Format-Table
dir | Format-Table | Out-Default
```

This is because the formatter knows how to get the default outputter, the default outputter knows how to find the default formatter, and the system in general knows how to find the defaults for both. The Möbius strip of subsystems!

As with the formatters, there are several outputter cmdlets available in PowerShell out of the box. You can use the `Get-Command` command to find them:

```
PS (1) > Get-Command Out-* | Format-Wide -Column 3

Out-Default       Out-File       Out-GridView
Out-Host          Out-Null       Out-Printer
Out-String
```

Here we have a somewhat broader range of choices. We've already talked about `Out-Default`. The next one we'll talk about is `Out-Null`. This is a simple outputter; anything sent to `Out-Null` is discarded. This is useful when you don't care about the output for a command; you want the side effect of running the command.

> **NOTE** Piping to `Out-Null` is the equivalent to doing redirecting to `$null` but invokes the pipeline and can be up to forty times slower than redirecting to `$null`.

Next we have `Out-File`. Instead of sending the output to the screen, this command sends it to a file. (This command is also used by I/O redirection when doing output to a file.) In addition to writing the formatted output, `Out-File` has several flags that control how the output is written. The flags include the ability to append to a file instead of replacing it, to force writing to read-only files, and to choose the output encodings for the file. This last item is the trickiest one. You can choose from a number of the text encodings supported by Windows. Here's a trick—enter the command with an encoding that you know doesn't exist:

```
PS (9) > out-file -encoding blah
Out-File : Cannot validate argument on parameter 'Encoding'. The argument
     "blah" doesn't belong to the set
     "unknown,string,unicode,bigendianunicode,
utf8,utf7,utf32,ascii,default,oem" specified by the ValidateSet attribute.
Supply an argument found in the set and then try the command again.
```

You can see in the error message that all the valid encoding names are displayed. If you don't understand what these encodings are, don't worry about it, and let the system use its default value.

> **NOTE** Where you're likely to run into problems with output encoding (or input encoding for that matter) is when you're creating files that are going to be read by another program. These programs may have limitations on what encodings they can handle, particularly older programs. To find out more about file encodings, search for "file encodings" on http://msdn.microsoft.com. MSDN contains a wealth of information on this topic. Chapter five also contains additional information about working with file encodings in PowerShell.

The `Out-Printer` cmdlet doesn't need much additional explanation; it routes its text-only output to the default printer instead of to a file or to the screen.

The `Out-Host` cmdlet is a bit more interesting—it sends its output back to the host. This has to do with the separation in PowerShell between the interpreter or engine, and the application that hosts that engine. The host application has to implement a special set of interfaces to allow `Out-Host` to render its output properly. (We see this used in PowerShell versions 2.0 to 5.0 which include two hosts: the console host and the Integrated Scripting Environment (ISE).)

NOTE `Out-Default` delegates the actual work of outputting to the screen to `Out-Host`.

The last output cmdlet to discuss is `Out-String`. This one's a bit different. All the other cmdlets terminate the pipeline. The `Out-String` cmdlet formats its input and sends it as a string to the next cmdlet in the pipeline. Note that we said *string*, not *strings*. By default, it sends the entire output as a single string. This isn't always the most desirable behavior—a collection of lines is usually more useful—but at least once you have the string, you can manipulate it into the form you want. If you do want the output as a series of strings, use the `-Stream` switch parameter. When you specify this parameter, the output will be broken into lines and streamed one at a time.

Note that this cmdlet runs somewhat counter to the philosophy of PowerShell; once you've rendered the object to a string, you've lost its structure. The main reason for including this cmdlet is for interoperation with existing APIs and external commands that expect to deal with strings. If you find yourself using `Out-String` a lot in your scripts, stop and think if it's the best way to attack the problem.

PowerShell version 2 introduced one additional output command: `Out-GridView`. As you might guess from the name, this command displays the output in a grid, but rather than rendering the output in the current console window, a new window is opened with the output displayed using a sophisticated grid control (see figure 1.7). The underlying grid control used by `Out-GridView` has all the features you'd expect from a modern Windows interface: columns can be reordered by dragging and dropping them, and the output can be sorted by clicking a column head. This control also introduces sophisticated filtering capabilities. This filtering allows you to drill into a dataset without having to rerun the command.

That's it for the basics: commands, parameters, pipelines, parsing, and presentation. You should now have a sufficient foundation to start moving on to more advanced topics in PowerShell.

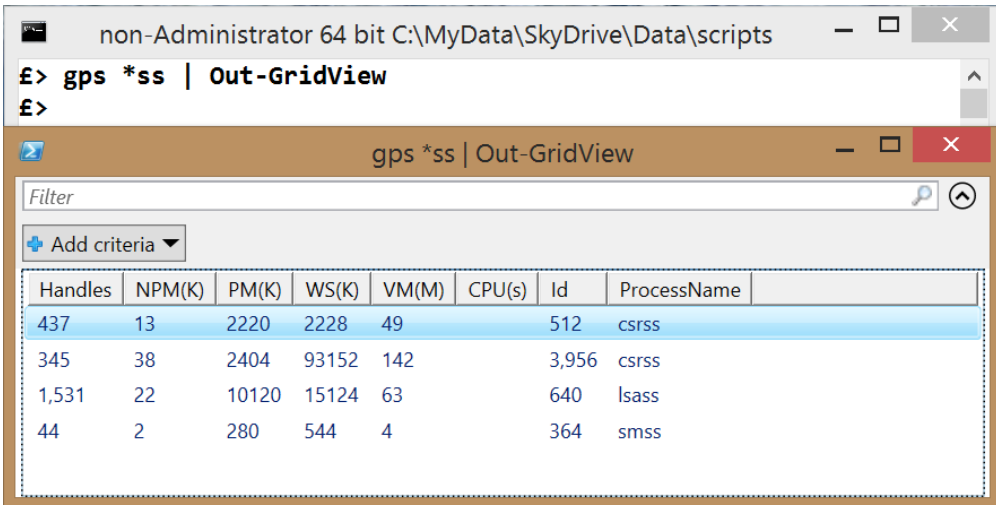
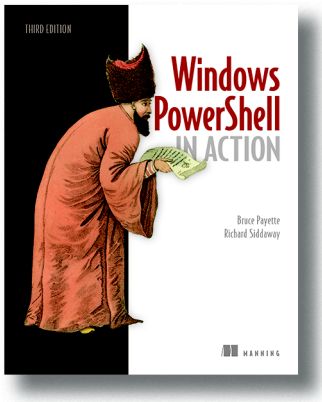


Figure 1.7 Displaying output with `Out-GridView`

1.7 Summary

- PowerShell is Microsoft's command-line/scripting environment that's at the center of Microsoft server and application management technologies. Microsoft's most important server products, including Exchange, Active Directory, and SQL Server, now use PowerShell as their management layer.
- PowerShell incorporates object-oriented concepts into a command-line shell using the .NET object model as the base for its type system, but can also access other object types like WMI.
- Shell operations, like navigation and file manipulation in PowerShell, are similar to what you're used to in other shells.
- The way to get help about things in PowerShell is to use the `Get-Help` command.
- PowerShell has a full range of calculation, scripting, and text processing capabilities.
- PowerShell supports a comprehensive set of remoting features to allow you to do scripted automation of large collections of computers.
- PowerShell has a number of command types, including cmdlets, functions, script commands, and native commands, each with slightly different characteristics.
- PowerShell supports an elastic syntax—concise on the command line and complete in scripts. Aliases are used to facilitate elastic syntax.
- PowerShell parses scripts in two modes: expression mode and command mode, which is a critical point to appreciate when using PowerShell.
- The PowerShell escape character is a backtick (`), not a backslash.

- PowerShell supports both double quotes and single quotes; variable and expression expansion is done in double quotes, but not in single quotes.
- Line termination is handled specially in PowerShell because it's a command language.
- PowerShell has two types of comments: line comments that begin with # and block comments that start with <# and end with #>. The block comment notation was introduced in PowerShell version 2 with the intent of supporting inline documentation for scripts and functions.
- PowerShell uses a sophisticated formatting and outputting system to determine how to render objects without requiring detailed input from the user.



Windows PowerShell in Action, Third Edition is a completely revised edition of the bestselling book on PowerShell. It keeps the same crystal-clear introduction to PowerShell as the last edition and adds extensive coverage of v3, v4, and v5 features such as PowerShell Workflows, Desired State Configuration, PowerShell classes and the PowerShell APIs, new error handling and debugging features. It includes full chapters on these topics and also covers new language elements and operators, PowerShell remoting, CIM, events, working with data such as XML and flat files, The Second Edition's coverage of batch scripting and string processing, COM, WMI, and .NET have all been significantly revised and expanded. The book includes many popular usage scenarios and is rich in interesting examples that will spark your imagination. This is the definitive book on PowerShell - whichever version you use.

Windows PowerShell transformed the way administrators and developers interact with Windows. PowerShell, an elegant dynamic language from Microsoft, lets you script administrative tasks and control Windows from the command line. Because it's a full-featured, first-class Windows programming language, programmers and power-users can now do things in a shell that previously required VB, VBScript, or C#.

What's inside:

- Writing modules and scripts
- PowerShell Workflows
- Desired State Configuration
- PowerShell background jobs
- PowerShell classes
- Programming APIs, pipelines and ISE extensions
- Error handling and debugging

Written for developers and administrators with intermediate level scripting knowledge. No prior experience with PowerShell is required.

PowerShell Remoting

Remoting – the ability to connect to and administer remote machines – is the foundation of administering your Windows environment with PowerShell. Being able to work with tens, hundreds, or even thousands of machines, brings great flexibility and time savings. You need a good understanding of the theoretical and practical aspects of remoting to get the best from it. This chapter provides that mixture of theory and practice that'll help you learn how to use and troubleshoot PowerShell remoting.

Chapter 10 from *PowerShell in Depth, Second Edition* by Don Jones, Jeffrey Hicks, and Richard Siddaway

PowerShell Remoting

This chapter covers

- Outlining Remoting technologies and protocols
- Configuring and securing Remoting endpoints
- Exploring Remoting scenarios
- Using implicit Remoting

Remoting was one of the major new technologies introduced in PowerShell v2 and in the broader Windows Management Framework v2 (WMF v2), of which PowerShell is a part. With v4, Microsoft has continued to invest in this important foundational technology. Most Windows machines, client or server, can be used as the local or remote machine—that is, you can create remote connections *to* them and you can create remote connections *from* them. The one exception is Windows RT—you can only remote from machines running that version.

NOTE There's very little difference between Remoting in PowerShell v3 and v4. Unless we state otherwise, everything in this chapter applies equally to PowerShell v3 and v4.

Remoting is a complex technology, and we'll do our best to explore it as thoroughly as possible. But some uses for Remoting are outside the purview of an administrator: Programming custom-constrained runspaces, for example, requires software development skills that are outside the scope of this book.

> **NOTE** Everything in this chapter focuses on PowerShell v4 and v3, but the majority of the material also applies to v2. The three versions of the shell can talk to each other via Remoting—that is, a v2 shell can connect to a v3 or v4 shell, and vice versa. PowerShell Remoting between v3 and v4 works seamlessly.

10.1 The many forms of remote control

The first thing we need to clear up is the confusion over the word *remote*. PowerShell v2 offers two means for connecting to remote computers:

- Cmdlets, which have their own `-ComputerName` parameter. They use their own proprietary communications protocols, most often DCOM or RPC, and are generally limited to a single task. They don't use PowerShell Remoting (with a couple of exceptions that we'll cover later in this chapter).
- Cmdlets that specifically use the Remoting technology: `Invoke-Command`, anything with the `-PSSession` noun, and a few others that we'll cover in this chapter.

In this chapter, we're focusing exclusively on the second group. The nice thing about it is that any cmdlet—whether it has a `-ComputerName` parameter or not—can be used through Remoting.

> **NOTE** PowerShell v3 introduced another type of Remoting: CimSessions. These are analogous to PowerShell Remoting sessions and also work over WSMAN by default. They are covered in detail in chapter 39.

What exactly is Remoting? It's the ability to send one or more commands over the network to one or more remote computers. The remote computers execute the commands using their own local processing resources (meaning the command must exist and be loaded on the remote computers). The results of the commands—like all PowerShell commands—are objects, and PowerShell *serializes* them into XML. The XML is transmitted across the network to the originating computer, which *deserializes* them back into objects and puts them into the pipeline. The serialize/deserialize part of the process is crucial, because it offers a way to get complex data structures into a text form that's easily transmitted over a network. Don't overthink the serializing thing, though: It's not much more complicated than piping the results of a command to `Export-CliXML` and then using `Import-CliXML` to load the results back into the pipeline as objects. It's almost exactly like that, in fact, with the additional benefit of having Remoting taking care of getting the data across the network.

PowerShell Web Access (PWA—Microsoft uses PSWA but the PowerShell community prefers PWA as an acronym) was introduced in Windows Server 2012 and enhanced in Windows Server 2012 R2. PWA is covered in appendix B. PWA uses PowerShell Remoting "under the hood." It's best to consider PWA as a presentation layer superimposed on PowerShell Remoting, which is why we don't cover it here.

10.2 Remoting overview

Terminology gets a lot of people messed up when it comes to Remoting, so let's get that out of the way.

- WSMAN is the network protocol used by PowerShell Remoting. It stands for Web Services for Management, and it's more or less an industry-standard protocol. You can find implementations on platforms other than Windows, although they're not yet widespread. WSMAN is a flavor of good-old HTTP, the same protocol your web browser uses to fetch web pages from a web server.
- *Windows Remote Management*, or *Win*RM, is a Microsoft service that implements the WSMAN protocol and that handles communications and authentication for connections. WinRM is designed to provide communications services for any number of applications; it isn't exclusive to PowerShell. When WinRM receives traffic, that traffic is tagged for a specific application—such as PowerShell—and WinRM takes care of getting the traffic to that application as well as accepting any replies or results that the application wants to send back.
- *Remoting* is a term applied to PowerShell's use of WinRM. Therefore, you can't do "Remoting" with anything other than PowerShell—although other applications could certainly have their own specific uses for WinRM.

One of the features introduced in PowerShell v3 was a set of Common Information Model (CIM) cmdlets. Over time, they'll replace the legacy Windows Management Instrumentation (WMI) cmdlets that have been in PowerShell since v1, although for now the WMI and CIM cmdlets live side by side and have a lot of overlapping functionality. Both sets of cmdlets use the same underlying WMI data repository; one of the primary differences between the two sets is in how they communicate over the network. The WMI cmdlets use remote procedure calls (RPCs), whereas the CIM cmdlets use WinRM. The CIM cmdlets *aren't using Remoting*—they provide their own utilization of WinRM (more details in chapter 39). We point this out only as an example of how confusing the terminology can be. In the end, you don't have to worry about it all the time, but when it comes to troubleshooting you'll definitely need to understand which parts are using what.

Now for a bit more terminology, this time diving into some of the specific implementation details:

- An *endpoint* is a particular configuration item in WinRM. An endpoint represents a specific application for which WinRM can receive traffic, along with a group of settings that determine how the endpoint behaves. It's entirely possible for a single application, like PowerShell, to have multiple endpoints set up. Each endpoint might be for a different purpose and might have different security, network settings, and so forth associated with it.
- A *listener* is another configuration item in WinRM, and it represents the service's ability to accept incoming network traffic. A listener is configured to have a TCP port number, is configured to accept traffic on one or more IP addresses, and so

forth. A listener also is set up to use either HTTP or HTTPS; if you want to be able to use both protocols, then you must have two listeners set up.

10.2.1 Authentication

WinRM has two levels of authentication: machine-level and user-level. User-level authentication involves the delegation of your logon credentials to the remote machine that you've connected to. The remote machine can undertake any tasks you've specified using your identity, meaning you'll be able to do whatever you have permission to do and no more. By default, the remote machine can't delegate your credentials to any other machines—which can lead to a problem called "the second hop" where you attempt, and usually fail, to perform an action on a third machine from within your remote session. We'll deal with that later in the chapter.

Remoting also supports machine-level authentication. In other words, when you connect to a remote machine, your computer must trust that machine. Trust normally comes through mutual membership in an Active Directory domain, although it can also be manually configured in a number of ways. The practical upshot is that your computer will refuse to connect to any remote machine that it doesn't know and trust. That can create complications for some environments where the machines aren't all in the same domain, requiring additional configuration to get Remoting to work.

10.2.2 Firewalls and security

One of the joys of Remoting is that it operates over a single port: 5985 for HTTP and 5986 for HTTPS, by default, although you can reconfigure them if you like. It's therefore easy to set up firewall exceptions that permit Remoting traffic.

Some organizations, mainly those with very tight network security, may have some trepidation about enabling Remoting and its firewall exceptions. Our only advice is to "get over it." Remoting is now a foundational, mandatory technology in Windows. Not allowing it would be like not allowing Ethernet. Without Remoting, you'll find that many of Windows' administrative tools and features simply don't work, especially in Windows Server 2012 and later.

Remoting is more secure than what we've used in the past for these tasks. It authenticates, by default, using the Kerberos protocol, which never transmits passwords on the network (encrypted or otherwise). Remoting uses a single, customizable port, rather than the thousands required by older protocols like RPCs. WinRM and Remoting have a huge variety of configuration settings that let you control who can use it, how much they can use it, and soon.

10.3 Using Remoting

In the next few sections, we're going to walk you through the complete process of setting up and using Remoting. This will specifically cover the "easy scenario," meaning that both your computer and the remote computer are in the same Active Directory domain. After we go over these basics, we'll dive into all of the other scenarios that you might have to configure.

10.3.1 Enabling Remoting

Remoting needs to be enabled on any machine that will receive connections, which can include computers running either the server or a client version of the Windows operating system. Windows Server 2012, and later versions of the server OS, has Remoting enabled by default though client version of Windows don't. The easy way to set up Remoting is to run `Enable-PSRemoting` (you need to be running PowerShell with elevated privileges). You could perform all of the steps manually but we don't recommend it.

> **NOTE** You have to set up PowerShell Remoting on the machine itself. You can't do it remotely. Having it enabled by default is a good step forward—one less configuration step on new machines.

The `Enable-PSRemoting` command performs several tasks:

- Starts (or restarts, if it's already started) the WinRM service.
- Sets the WinRM service to start automatically from now on.
- Creates a WinRM listener for HTTP traffic on port 5985 for all local IP addresses.
- Creates a Windows Firewall exception for the WinRM listener. Note that this will fail on client versions of Windows if any network cards are configured to have a type of "Public," because the firewall will refuse to create new exceptions on those cards. If this happens, change the network card's type to something else (like "Work" or "Private," as appropriate—Windows 8/2012 provides the `Set-NetConnectionProfile` cmdlet for this task) and run `Enable-PSRemoting` again. Alternately, if you know you have some Public network cards, add the `-SkipNetworkProfileCheck` parameter to `Enable-PSRemoting`. Doing so will successfully create a Firewall exception that allows incoming Remoting traffic only from the computer's local subnet.

The command will also set up one or more of these endpoints:

- Microsoft.PowerShell
- Microsoft.PowerShell32
- Microsoft.ServerManager (for Server Manager)
- Microsoft.Windows.ServerManagerWorkflows (for Server Manager workflows)
- Microsoft.PowerShell.Workflow (for PowerShell workflow)

You'll be prompted several times as the command runs; be sure to reply "Y" for "Yes" so that each step can complete properly. You can avoid the prompts by using the `-Force` parameter.

Discovering WSMAN endpoints

You can find the endpoints that exist on your system through the WSMAN provider. The configuration information is exposed through a PowerShell drive—WSMAN:

```
PS C:\> dir WSMan:\localhost\Plugin
   WSManConfig: Microsoft.WSMan.Management\WSMan::localhost\Plugin
Type       Keys                            Name
----       ----                            ----
Container  {Name=Event Forwarding Plugin}  Event Forwarding Plugin
Container  {Name=microsoft.powershell}     microsoft.powershell
Container  {Name=microsoft.powershell...   microsoft.powershell.workflow
Container  {Name=microsoft.powershell32}   microsoft.powershell32
Container  {Name=WMI Provider}             WMI Provider
```

This example is taken from a Windows 8.1 64-bit machine. You'll notice what appears to be two endpoints that we haven't mentioned:

- Event Forwarding Plugin
- WMI Provider

Windows servers have another apparent endpoint that we haven't mentioned: SEL Plugin.

The simple reason we haven't mentioned them is that they aren't Remoting endpoints as such. Their purpose is to provide WSMAN connectivity for other activities. Event forwarding and WMI are self-explanatory whereas SEL is for hardware management.

The WSMAN configurations that are purely for Remoting can be discovered by using `Get-PSSessionConfiguration`:

```
PS C:\> Get-PSSessionConfiguration | format-table Name, PSVersion -auto
Name                          PSVersion
----                          ---------
microsoft.powershell          4.0
microsoft.powershell.workflow 4.0
microsoft.powershell32        4.0
```

Table 10.1 illustrates some example endpoint configurations. On a 32-bit machine, the endpoint is referred to as PowerShell rather than PowerShell32.

Table 10.1 Example endpoint configurations. The table reports the "out-of-the-box" configuration. Any machine originally running PowerShell v3 that has been upgraded to PowerShell v4 will show the PowerShell version as 4.

	PowerShell version	PowerShell 32-bit	PowerShell 64-bit	Server Manager	Server Manager workflow	PowerShell workflow
Windows Server 2008 R2	2	Y	Y	Y		
Windows 7 64-bit	2	Y	Y			Y

Table 10.1 Example endpoint configurations. The table reports the "out-of-the-box" configuration. Any machine originally running PowerShell v3 that has been upgraded to PowerShell v4 will show the PowerShell version as 4. *(continued)*

	PowerShell version	PowerShell 32-bit	PowerShell 64-bit	Server Manager	Server Manager workflow	PowerShell workflow
Windows 8 32-bit client	3	Y				Y
Windows 8.1 64-bit client	4	Y	Y			Y
Windows Server 2012	3	Y	Y		Y	Y
Windows Server 2012 R2	4	Y	Y		Y	Y
Windows 7 client 32-bit stand-alone	2	Y				

In an enterprise you'll probably use Group Policy to configure Remoting. That approach has slightly a different outcome compared to using `Enable-PSRemoting`, as shown in table 10.2.

Table 10.2 The outcome when enabling Remoting through different mechanisms

	`Enable-PSRemoting`	Group Policy	Manually step-by-step
Set WinRM to autostart and start the service	Yes	Yes	Yes; use `Set-Service` and `Start-Service`.
Configure HTTP listener	Yes	You can configure autoregistration of listeners, but you can't create custom listeners.	Yes; use the Winrm command-line utility and WSMan: drive in PowerShell
Configure HTTPS listener	No	No	Yes; use the winrm command-line utility and WSMan: drive in PowerShell
Configure endpoints/session configurations	Yes	No	Yes; use `PSSession-Configuration` cmdlets
Configure Windows Firewall exception	Yes, but not on a Public network	Yes, but not on a Public network	Yes, but not on a Public network

10.3.2 1-to-1 Remoting

The most straightforward way to use Remoting is called *1-to-1 Remoting*, in which you essentially bring up an interactive PowerShell prompt on a remote computer. It's pretty simple, once Remoting is enabled on the remote machine:

```
PS C:\> Enter-PSsession -ComputerName Win8
[Win8]: PS C:\Users\Administrator\Documents>
```

> **NOTE** If you want to experiment with this, just use localhost as the computer name, once you've enabled Remoting on your computer. You'll be "remotely controlling" your local machine, but you'll get the full Remoting experience.

Notice how the PowerShell prompt changes to include the name of the computer you're now connected to. From here, it's almost exactly as if you were physically standing in front of that computer, and you can run any command that the remote machine contains. Keep these important caveats in mind:

- By default, when the PowerShell prompt contains any computer name (even localhost), you can't execute any other commands that initiate a Remoting connection. Doing so would create a "second hop," which won't work by default.
- You can't run any commands that start a graphical application. If you do so, the shell may appear to freeze; press Ctrl-C to end the process and regain control.
- You can't run any command program that has its own "shell" like nslookup or netsh—though you can run them as commands rather than interactively.
- You can only run scripts on the remote machine if its execution policy permits you to do so (we discuss that in chapter 17).
- You aren't connected to an interactive desktop session; your connection will be audited as a "network logon," much as if you were connecting to a file share on the remote machine. As a result of the connection type, Windows won't execute profile scripts, although you'll be connected to your profile home folder on the remote machine.
- Nothing you do will be visible by any other user who's connected to the same machine, even if they're interactively logged onto its desktop console. You can't run some application and have it "pop up" in front of the logged-on user.
- You must specify the computer's name as it appears in Active Directory or in your local Trusted Hosts list; you can't use IP addresses or DNS CNAME aliases unless they've been added to your Trusted Hosts list.

When you've finished with the remote machine, run `Exit-PSSession`. This will return you to your local prompt, close the connection to the remote machine, and free up resources on the remote machine. This will also happen automatically if you just close the PowerShell window.

```
[Win8]: PS C:\Users\Administrator\Documents> Exit-PSSession
PS C:\>
```

The way we've used `Enter-PSSession` will always connect to the remote machine's default PowerShell endpoint. On a 64-bit operating system, that'll be the 64-bit version of PowerShell. Later, we'll show you how to connect to other endpoints (remembering that `Enable-PSRemoting` will create multiple endpoints).

10.3.3 1-to-many Remoting

One-to-many Remoting, also known as fan-out Remoting, is a powerful technique that highlights the value of Remoting. You transmit a command (or a series of commands) to multiple remote computers. They each execute the command, serialize the results into XML, and send the results back to you. Your copy of PowerShell deserializes the XML into objects and puts them in the pipeline. For example, suppose you want to get a list of all processes whose names start with the letter "s," from two different computers:

```
PS C:\> Invoke-Command -ScriptBlock { Get-Process -name s* } -ComputerName
➥  localhost,win8

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName PSComputerName
-------  ------    -----      ----- -----   ------     -- ----------- --------------
    217      11     3200       7080    33     1.23    496 services    win8
     50       3      304        980     5     0.13    248 smss        win8
    315      16     2880       8372    46     0.03     12 spoolsv     win8
    472      36     8908      11540    60     0.31    348 svchost     win8
    306      12     2088       7428    36     0.19    600 svchost     win8
    295      15     2372       5384    29     0.61    636 svchost     win8
    380      15    17368      19428    55     0.56    728 svchost     win8
   1080      41    12740      25456   120     2.19    764 svchost     win8
    347      19     3892       8812    93     0.03    788 svchost     win8
    614      52    13820      18220  1129     2.28    924 svchost     win8
     45       4      508       2320    13     0.02   1248 svchost     win8
    211      18     9228       8408  1118     0.05   1296 svchost     win8
     71       6      804       3540    28     0.00   1728 svchost     win8
   2090       0      120        292     3    10.59      4 System      win8
    217      11     3200       7080    33     1.23    496 services    loca...
     50       3      304        980     5     0.13    248 smss        loca...
    315      16     2880       8372    46     0.03     12 spoolsv     loca...
    469      36     8856      11524    59     0.31    348 svchost     loca...
    306      12     2088       7428    36     0.19    600 svchost     loca...
    295      15     2372       5384    29     0.61    636 svchost     loca...
    380      15    17368      19428    55     0.56    728 svchost     loca...
   1080      41    12740      25456   120     2.19    764 svchost     loca...
    347      19     3892       8812    93     0.03    788 svchost     loca...
    607      49    13756      18132  1129     2.28    924 svchost     loca...
     45       4      508       2320    13     0.02   1248 svchost     loca...
    211      18     9228       8408  1118     0.05   1296 svchost     loca...
     71       6      804       3540    28     0.00   1728 svchost     loca...
   2089       0      120        292     3    10.59      4 System      loca...
```

The command is `Invoke-Command`. Its `-ScriptBlock` parameter accepts the commands (use semicolons to separate multiple commands) you want transmitted to the

remote machines; the -ComputerName parameter specifies the machine names. Alternatively, for longer commands a script block object could be created:

```
$sb = {Get-Process -Name s*}
Invoke-Command -ComputerName localhost,win8 -ScriptBlock $sb
```

As with Enter-PSSession, you must specify the computer's name as it appears in Active Directory or in your local Trusted Hosts list; you can't use IP addresses or DNS CNAME aliases unless they've been added to your Trusted Hosts list.

Notice anything interesting about the output? It contains an extra column named PSComputerName, which contains the name of the computer each result row came from. This is a handy way to separate, sort, group, and otherwise organize your results. This property is always added to the incoming results by PowerShell; if you'd rather not see the property in the output, add the -HideComputerName parameter to Invoke-Command. The property will still exist (and can be used for sorting and so forth), but it won't be displayed in the output by default.

As with Enter-PSSession, Invoke-Command will use the default PowerShell endpoint on the remote machine—which in the case of a 64-bit OS will be the 64-bit shell. We'll cover how to connect to a different endpoint later in this chapter.

By default, Invoke-Command will talk to only 32 computers at once. Doing so requires it to maintain a PowerShell instance in memory for each remote machine it's talking to; 32 is a number Microsoft came up with that seems to work well in a variety of situations. If you specify more than 32 computers, the extra ones will just queue up, and Invoke-Command will start working with them as the first 32 begin to complete. You can change the level of parallelism by using the command's -ThrottleLimit parameter, keeping in mind that higher numbers place a greater load on your computer but no extra load on the remote machines.

10.3.4 Remoting caveats

The data sent from a remote machine to your computer has to be packaged in a way that makes it easy to transmit over the network. Serialization and deserialization, which we've already mentioned, make it possible—but with some loss of functionality. For example, consider the type of object produced by Get-Service:

```
PS C:\> Get-Service | Get-Member

   TypeName: System.ServiceProcess.ServiceController

Name                      MemberType    Definition
----                      ----------    ----------
Name                      AliasProperty Name = ServiceName
RequiredServices          AliasProperty RequiredServices = ServicesDepe...
Disposed                  Event         System.EventHandler Disposed(Sy...
Close                     Method        System.Void Close(
Continue                  Method        System.Void Continue()
CreateObjRef              Method        System.Runtime.Remoting.ObjRef ...
Dispose                   Method        System.Void Dispose()
Equals                    Method        bool Equals(System.Object obj)
```

```
ExecuteCommand            Method       System.Void ExecuteCommand(int ...
GetHashCode               Method       int GetHashCode()
GetLifetimeService        Method       System.Object GetLifetimeService()
GetType                   Method       type GetType()
InitializeLifetimeService Method       System.Object InitializeLifetim...
Pause                     Method       System.Void Pause()
Refresh                   Method       System.Void Refresh()
Start                     Method       System.Void Start(), System.Voi...
Stop                      Method       System.Void Stop()
WaitForStatus             Method       System.Void WaitForStatus(Syste...
CanPauseAndContinue       Property     bool CanPauseAndContinue {get;}
CanShutdown               Property     bool CanShutdown {get;}
CanStop                   Property     bool CanStop {get;}
Container                 Property     System.ComponentModel.IContaine...
DependentServices         Property     System.ServiceProcess.ServiceCo...
DisplayName               Property     string DisplayName {get;set;}
MachineName               Property     string MachineName {get;set;}
ServiceHandle             Property     System.Runtime.InteropServices....
ServiceName               Property     string ServiceName {get;set;}
ServicesDependedOn        Property     System.ServiceProcess.ServiceCo...
ServiceType               Property     System.ServiceProcess.ServiceTy...
Site                      Property     System.ComponentModel.ISite Sit...
Status                    Property     System.ServiceProcess.ServiceCo...
ToString                  ScriptMethod System.Object ToString();
```

As you can see, these objects' members include several methods, which let you stop the service, pause it, and so on. Now consider that exact same kind of object retrieved, via Remoting, from a remote machine:

```
PS C:\> Invoke-Command -ComputerName win8 -ScriptBlock { Get-Service } |
➥  Get-Member

   TypeName: Deserialized.System.ServiceProcess.ServiceController

Name                 MemberType   Definition
----                 ----------   ----------
ToString             Method       string ToString(), string ToString(str...
Name                 NoteProperty System.String Name=AeLookupSvc
PSComputerName       NoteProperty System.String PSComputerName=win8
PSShowComputerName   NoteProperty System.Boolean PSShowComputerName=True
RequiredServices     NoteProperty Deserialized.System.ServiceProcess.Ser...
RunspaceId           NoteProperty System.Guid RunspaceId=00e784f7-6c27-4...
CanPauseAndContinue  Property     System.Boolean {get;set;}
CanShutdown          Property     System.Boolean {get;set;}
CanStop              Property     System.Boolean {get;set;}
Container            Property      {get;set;}
DependentServices    Property     Deserialized.System.ServiceProcess.Ser...
DisplayName          Property     System.String {get;set;}
MachineName          Property     System.String {get;set;}
ServiceHandle        Property     System.String {get;set;}
ServiceName          Property     System.String {get;set;}
ServicesDependedOn   Property     Deserialized.System.ServiceProcess.Ser...
ServiceType          Property     System.String {get;set;}
Site                 Property      {get;set;}
Status               Property     System.String {get;set;}
```

The methods (except for the universal `ToString()` method) are gone. That's because you're looking at a deserialized version of the object (it says so right in the `TypeName` at the top of the output), and the methods are stripped off. Essentially, you're getting a read-only, static version of the object.

This isn't necessarily a downside; serialization and the removal of methods doesn't occur until the remote commands finish executing and their output is being packaged for transmission. The objects are still "live" objects when they're on the remote computer, so you have to start them, stop them, pause them, or whatever on the remote machine. In other words, any "actions" you want to take must be part of the command you send to the remote machine for execution.

10.3.5 Remoting options

Both `Invoke-Command` and `Enter-PSSession` offer a few basic options for customizing their behavior.

ALTERNATE CREDENTIALS

By default, PowerShell delegates whatever credential you used to open the shell on your computer. That may not always be what you want, so you can specify an alternate username by using the `-Credential` parameter. You'll be prompted for the account's password, and that account will be used to connect to the remote machine (or machines) and run whatever commands you supply.

> **NOTE** In chapter 17, on PowerShell security, we discuss the `-Credential` parameter in more detail and offer other ways in which it can be used.

ALTERNATE PORT NUMBER

PowerShell defaults to using port 5985 for Remoting; you can change that when you set up WinRM listeners. You can also change your computer to use a different port when it initiates connections, which makes sense if you've changed the port your servers are listening to.

You'll find the port being listened to (the port on which traffic will be accepted) by examining your WSMan drive in PowerShell. Here's an example. (Note that your computer's listener ID will be different than the `Listener_1084132640` shown here, but you can find your ID by getting a directory listing of `WSMan:\localhost\Listener`.)

```
PS WSMan:\localhost\Listener\Listener_1084132640> ls

   WSManConfig:
Microsoft.WSMan.Management\WSMan::localhost\Listener\Listener_1084132640

Type            Name                             SourceOfValue   Value
----            ----                             -------------   -----
System.String   Address                                          *
System.String   Transport                                        HTTP
System.String   Port                                             5985
System.String   Hostname
System.String   Enabled                                          true
System.String   URLPrefix                                       wsman
```

```
System.String   CertificateThumbprint
System.String   ListeningOn_1638538265                    10.211.55.6
System.String   ListeningOn_1770022257                    127.0.0.1
System.String   ListeningOn_1414502903                    ::1
System.String   ListeningOn_766473143                     2001:0:4...
System.String   ListeningOn_86955851                      fdb2:2c2...
System.String   ListeningOn_1728280878                    fe80::5e...
System.String   ListeningOn_96092800                      fe80::98...
System.String   ListeningOn_2037253461                    fe80::c7...
```

Keep in mind that to work with the WSMAN PSDrive, you must be in an elevated PowerShell session. To change the port (using port 1000 as an example), type this:

```
PS C:\> Set-Item WSMan:\localhost\listener\*\port 1000
```

Now let's look at the client-side configuration, which tells your computer which port the server will be listening to:

```
PS WSMan:\localhost\Client\DefaultPorts> dir

   WSManConfig:
Microsoft.WSMan.Management\WSMan::localhost\Client\DefaultPorts

Type            Name                           SourceOfValue   Value
----            ----                           -------------   -----
System.String   HTTP                                           5985
System.String   HTTPS                                          5986
```

If you've set all of your servers to port 1000 (for example), then it makes sense to also reconfigure your clients so that they use that port by default:

```
PS C:\> Set-Item WSMan:\localhost\client\DefaultPorts\HTTP 1000
```

Alternately, both `Invoke-Command` and `Enter-PSSession` have a `-Port` parameter, which can be used to specify a port other than the one listed in the `DefaultPorts` configuration. That's useful if you have to use an alternate port for just one or two servers in your environment and don't want to change the client's defaults.

> **TIP** If you want to change default ports for your enterprise, we suggest you use Group Policy to push out these settings.

The default ports should only be changed if you have a good reason. If you do change the ports, make sure that your change is documented and applied across your enterprise (including firewalls) to avoid unnecessary troubleshooting efforts if Remoting connections fail.

USING SSL

If a server is configured with an HTTPS endpoint (which isn't the case after running `Enable-PSRemoting`; you have to set that up manually, which we'll get to later), then specify the `-UseSSL` parameter of `Invoke-Command` or `Enter-PSSession` to use the HTTPS port. That's port 5986 by default.

SENDING A SCRIPT INSTEAD OF A COMMAND

Our example of `Invoke-Command` showed how to send just one command, or even a few commands separated by semicolons. For example, to run a command that's located in a module, you first need to load the module:

```
PS C:\> Invoke-Command -ScriptBlock { Import-Module ActiveDirectory;
➥ Get-ADUser -filter * } -ComputerName WINDC1
```

PowerShell v3 and v4 autoloads modules by default, though you won't see them using `Get-Module -ListAvailable` until you've used them. Forcing the module to load is required for PowerShell v2 and does no harm in v3 or later. In a mixed environment, it's essential. The module has to be available on the remote machine. `Invoke-Command` can also send an entire script file, if you prefer. The file path and name are provided to the `-FilePath` parameter, which you'd use in place of `-ScriptBlock`. PowerShell will read the contents of the file from the local machine and transmit them over the network—the remote machines don't need direct access to the file itself.

10.4 PSSessions

So far, your use of Remoting has been ad hoc. You've allowed PowerShell to create the connection, it's run your commands, and then it closes the connection. Without realizing it, you've been creating a temporary *PowerShell session*, or PSS*ession*. A PSSession represents the connection between your computer and a remote one. Some overhead is involved in setting up a connection and then closing it down, and if you plan to connect to the same computer several times within a period of time, you may want to create a persistent connection to avoid that overhead.

Persistent connections have another advantage: They represent a running copy of PowerShell on a remote machine. Using the ad hoc Remoting that we've shown you so far, every single command you send runs in a new, fresh copy of the shell. With a persistent connection, you could continue to send commands to the same copy of PowerShell, and the results of those commands—such as importing modules—would remain in effect until you closed the connection.

10.4.1 Creating a persistent session

The `New-PSSession` command sets up one or more new sessions. You might want to assign these session objects to a variable so that you can easily refer to them in the future:

```
PS C:\> $win8 = New-PSsession -ComputerName win8
PS C:\> $domaincontrollers = New-PSsession -ComputerName win8,windc1
```

Here, you've created a variable, `$win8`, that contains a single session object, and a variable, `$domaincontrollers`, that contains two session objects.

> **NOTE** `New-PSSession` offers the same options for using alternate credentials, using SSL, and using port numbers as `Enter-PSSession` and `Invoke-Command`.

10.4.2 Using a session

Both `Invoke-Command` and `Enter-PSSession` can use an already-open session object. Provide the object (or objects) to the commands' `-Session` parameter, instead of using the `-ComputerName` parameter. For example, to initiate a 1-to-1 connection to a computer, use this:

```
PS C:\> Enter-PSSession -Session $win8
[win8]: PS C:\Users\Administrator\Documents>
```

Be careful to pass only a single session to `Enter-PSSession`; if you give it multiple objects, the command can't function properly. `Invoke-Command`, though, can accept multiple sessions:

```
PS C:\> Invoke-Command -Session $domaincontrollers -ScriptBlock {
➥ get-eventlog -LogName security -Newest 50 }
```

As we mentioned, it's a lot easier to work with sessions if you keep them in a variable. That isn't mandatory, though, because you can use `Get-PSSession` to retrieve sessions. For example, if you have an open session to a computer named `WINDC1`, you can retrieve the session and connect to it like this:

```
PS C:\> Enter-PSSession -Session (Get-PSSession -computername WINDC1)
```

The parenthetical `Get-PSSession` runs first, returning its session object to the `-Session` parameter of `Enter-PSSession`. If you have multiple sessions open to the same computer, the command will fail.

10.4.3 Managing sessions

Session objects will remain open and available for quite some time by default; you can configure a shorter idle timeout if you want. You can display a list of all sessions, and their status, by running `Get-PSSession` with no parameters:

```
PS C:\> Get-PSSession

 Id Name            ComputerName    State         ConfigurationName     Ava
                                                                        ila
                                                                        bil
                                                                        ity
 -- ----            ------------    -----         -----------------     ---
  6 Session6        win8            Opened        Microsoft.PowerShell ble
  7 Session7        win8            Opened        Microsoft.PowerShell ble
```

Note that the output includes both the state (Opened, in this case) and availability (Available, although our output here is a bit truncated). You can also see the name of the endpoint that the session is connected to—Microsoft.PowerShell in both instances in this example. One reason you might maintain multiple connections to a single remote machine is to connect to different endpoints—perhaps, for example, you might want a connection to both a 64-bit and a 32-bit PowerShell session.

When you've finished with a session, you can close it to free up resources. For example, to close all open sessions, use this:

```
PS C:\> Get-PSSession | Remove-PSSession
```

`Get-PSSession` is quite flexible. It provides parameters that let you retrieve just a subset of available sessions without having to get them all and then filter them through `Where-Object`:

- `-ComputerName` retrieves all sessions for the specified computer name.
- `-ApplicationName` retrieves all sessions for the specified application.
- `-ConfigurationName` retrieves all sessions connected to the specified endpoint, such as Microsoft.PowerShell.

10.4.4 Disconnecting and reconnecting sessions

PowerShell v3 introduced the ability to disconnect a session and then later reconnect it. A disconnected session is still running on the remote machine, meaning you can potentially start a long-running process, disconnect, and then reconnect later to check your results. You can even receive the results from a disconnected session without having to explicitly reconnect.

Note that the disconnection isn't necessarily automatic. If you just close your shell window, or if your computer crashes, PowerShell won't automatically put the remote session into a disconnected state. Instead, it'll shut the session down. Disconnecting is something you have to explicitly do, although PowerShell *can* automatically put a session into a disconnected state after a long timeout period or a network outage. The neat thing is that you can start a session from one computer, disconnect it, and then reconnect to that session from another computer. For example, to start a session and then disconnect it, use this:

```
PS C:\> New-PSSession -ComputerName win8

 Id Name            ComputerName    State         ConfigurationName    Ava
                                                                       ila
                                                                       bil
                                                                       ity
 -- ----            ------------    -----         -----------------    ---
 16 Session16       win8            Opened        Microsoft.PowerShell ble
```

Availability value when the session is open is - `Available`

```
PS C:\> Get-PSSession -ComputerName win8 | Disconnect-PSSession

 Id Name            ComputerName    State         ConfigurationName    Ava
                                                                       ila
                                                                       bil
                                                                       ity
 -- ----            ------------    -----         -----------------    ---
 16 Session16       win8            Disconnected  Microsoft.PowerShell one
```

Availability value when the session is disconnected is – `None`.

Now you can shut down your shell window, move to an entirely different computer, and reconnect the session from there. To do so, run `Connect-PSSession` and specify the computer name on which the session is running (you can also specify an application name and configuration name using the appropriate parameters):

```
PS C:\> Connect-PSSession -ComputerName win8

 Id Name            ComputerName    State         ConfigurationName    Ava
                                                                       ila
                                                                       bil
                                                                       ity
 -- ----            ------------    -----         -----------------    ---
 16 Session16       win8            Opened        Microsoft.PowerShell ble
```

Here's an important thing to note: You can reconnect to someone else's session. For example, it's possible for Bob to "grab" a session that was originally opened by, and disconnected by, Jane. You need to be an administrator to seize someone else's session as long as you have the credentials.

`Invoke-Command` can be used in its ad hoc mode—when you specify a computer name rather than a session—and told to create a disconnected session. The command will start up a session, send the command you specify, and then leave the session disconnected and still running that command. You can reconnect later or receive the results from the session. Here's an example:

```
PS C:\> Invoke-Command -ComputerName win8 -ScriptBlock { get-eventlog
➥  -LogName security -Newest 1000 } -Disconnected

 Id Name            ComputerName    State         ConfigurationName    Ava
                                                                       ila
                                                                       bil
                                                                       ity
 -- ----            ------------    -----         -----------------    ---
 13 Session12       win8            Disconnected  http://schemas.mi... one

PS C:\> Receive-PSSession -Session (Get-PSSession -ComputerName win8)

   Index Time          EntryType   Source                 InstanceID Me PS
                                                                 ss Co
                                                                 ag mp
                                                                 e  ut
                                                                    er
                                                                    Na
                                                                    me
   ----- ----          ---------   ------                 ---------- -- --
     299 Mar 14 16:24  SuccessA... Microsoft-Windows...         4616 Th wi
     298 Mar 14 15:23  SuccessA... Microsoft-Windows...         4616 Th wi
     297 Mar 14 14:22  SuccessA... Microsoft-Windows...         4616 Th wi
     296 Mar 14 13:21  SuccessA... Microsoft-Windows...         4616 Th wi
```

Here, you can see that we invoked the command and asked it to create a disconnected session. The `-Disconnected` parameter we used is an alias for `-InDisconnectedSession`. Normally, when you specify a computer name the session will start, run the command, and then send you the results and close. In this case, you anticipate the command taking a few moments to complete, so you leave the session running and disconnected. `Receive-PSSession` is used to retrieve the results. The session is still running and disconnected, but if you want to run further commands in it, you can easily reconnect it to do so:

```
PS C:\> Get-PSSession -ComputerName win8 | Connect-PSSession

 Id Name            ComputerName    State         ConfigurationName     Ava
                                                                        ila
                                                                        bil
                                                                        ity
 -- ----            ------------    -----         -----------------     ---
 13 Session12       win8            Opened        http://schemas.mi... ble
PS C:\> invoke-command -ScriptBlock { get-service } -Session (Get-PSSession
➥  -ComputerName win8)
Status   Name               DisplayName                           PSCompu
                                                                  terName
------   ----               -----------                           -------
Stopped  AeLookupSvc        Application Experience                win8
Stopped  ALG                Application Layer Gateway Service     win8
Stopped  AllUserInstallA... Windows All-User Install Agent        win8
Stopped  AppIDSvc           Application Identity                  win8
Stopped  Appinfo            Application Information               win8
Stopped  AppMgmt            Application Management                win8
```

10.5 Advanced session techniques

There's a lot more you can do with sessions. Keep in mind that Remoting always involves a session—even if it's one that's created, used, and closed automatically. Therefore, most of the options we'll discuss in the next two sections apply both to the `-PSSession` cmdlets as well as `Invoke-Command`, because all of them involve the use of Remoting sessions.

10.5.1 Session parameters

Several common parameters are used by the Remoting cmdlets:

- `-Authentication` specifies an authentication mechanism. Kerberos is the default; you can also specify Basic, CredSSP, Digest, Negotiate, and Negotiate-WithImplicitCredential. CredSSP is a common alternative that offers a solution to the "second hop" problem, which we'll discuss later. Note that the protocol you specify must be enabled in WinRM before it can be used, and only Kerberos is enabled by default. You can see the authentication protocols configured on the client by using this:

  ```
  dir wsman:\localhost\client\auth
  ```

  The remote authentication configuration can be viewed like this:

  ```
  Connect-WSMan -ComputerName server02
  dir wsman:server02\service\auth
  ```

  `-SessionOption` specifies a Session Options object, which wraps up a number of advanced configuration settings. We'll discuss those next.

- `-AllowRedirection` allows your Remoting session to be redirected from the computer you originally specified and handled by another remote machine instead. It's unusual to use this on an internal network, but it's common when

you're connecting to a cloud infrastructure. Microsoft Office 365 is an excellent example: You'll often connect PowerShell to a generic computer name and then be redirected to the specific server that handles your organization's data.

- `-ApplicationName` connects a session to the specified application name, such as http://localhost:5985/WSMAN. The application name is always a URI starting with http:// or https://.
- `-ConfigurationName` connects a session to the specified configuration or endpoint. This can either be a name, like Microsoft.PowerShell, or a full URI, such as http://schemas.microsoft.com/powershell.
- `-ConnectionURI` specifies the connection endpoint—this is more or less an alternate way of specifying a computer name, port number, and application name in one easy step. These look something like http://SERVER2:5985/PowerShell, including the transport (http or https), the computer name, the port, and the application name.

When creating a new session with either `Invoke-Command` or `New-PSSession`, you can specify a friendly name for the session. Just use `-SessionName` with `Invoke-Command`, or use `-Name` with `New-PSSession`. Once you've done so, it's a bit easier to retrieve the session again: Just use `Get-PSSession` and the `-Name` parameter to specify the friendly name of the desired session.

10.5.2 Session options

On most of the Remoting-related commands you'll notice a `-SessionOption` parameter, which accepts a Session Options object. This object consolidates a number of advanced parameters that can be used to set up a new session. Typically, you'll create the options object using `New-PSSessionOption`, export the session to an XML file (or store it in a variable), and then reimport it (or specify the variable) to utilize the options. `New-PSSessionOption` supports a number of parameters, and you can read all about them in its help file.

For example, suppose you occasionally want to open a new session with no compression or encryption. Here's how you could create a reusable options object and then use it to open a new session:

```
PS C:\> New-PSSessionOption -NoCompression
➥  -NoEncryption | Export-Clixml NoCompNoEncOption.xml
PS C:\> New-PSSession -ComputerName win8
➥  -SessionOption (Import-Clixml .\NoCompNoEncOption.xml)
```

NOTE This particular set of session options won't work by default, because the default client profile doesn't permit unencrypted traffic. We modified our test computer to permit unencrypted traffic to help ease troubleshooting and experimentation in our lab.

`New-PSSessionOption` has a whole slew of parameters; none of them are mandatory. Specify the ones you want, and omit the ones you don't care about, when creating a new session options object.

10.6 Creating a custom endpoint

The `New-PSSessionConfigurationFile` cmdlet makes it easy to set up new endpoints. You're not technically creating anything related to a PSSession, despite what the cmdlet name implies; you're creating a new Remoting configuration, also known as an *endpoint*, that will run Windows PowerShell. The command uses a number of parameters, most of which are optional. We'll let you read the command's help for full details and stick with the most important parameters. The first, `-Path`, is mandatory and specifies the path and filename of the session configuration file that you want to create. You must give the file the ".pssc" filename extension.

Everything else is optional. Some of the parameters, such as `-AliasDefinitions`, accept a hash table (we cover those in chapter 16). This parameter, for example, defines a set of aliases that'll be available to anyone who connects to this new endpoint. You'd specify something like `-AliasDefinitions @{Name='hlp';definition='Get-Help'; options='ReadOnly'}` to define an alias named `hlp` that runs the `Get-Help` cmdlet and that isn't modifiable by anyone using the endpoint (ReadOnly).

Here's an example:

```
PS C:\> New-PSSessionConfigurationFile -Path Restricted.pssc
➥  -LanguageMode Restricted -VisibleProviders FileSystem
➥  -ExecutionPolicy Restricted -PowerShellVersion 3.0
```

This code creates a new configuration file that specifies:

- The endpoint will be in Restricted Language mode. Users will be able to run cmdlets and functions, but they may not create script blocks or variables and may not use other scripting language features. Only basic comparison operators will be available (all of this is documented in the command's help for the `-LanguageMode` parameter).
- The endpoint will be PowerShell 3.0. If you omit this parameter the newest available version of Windows PowerShell is used. Valid values are 2.0 and 3.0 even in PowerShell v4 and later. We recommend using the newest available version.
- Only the FileSystem PSProvider will be available; other forms of storage won't be connected as drives.
- Script execution won't be permitted, meaning that only cmdlets will be available to run.

Next, you ask the shell to use that configuration file to create the new endpoint, registering it with WinRM:

```
PS C:\> Register-PSSessionConfiguration -Path .\Restricted.pssc -Force
➥  -Name MyEndpoint

   WSManConfig: Microsoft.WSMan.Management\WSMan::localhost\Plugin

Type            Keys                                Name
----            ----                                ----
Container       {Name=MyEndpoint}                   MyEndpoint
```

You define the name MyEndpoint for this new endpoint, so to create a session that connects to it, you go to another computer and use `New-PSSession`:

```
PS C:\> $sess = New-PSSession -ComputerName win8
➥  -ConfigurationName MyEndpoInt
```

Now you can use that session object with `Enter-PSSession` or `Invoke-Command`, as you learned earlier in this chapter.

There are other commands used for unregistering a configuration, disabling and enabling them (while leaving them registered), and so forth:

```
PS C:\> Get-Command -Noun pssessionconfiguration*

Capability      Name
----------      ----
Cmdlet          Disable-PSSessionConfiguration
Cmdlet          Enable-PSSessionConfiguration
Cmdlet          Get-PSSessionConfiguration
Cmdlet          New-PSSessionConfigurationFile
Cmdlet          Register-PSSessionConfiguration
Cmdlet          Set-PSSessionConfiguration
Cmdlet          Test-PSSessionConfigurationFile
Cmdlet          Unregister-PSSessionConfiguration
```

When you create a custom session configuration file, as you've seen, you can set its language mode. The language mode determines what elements of the PowerShell scripting language are available in the endpoint, and the language mode can be a bit of a loophole. With the `Full` language mode, you get the entire scripting language, including *script blocks*. A script block is any executable hunk of PowerShell code contained within curly brackets {}. They're the loophole. Any time you allow the use of script blocks, they can run any legal command, even if your endpoint used `-VisibleCmdlets` or `-VisibleFunctions` or another parameter to limit the commands in the endpoint.

In other words, if you register an endpoint that uses `-VisibleCmdlets` to expose `Get-ChildItem` but you create the endpoint's session configuration file to have the full language mode, *then any script blocks inside the endpoint can use any command*. Someone could run:

```
PS C:\> & { Import-Module ActiveDirectory; Get-ADUser -filter * |
➥ Remove-ADObject }
```

Eek! This can be especially dangerous if you configured the endpoint to use a RunAs credential to run commands under elevated privileges. It's also somewhat easy to let this happen by mistake, because you set the language mode when you create the new session configuration file (`New-PSSessionConfigurationFile`), not when you *register* the session (`Register-PSSessionConfiguration`). So if you're using a session configuration file created by someone else, pop it open and confirm its language mode before you use it!

You can avoid this problem by setting the language mode to `NoLanguage`, which shuts off script blocks and the rest of the scripting language. Or, go for `RestrictedLanguage`,

which blocks script blocks while still allowing some basic operators if you want users of the endpoint to be able to do basic filtering and comparisons.

Understand that this isn't a bug—the behavior we're describing here is by design. But it can be a problem if you don't know about it and understand what it's doing.

NOTE Much thanks to fellow MVP Aleksandar Nikolic for helping us understand the logic of this loophole!

10.6.1 Custom endpoints for delegated administration

One of the coolest things you can do with a custom endpoint is called *delegated administration*. You set up the endpoint so that it runs all commands under a predefined user account's authority, rather than using the permissions of the user who connected to the endpoint. This is especially useful for PowerShell Web Access.

To start, you create a custom endpoint, just as we showed you earlier. When creating the new session configuration file, you restrict the endpoint. So, when you're running `New-PSSessionConfigurationFile`, you'll generally do something like this:

- Use `-ExecutionPolicy` to define a `Restricted` execution policy if you don't want people running scripts in the endpoint.
- Use `-ModulesToImport` to specify one or more modules to load into the session.
- Use `-FunctionDefinitions` to define custom functions that will appear within the session.
- Potentially use `-LanguageMode` to turn off PowerShell's scripting language; this is useful if you want people to run only a limited set of commands.
- Use `-SessionType` to set the session type to `RestrictedRemoteServer`. This turns off most of the core PowerShell commands, including the ability to import any modules or extensions that aren't part of the session configuration file.
- Use `-VisibleCmdlets` to specify which commands you want visible within the session. You have to make sure their module is imported, but this lets you expose less than 100 percent of the commands in a module. Use `-VisibleFunctions` to do the same thing for imported functions, and use `-VisibleProviders` to make specific PSProviders available.

Register the new session configuration using `Register-PSSessionConfiguration`. When you do so, use the `-RunAsCredential` parameter to specify the username that all commands within the session will run as. You'll be prompted for the password. You might also want to consider these parameters:

- `-AccessMode` lets you specify that the endpoint can only be used by local users ("Local") or by local and remote ("Remote").
- `-SecurityDescriptorSddl` lets you specify, in the Security Descriptor Definition Language (SDDL), who can use the endpoint. Users must have, at a minimum, "Execute(Invoke)" in order to be able to use the session. We find SDDL to

be complex, so you could specify the `-ShowSecurityDescriptorUI` parameter, which lets you set the endpoint permissions in a GUI dialog box. See, GUIs are still useful for some things!

In the end, you've created an endpoint that (a) only certain people can connect to, and that (b) will run commands under an entirely different set of credentials. Delegated administration! The people using the endpoint don't need permission to run the commands you've allowed within it!

10.7 Connecting to non-default endpoints

To connect to an endpoint other than the default PowerShell endpoint, you need to know the endpoint name, also called its configuration name. You can run `Get-PSSessionConfiguration` to see all of the endpoints configured on the local machine:

```
PS C:\> Get-PSSessionConfiguration

Name          : microsoft.powershell
PSVersion     : 4.0
StartupScript :
RunAsUser     :
Permission    : BUILTIN\Administrators AccessAllowed,
                BUILTIN\Remote Management Users AccessAllowed

Name          : microsoft.powershell.workflow
PSVersion     : 4.0
StartupScript :
RunAsUser     :
Permission    : BUILTIN\Administrators AccessAllowed,
                BUILTIN\Remote Management Users AccessAllowed

Name          : microsoft.powershell32
PSVersion     : 4.0
StartupScript :
RunAsUser     :
Permission    : BUILTIN\Administrators AccessAllowed,
                BUILTIN\Remote Management Users AccessAllowed
```

This output shows you the configuration name, which you provide to the `New-PSSession -ConfigurationName` parameter when creating a new session:

```
PS C:\> New-PSSession -ComputerName win8
➥  -ConfigurationName 'microsoft.powershell32'
 Id Name            ComputerName    State         ConfigurationName    Ava
                                                                       ila
                                                                       bil
                                                                       ity
 -- ----            ------------    -----         -----------------    ---
 19 Session19       win8            Opened        microsoft.powersh... ble
```

You'll also find a `-ConfigurationName` parameter on `Invoke-Command` and `Enter-PSSession`, which enables those cmdlets to connect to an alternate endpoint without creating a persistent session object first.

`Get-PSSessionConfiguration` only works on the local machine. If you need to discover the endpoints on a remote machine, you can do one of two things. Your first option is to create a session to the remote machine and use `Get-PSSessionConfiguration`:

```
PS C:\> Enter-PSSession -ComputerName dc02
[dc02]: PS C:\Users\Richard\Documents> Get-PSSessionConfiguration
```

Alternatively, you could use `Connect-WSMan` like this:

```
PS C:\> Connect-WSMan -ComputerName w12standard
PS C:\> dir wsman:\w12standard\plugin
```

Both methods work and give the required results as long as Remoting is enabled on the remote system.

10.8 Enabling the "second hop"

We've mentioned this "second hop" thing a number of times. It's essentially a built-in, default limitation on how far your credentials can be delegated. Here's the scenario:

- You're using a computer named CLIENT. You open PowerShell, making sure that the shell is run as Administrator. You can run whatever commands you like.
- You use `Enter-PSSession` to remote to a machine named SERVER1. Your credentials are delegated via Kerberos, and you can run whatever commands you like.
- While still remoted into SERVER1, you use `Invoke-Command` to send a command, via Remoting, to SERVER2. Your credentials can't delegate across this "second hop," and so the command fails.

There are two workarounds to solve this problem. The first is easy: Specify a `-Credential` parameter any time you're launching a new Remoting connection across the second and subsequent hops. In our example scenario, while running `Invoke-Command` on SERVER1 to connect to SERVER2, provide an explicit credential. That way, your credential doesn't need to be delegated, and you avoid the problem.

> **NOTE** If you're a domain administrator and the local machine (CLIENT in this example) is a domain controller, some elements of the delegation to enable "second hop" processing are available by default. We don't recommend using domain controllers as administration workstations!

The second technique requires that you enable, and then use, the CredSSP authentication protocol on all machines involved in the chain of Remoting, starting with your computer (CLIENT in our example scenario) and including every machine that you'll remote to. Enabling CredSSP is most easily done through Group Policy, where you can configure it for entire groups of computers at once. You can, though, enable it on a per-machine basis using the WSMan: drive in PowerShell:

```
PS WSMan:\localhost\Service\Auth> dir

   WSManConfig: Microsoft.WSMan.Management\WSMan::localhost\Service\Auth

Type            Name                              SourceOfValue   Value
----            ----                              -------------   -----
System.String   Basic                                             false
System.String   Kerberos                                          true
System.String   Negotiate                                         true
System.String   Certificate                                       false
System.String   CredSSP                                           false
System.String   CbtHardeningLevel                                 Relaxed

PS WSMan:\localhost\Service\Auth> set-item ./credssp $true
PS WSMan:\localhost\Service\Auth> dir

   WSManConfig: Microsoft.WSMan.Management\WSMan::localhost\Service\Auth

Type            Name                              SourceOfValue   Value
----            ----                              -------------   -----
System.String   Basic                                             false
System.String   Kerberos                                          true
System.String   Negotiate                                         true
System.String   Certificate                                       false
System.String   CredSSP                                           true
System.String   CbtHardeningLevel                                 Relaxed
```

Here, we've shown the protocol before and after enabling it in WSMan:\localhost\Service\Auth. Once it's enabled, specify `-Authentication CredSSP` when using `Invoke-Command`, `Enter-PSSession`, or `New-PSSession` to use the protocol. An alternative, and possibly simpler, technique is to use the `Enable-WSManCredSSP` cmdlet on the relevant machines.

On the client machine, run:

```
Enable-WSManCredSSP -Role Client -DelegateComputer SERVER1
```

We recommend that you only enable CredSSP when required rather than as a permanent configuration.

On the remote machine, run:

```
Enable-WSManCredSSP -Role Server
```

10.9 Setting up WinRM listeners

`Enable-PSRemoting` creates a single WinRM listener that listens on all enabled IP addresses on the system. You can discover the existing listeners by using this:

```
PS C:\> Get-WSManInstance winrm/config/Listener -Enumerate

cfg                 : http://schemas.microsoft.com/wbem/wsman/1/config/
                      listener
xsi                 : http://www.w3.org/2001/XMLSchema-instance
lang                : en-US
Address             : *
Transport           : HTTP
Port                : 5985
Hostname            :
Enabled             : true
```

```
URLPrefix             : wsman
CertificateThumbprint :
ListeningOn           : {10.10.54.165, 127.0.0.1, 192.168.2.165, ::1...}
```

And the IP addresses that are being listened on are discovered like this:

```
Get-WSManInstance winrm/config/Listener -Enumerate |
select -ExpandProperty ListeningOn
```

Alternatively, you can use the WSMAN provider:

```
PS C:\> dir wsman:\localhost\listener

   WSManConfig: Microsoft.WSMan.Management\WSMan::localhost\Listener

Type            Keys                                Name
----            ----                                ----
Container       {Address=*, Transport=HTTP}         Listener_809701527
```

Keep in mind that a single WinRM listener can service any number of endpoints and applications, as shown in figure 10.1; you only need to set up a new listener if the default one (which uses HTTP on port 5985) isn't what you want to use. It's easier to change the default listener to use different settings if you don't want to use its default settings at all. But if you want both that listener and an alternate one, then you need to create that alternate one.

Why might you want to create a new listener? The most probable answers are that you want to restrict the IP addresses, or ports, that are used for listening or you want to create a listener for secured traffic using HTTPS rather than HTTP. A combination of these conditions would allow only connections over HTTPS to a specific IP address and port. That approach is useful in an environment requiring secure transport and access—for example, to a server in the DMZ where you need to be able to connect over the management network but not from the internet-facing address.

10.9.1 Creating an HTTP listener

You can create a new listener by using the `New-WSManInstance` cmdlet:

```
PS C:\> New-WSManInstance winrm/config/Listener
➥  -SelectorSet @{Transport='HTTP'; Address="IP:10.10.54.165"}
➥  -ValueSet @{Port=8888}
```

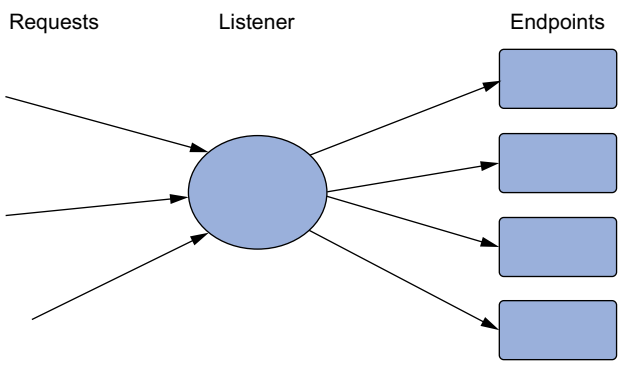


Figure 10.1 A single listener servicing multiple endpoints

The address, port, and transport protocol are specified, but notice that they're in two separate groups. That's because `New-WSManInstance` uses `-SelectorSet` to identify the individual instance (see the Keys column in the following code) and `-ValueSet` to define property values. You can see the new listener like this:

```
PS C:\> dir wsman:\localhost\listener | Format-Table -AutoSize

   WSManConfig: Microsoft.WSMan.Management\WSMan::localhost\Listener

Type      Keys                                        Name
----      ----                                        ----
Container {Address=*, Transport=HTTP}                 Listener_809701527
Container {Address=IP:10.10.54.165, Transport=HTTP}   Listener_886604375
```

10.9.2 Adding an HTTPS listener

Adding a listener for HTTPS is similar, but you need to go through a few steps first:

1. Create a certificate request. You can't do that in PowerShell and need to either ask your Certificate Services administrators for help or use the tools provided by your certificate provider.
2. Request the certificate using the request you've just created.
3. Download the certificate.
4. Install the certificate into the computer certificate store.
5. Find the new certificate in the PowerShell cert: drive and get its thumbprint.

You can now create the listener:

```
New-WSManInstance winrm/config/Listener
➥  -SelectorSet @{Transport='HTTPS'; Address="IP:10.10.54.165"}
➥  -ValueSet @{Hostname="<servername>";CertificateThumbprint="XXXXXXXX"}
```

where `Hostname` matches the server name in your SSL certificate.

You can remove a listener using `Remove-WSManInstance`:

```
PS C:\> Get-WSManInstance winrm/config/Listener
➥  -SelectorSet @{Transport='HTTP'; Address="IP:10.10.54.165"} |
Remove-WSManInstance
```

Or use

```
Remove-WSManInstance winrm/config/Listener
➥  -SelectorSet @{Transport='HTTP'; Address="IP:10.10.54.165"}
```

You remove the default listener like this:

```
Remove-WSManInstance winrm/config/Listener
➥  -SelectorSet @{Transport="HTTP"; Address="*"}
```

We recommend restarting the WinRM service after you modify the listeners.

There are two modifications you can make to a connection, whether using `Invoke-Command`, `Enter-PSSession`, or some other Remoting command that relates to HTTPS listeners. These are created as part of a session option object.

- `-SkipCACheck` causes WinRM to not worry about whether or not the SSL certificate was issued by a trusted CA. But untrusted CAs may in fact be untrustworthy! A poor CA might issue a certificate to a bogus computer, leading you to believe you're connecting to the right machine when in fact you're connecting to an imposter. Using this parameter is risky, so do so with caution.
- `-SkipCNCheck` causes WinRM to not worry about whether or not the SSL certificate on the remote machine was actually issued for that machine. Again, this is a great way to find yourself connected to an imposter. Half the point of SSL is mutual authentication, and this parameter disables that half.

10.10 Other configuration scenarios

So far in this chapter, we've tried to focus on the easy and common Remoting configuration scenarios, but we know there are other scenarios you'll have to confront. In the next few sections, we'll cover some of these "outside the lines" cases. There are certainly others, and you'll find most of those documented in PowerShell's about_remote_troubleshooting help file, which we heartily recommend that you become familiar with. That file also explains how to configure many of the Remoting configuration settings, set up firewall exceptions, and perform other tasks via Group Policy—which is a lot easier than configuring individual machines one at a time.

10.10.1 Cross-domain Remoting

Remoting doesn't work across Active Directory domains by default. If your computer is in DOMAINA, and you need to remote into a machine that belongs to DOMAINB, you'll have to do a bit of work first. You'll still need to ensure that your user account has permissions to do whatever it is you're attempting in DOMAINB—the configuration setting we're showing you only enables the Remoting connectivity. This is a Registry setting, so be careful when making this change:

```
PS C:\> New-ItemProperty -Name LocalAccountTokenFilterPolicy -Path
➥  HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\System
➥  -PropertyType DWord -Value 1
```

This code will enable all members of a machine's Administrators group, regardless of the domain they're in, to use Remoting on the machine. So, in our example, you'd make this change on the machine in DOMAINB—the destination machine of the Remoting connection.

10.10.2 Quotas

The great thing about Remoting is that it exists and solves a number of administration problems. The bad thing (and there's always one of those) is that too much Remoting can damage your system health. Imagine the scenario where you've implemented a server to support a new business-critical application. The application is being rolled out across the enterprise and the number of users is growing rapidly. At a certain loading you realize that the application is breaking down and consuming more resources than it should. You need to restrict the amount of resources devoted to PowerShell Remoting. How? You set quotas.

If you look in the WSMAN provider, you'll see a number of possible quota sessions:

```
PS C:\> dir wsman:\localhost | select Name, Value

Name                                                            Value
----                                                            -----
MaxEnvelopeSizekb                                               500
MaxTimeoutms                                                    60000
MaxBatchItems                                                   32000
MaxProviderRequests                                             4294967295

PS C:\> dir wsman:\localhost\service | select Name, value

Name                                                            Value
----                                                            -----
MaxConcurrentOperations                                         4294967295
MaxConcurrentOperationsPerUser                                  1500
EnumerationTimeoutms                                            240000
MaxConnections                                                  300
MaxPacketRetrievalTimeSeconds                                   120
```

We haven't come across a situation where the defaults needed to be changed, but just in case you should ever need to make a change, this is how you do it:

```
Set-Item wsman:\localhost\MaxEnvelopeSizeKB -value 200
```

This code sets a global value for the size of the envelope (message) to 200 KB. Quotas can be set on individual session configurations:

```
Set-PSSessionConfiguration -name microsoft.powershell
➥  -MaximumReceivedObjectSizeMB 11 -Force
```

This increases the maximum object size for the microsoft.powershell endpoint. Other quota values can be found in a number of areas of the listener and endpoint configurations:

```
dir wsman:\localhost\plugin\microsoft.powershell\quotas
dir wsman:\localhost\plugin\microsoft.powershell\InitializationParameters
```

10.10.3 Configuring on a remote machine

You may run into instances where you need to modify the WinRM configuration on a remote computer. WinRM needs to be up and running on that system, and you can use the `Connect-WSMan` cmdlet to create the connection:

```
PS WSMan:\> Connect-WSMan -ComputerName win8
PS WSMan:\> dir

   WSManConfig:

ComputerName                                    Type
------------                                    ----
localhost                                       Container
win8                                            Container
```

As you can see here, the new computer shows up alongside localhost in your WSMan: drive, enabling you to access the machine's WinRM configuration. You might also want to use the `Test-WSMan` cmdlet to verify everything:

```
PS C:\> Test-WSMan -comp quark -Authentication default

wsmid           : http://schemas.dmtf.org/wbem/wsman/identity/1/
                  wsmanidentity.xsd
ProtocolVersion : http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd
ProductVendor   : Microsoft Corporation
ProductVersion  : OS: 6.2.8250 SP: 0.0 Stack: 3.0
```

In addition to validating that Remoting is working, you can see the WinRM stack version (the `OS` and `SP` values will only be visible if the `-Authentication default` parameter is used). In this example, Quark is running PowerShell 3.0 and therefore WSMAN 3.0 is shown in the `Stack` property.

NOTE WSMAN version 3.0 is used in PowerShell v3 and v4.

For the most part you shouldn't run into any issues Remoting from a PowerShell 4.0, or 3.0, machine to one running PowerShell 2.0, but this is a handy tool for double-checking version information. You'll need this when we discuss CIM sessions in chapter 39.

10.10.4 Key WinRM configuration settings

All of these settings are located in your WSMan: drive; we'll cover the ones of most common interest but you can explore the drive to discover others. Many of these can also be configured via Group Policy—look for the "Windows Remote Management" section of the Group Policy object, under the Computer Configuration container.

- \Shell\IdleTimeout—The number of milliseconds a Remoting session can sit idle before being disconnected
- \Shell\MaxConcurrentUsers—The maximum number of Remoting sessions any number of users can have to a machine
- \Shell\MaxShellRunTime—The maximum time any Remoting session can be open, in milliseconds
- \Shell\MaxProcessesPerShell—The maximum number of processes any Remoting session can run
- \Shell\MaxMemoryPerShellMB—The maximum amount of memory any Remoting session can utilize
- \Shell\MaxShellsPerUser—The maximum number of Remoting sessions any one user can open to the machine

To change one of these settings manually, use the `Set-Item` cmdlet:

```
PS C:\> Set-Item WSMAN:\Localhost\Shell\IdleTimeout -Value 3600000
```

WARNING The updated configuration might affect the operation of the plug-ins having a per-plug-in quota value greater than 3600000. Verify the configuration of all the registered plug-ins and change the per-plug-in quota values for the affected plug-ins.

Some WSMAN settings can be configured at a global and individual plug-in level (a plug-in is another way of looking at a session configuration). This is especially true when the plug-in needs to use the capability of the shell. If you run this code

```
Get-Item -Path wsman:\localhost\shell\IdleTimeout
Get-ChildItem wsman:\localhost\plugin |
foreach {
 Get-Item "wsman:\localhost\plugin\$($_.Name)\quotas\IdleTimeoutms"
}
```

you'll get back something like this:

```
   WSManConfig: Microsoft.WSMan.Management\WSMan::localhost\Shell

Type            Name                              SourceOfValue   Value
----            ----                              -------------   -----
System.String   IdleTimeout                                     7200000

   WSManConfig:
Microsoft.WSMan.Management\WSMan::localhost\Plugin\microsoft.powershell
\Quotas

Type            Name                              SourceOfValue   Value
----            ----                              -------------   -----
System.String   IdleTimeoutms                                   7200000

   WSManConfig: Microsoft.WSMan.Management\WSMan::localhost\Plugin\microsoft
     .powershell.
workflow\Quotas

Type            Name                              SourceOfValue   Value

----            ----                              -------------   -----
System.String   IdleTimeoutms                                   7200000
```

As the error message on the `Set-Item` call explains, if you change the timeout setting at the shell level it will conflict with the setting at the plug-in level. The plug-in needs to be modified to match the shell. As with quotas, the default settings work very well and we don't know any reason for changing them in normal operating conditions.

10.10.5 Adding a machine to your Trusted Hosts list

Remoting doesn't like to connect to machines that it doesn't trust. You might think you're connecting to a remote machine named SERVER1, but if an attacker could somehow spoof DNS or perform some other trickery, they could hijack your session and have you connect to the attacker's machine instead. They could then capture all manner of useful information from you. Remoting's concept of trust prevents that from happening. By default, Remoting trusts only machines that are in the same Active Directory domain as your computer, enabling it to use Kerberos authentication

to confirm the identity of the remote machine. That's why, by default, you can't remote to a machine using an IP address or hostname alias: Remoting can't use those to look up the machine's identity in Active Directory.

You can modify this behavior by manually adding machine names, IP addresses, and other identifiers to a persistent, static Trusted Hosts list that's maintained by WinRM. WinRM—and thus Remoting—will always trust machines on that list, although it doesn't actually authenticate them. You're opening yourself up to potential hijacking attempts—although it's rare for those to occur on an internal network.

You modify the list by using the WSMan: drive, as shown here:

```
PS WSMan:\localhost\Client> dir

   WSManConfig: Microsoft.WSMan.Management\WSMan::localhost\Client

Type            Name                             SourceOfValue   Value
----            ----                             -------------   -----
System.String   NetworkDelayms                                  5000
System.String   URLPrefix                                       wsman
System.String   AllowUnencrypted                                false
Container       Auth
Container       DefaultPorts
System.String   TrustedHosts

PS WSMan:\localhost\Client> Set-Item .\TrustedHosts *

WinRM Security Configuration.
This command modifies the TrustedHosts list for the WinRM client. The
computers in the TrustedHosts list might not be authenticated. The client
might send credential information to these computers. Are you sure that
you want to modify this list?
[Y] Yes  [N] No  [S] Suspend  [?] Help (default is "Y"): y
PS WSMan:\localhost\Client>
```

We've added `*` to TrustedHosts, essentially meaning we'll be able to use Remoting with any computer. We don't necessarily recommend that as a best practice, but it's useful in a lab environment where you just want stuff to work. In a production environment, we generally prefer to see a managed list of trusted hosts rather than the * wildcard. For example, `*.company.pri` would trust all hosts in the company.pri domain. Read the about_remote_troubleshooting PowerShell help file for a lot more detail and examples.

10.10.6 Using Group Policy to configure Remoting

This is a reminder that in a production environment the best way to configure Remoting is to use Group Policy. Full details on configuring Remoting via Group Policy can be found in the help file about_remote_troubleshooting.

We strongly recommend that you fully understand the settings by configuring manually in a lab before applying a Group Policy to your enterprise.

10.11 Implicit Remoting

Implicit Remoting is an incredibly cool trick and one that you'll get more and more use out of in the future. The basic idea is this: Rather than installing every possible PowerShell module on your computer, you leave the modules installed out on servers. You can then "import" the modules into your current PowerShell session, making it look like the commands in the modules all live locally. In reality, your computer will contain "shortcuts" to the commands, and the commands will execute out on the servers you got them from. The results—and even the commands' help—will be brought to your computer via Remoting.

Here's an example where you'll import the ServerManager module from a remote server:

```
PS C:\> $sess = New-PSSession -ComputerName win8
PS C:\> Invoke-Command -Session $sess -ScriptBlock { Import-Module
➥  servermanager }
PS C:\> Import-PSSession -Session $sess -Module ServerManager -Prefix RemSess

ModuleType Name                                     ExportedCommands
---------- ----                                     ----------------
Script     tmp_1hn0kr5w.keb                         {Get-WindowsFeature, Ins...
```

Here's what you did:

1. You opened a session to the remote machine, saving the session object in a variable for easy use later.
2. You invoked a command against that session, asking it to load the desired module into memory.
3. You imported that session, grabbing only the commands in the ServerManager module. To make these commands easy to distinguish, you added the prefix "RemSess" to the noun of all imported commands. The prefix is optional but is recommended especially if you're importing to a Windows 8, Windows Server 2012, or later system with the greatly increased number of cmdlets.

You can quickly check to see which commands you brought over:

```
PS> Get-Command -Noun RemSess*

CommandType Name
----------- ----
Alias       Add-RemSessWindowsFeature
Alias       Remove-RemSessWindowsFeature
Function    Disable-RemSessServerManagerStandardUserRemoting
Function    Enable-RemSessServerManagerStandardUserRemoting
Function    Get-RemSessWindowsFeature
Function    Install-RemSessWindowsFeature
Function    Uninstall-RemSessWindowsFeature
```

NOTE The module name column has been removed to enable the display to fit the page width.

You can now run these commands, just as if they were locally installed, and can even access their help (provided the server has had `Update-Help` run so that it has a copy of the help locally). The only caveat is the one that applies to all results in Remoting: The results of your commands won't have any methods attached to them, because the results will have been through the serialization/deserialization process.

These "imported" commands will exist as long as your session to the remote machine is open and available. Once it's closed, the commands will vanish. If you want to make these commands always available to you, save the remote session information to a module using the `Export-PSSession` cmdlet.

There are a few ways you might want to use this. First, take your current session and export everything to a module:

```
PS C:\> Export-PSSession -Session $q -OutputModule QuarkAll
```

The session $q is to the computer named Quark. This command will create a module called QuarkAll under $home\Documents\WindowsPowerShell\Modules:

```
PS C:\> Get-Module -ListAvailable QuarkAll

ModuleType Name                     ExportedCommands
---------- ----                     ----------------
Manifest   QuarkAll                 {}
```

Later, you can import this module as you would with implicit Remoting. Because the imported cmdlet names may conflict, add a prefix:

```
PS C:\> Import-Module QuarkAll -Prefix Q
```

The first time you try to run one of the commands, PowerShell dynamically creates the necessary session and establishes a remote connection:

```
PS C:\> Get-Qsmbshare
Creating a new session for implicit Remoting of "Get-SmbShare" command...
```

If you check sessions, you should see a new one created for this module:

```
PS C:\> Get-PSSession | select *

State                 : Opened
ComputerName          : quark
ConfigurationName     : Microsoft.PowerShell
InstanceId            : 662484ed-d350-4b76-a146-865a8d43f603
Id                    : 2
Name                  : Session for implicit Remoting module at
                        C:\Users\Jeff\Documents\WindowsPowerShell\Modules\
                        QuarkAll\QuarkAll.psm1
Availability          : Available
ApplicationPrivateData : {PSVersionTable}
Runspace              : System.Management.Automation.RemoteRunspace
```

If you remove the module, the session is also automatically removed.

You can also create a limited module by only exporting the commands you want. First, create a session:

```
PS C:\> $q=New-PSSession Quark
```

Then, create a new module exporting only the Get cmdlets:

```
PS C:\> Export-PSSession -Session $q -OutputModule QuarkGet -CommandName
➥  Get* [CA] -CommandType cmdlet
```

When you import the module, the only commands you can run remotely on Quark are the Get cmdlets:

```
PS C:\> Import-Module QuarkGet -Prefix Q
PS C:\> Get-Command -module QuarkGet

CommandType     Name                                Definition
-----------     ----                                ----------
Function        Get-QAppLockerFileInformation       ...
Function        Get-QAppLockerPolicy                ...
Function        Get-QAppxProvisionedPackage         ...
Function        Get-QAutoEnrollmentPolicy           ...
Function        Get-QBitsTransfer                   ...
...
```

One thing we should point out is that when you export a session, any commands with names that might conflict on your local computer are skipped unless you use the `-AllowClobber` parameter. In the examples with Quark, you're connecting from a computer running PowerShell 2.0 to one running PowerShell 4.0, or 3.0, and thus are able to use the cmdlets of the later versions of PowerShell just as if they were installed locally:

```
PS C:\> get-qciminstance win32_operatingsystem | Select
➥  CSName,BuildNumber,Version
Creating a new session for implicit Remoting of "Get-CimInstance" command...

CSName                    BuildNumber               Version
------                    -----------               -------
QUARK                     8250                      6.2.8250
```

Implicit Remoting is an incredibly powerful technique—and a necessity for working with remote Exchange servers—that lets you take advantage of modules, snap-ins, and tools that you may not have installed locally. If you find yourself needing these tools often, take the time to export a session to a module; then you'll be ready for anything.

10.12 Standard troubleshooting methodology

Troubleshooting can be difficult, especially with Remoting because there are so many layers in which something can go wrong. We strongly recommend that you read, learn, and inwardly digest the help file about_Remote_Troubleshooting. It contains a lot of useful information that will improve your knowledge of Remoting and enable you to troubleshoot problems. When you have to diagnose problems with Remoting, we recommend that you follow these four steps:

1. Test Remoting with its default configuration. If you've tinkered with it, undo your changes and start from scratch.
2. Start by attempting to connect from the initiating machine to the target machine by using something other than Remoting but that's still security-sensitive. For

example, use Windows Explorer to open the remote machine's C$ shared folder. If that doesn't work, you have broader security issues. Make a note of whether you need to provide alternate credentials—if you do, Remoting will need them as well.

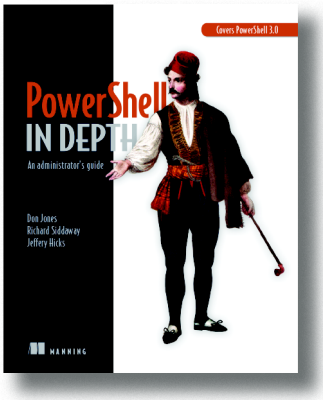
3. Install a Telnet client on the initiating machine (a simple command-line client, like the Windows native one, will do). Attempt to connect to the HTTP WinRM listener by running `telnet machine_name:5985`. You should get a blank screen, and Ctrl-C will end the session. If this doesn't work, there's a basic connectivity problem (such as a blocked port) you need to resolve.
4. Use `Test-WSMan`, using an alternate credential if necessary. Make sure that you're using the machine's real name as it appears in Active Directory or that you've taken one of the other approaches (TrustedHosts plus a credential, or SSL plus a credential). If that doesn't work, you have a problem in the WSMAN configuration.

Walking through these four steps, in this order, can help you pinpoint at least the general cause of most problems.

10.13 Summary

Remoting was the most eagerly awaited feature in PowerShell v2. It moved PowerShell's capabilities up by several levels. You can gain remote access to systems through a number of cmdlets that have a `-ComputerName` parameter or through the WSMAN-based Remoting technology.

Once you've mastered the material in this chapter, you'll be able to administer all the machines in your environment from the comfort of your own workstation.



PowerShell in Depth, Second Edition is the go-to reference for administrators working with Windows PowerShell. Every major technique, technology, and tactic is carefully explained and demonstrated, providing a hands-on guide to almost everything an admin would do in the shell. Written by three experienced authors and PowerShell MVPs, this is the PowerShell book you'll keep next to your monitor—not on your bookshelf!

A Windows admin using PowerShell every day may not have the time to search the net every time he or she hits a snag. Wouldn't it be great to have a team of seasoned PowerShell experts ready to answer even the toughest questions? That's what you get with this book.

PowerShell in Depth, Second Edition is the go-to reference for administrators working with Windows PowerShell. Every major technique, technology, and tactic is carefully explained and demonstrated, providing a hands-on guide to almost everything an admin would do in the shell. Written by PowerShell MVPs Don Jones, Jeffrey Hicks, and Richard Siddaway, each valuable technique was developed and thoroughly tested, so you'll be able to consistently write production-quality, maintainable scripts while saving hours of time and effort.

What's inside:

- Automating tasks
- Packaging and deploying scripts
- Introduction to Desired State Configuration
- PowerShell security
- Covers PowerShell version 3 and later

This book assumes you know the basics of PowerShell.

PowerShell and SQL Server

SQL Server is a common member of Windows environments, and you may find yourself having to administer this technology. You can work with SQL Server with PowerShell in a number of ways—scripting, cmdlets and the SQL Server Provider. This chapter introduces the provider and gives you a number of scripts you can use in your environment to perform common tasks.

Chapter 23 from *PowerShell Deep Dives* edited by Jeffrey Hicks, Richard Siddaway, Oisin Grehan, and Aleksander Nikolic

PowerShell and the SQL Server provider

This chapter is written for the DBA who needs an efficient way to get information from or manage SQL Servers in their environment with just a few commands by using native PowerShell methods. When you're looking at the options for managing or getting information from a SQL Server by using PowerShell your choice is driven by a few scenarios. One use case might be to find out how many databases are in the instances you maintain while using the simplest way to reference these instances. Another might be to find out whether a certain object exists in a Software as a Service (SaaS) environment with thousands of databases and multiple servers while upgrading in a phased upgrade methodology. You may want to know which database has the object so you don't attempt to upgrade that database in the second wave and find that the object exists. When faced with these or other scenarios you can quickly accomplish your goal with part PowerShell methods and part SQL Server provider.

You have a few options for managing SQL Server using PowerShell. You can use straight Shared Management Objects (SMOs) by loading the SMOs individually or by using the SQL Server provider. This chapter discusses the SQL Server provider that was released with SQL Server 2008/2008 R2. The provider for SQL 2008/R2 is implemented as a Windows PowerShell snap-in (PSSnapin) and is implemented as a module in SQL Server 2012. The provider for SQL Server 2012 has a few more cmdlets and more properties and methods on the SMOs, but the functionality is the same. I'll start by introducing you to the SQL Server provider and then I'll show you practical ways to use the provider to get at SQL Server information using PowerShell cmdlets.

23.1 Requirements

Many modules and providers come with PowerShell in Windows, but the SQL Server provider is a separate element that you install like any other Windows application. It's installed when you install SQL Server 2008, 2008 R2, or 2012 Management Tools, or you can download and install a copy of the Feature Pack for 2008 R2 at http://mng.bz/ccVK or for 2012 at http://mng.bz/m8po. You'll need to download and install the following components for 2008/R2:

- 1033\x64\PowerShellTools.msi
- 1033\x64\SharedManagementObjects.msi
- 1033\x64\SQLSysClrTypes.msi

For 2012 you install the following components:

- Microsoft Windows PowerShell Extensions for Microsoft SQL Server 2012
- Microsoft SQL Server 2012 Shared Management Objects

These components for 2008/R2 are listed for the x64 platform, and the corresponding items are available for IA64 and x86. With the components installed and with access to a SQL Server you can start exploring the capabilities of the SQL Server provider.

23.2 Introduction to the SQL Server provider

Two snap-ins are registered with PowerShell when you install the SQL Server provider components for 2008/R2: `SqlServerCmdletSnapin100` and `SqlServerProvider-Snapin100`. You can verify that the snap-ins are available by using the first command in the following code. You can add them a couple of different ways, as shown:

```
Get-PSSnapin -Registered

Add-PSSnapin SqlServerCmdletSnapin100
Add-PSSnapin SqlServerProviderSnapin100
```

Alternatively you can use Wildcards, but be sure that you only get what you want:

```
Add-PSSnapin *SQL*
```

The first snap-in contains two cmdlets that you can use to execute commands against a SQL Server. The first cmdlet, `Invoke-PolicyEvaluation`, is used in SQL Server policy-based management in SQL Server 2008 and above. The second is `Invoke-SqlCmd`, which is a query executer. These two cmdlets are useful, and in future versions of the provider there are more cmdlets available.

The second snap-in is the SQL Server provider. It's used for navigating SQL Server objects in a manner similar to navigating a directory structure, folders, and items in folders. Think of a directory like C:\WINDOWS and how you can use the `dir` command to access the items in that folder. The objects that are returned from the SQL Server provider are SMO-based. You can do a search on "SQL SMO objects" and see the richness these objects can bring. We're familiar with objects in SQL Server because we deal with tables, columns, and indexes. SMOs represent SQL Server objects and have properties and methods to interact with objects in SQL Server, such as dropping an object, getting properties of an object, and altering an object. This provider becomes powerful when

automating certain processes or information-gathering procedures by simplifying the syntax to get these objects. Let's dive in and learn how to use this provider's power.

23.3 Using the SQL Server provider

The SQL Server provider is exposed as a PSDrive (PowerShell Drive) by using paths into the hierarchy of SQL Server objects. A PSDrive is a way to access items in a way that's similar to a directory structure. After you load the provider you can use the PowerShell command `Get-PSDrive` to show all the drives available for PowerShell to reference in a fashion similar to a file system.

The list of drives includes a SQLSERVER: drive when the provider is loaded. This drive begins the process of accessing SQL Server objects through a series of paths. Table 1 shows the drive structure and what each level represents.

Table 23.1 SQL Server provider paths

Path	Description
SQLSERVER:	The drive you use to access SQL, just as you would use C:. The root of this drive contains the following paths to explore: SQL SQLPolicy SQLRegistration DataCollection Utility DAC
SQLSERVER:\SQL	The root of the SQL services on the local machine, and the beginning of the path in SQL Server via the provider.
SQLSERVER:\SQL\Computer-Name	The beginning of SQL Server's journey in the provider. The computer name is the next part of the path and it can be local or remote. This doesn't include the instance name (default or named). Executing `Get-ChildItem` gets information about *all* instances on this machine, including the default, and shows you their properties.
SQLSERVER:\SQL\Computer-Name\Instance	The path that connects you to the instance of SQL Server and tries to log you in via your Windows credentials. The following folders are available: Audits BackupDevices Credentials CryptographicProviders Databases Endpoints JobServer Languages LinkedServers Logins Mail ResourceGovernor Roles ServerAuditSpecifications SystemDataTypes SystemMessages Triggers UserDefinedMessages

With an understanding of this information you can begin to use the SQL Server provider in a powerful way. The information in table 1 will be a valuable reference for you regarding where you can go in SQL Server because most of the objects are represented in the provider and SMO.

As I've said, the real power of the provider syntax is that it's like a directory structure. Think of what the path to a table would look like. Listing 1 demonstrates the basic use of the provider from a console or the Integrated Scripting Environment (ISE); this can eventually be wrapped in a function, where you can pass parameters for the server, instance, and other parameters. It also shows using path-like structures in PowerShell with the SQL Server provider. You can extend this to your advantage in other pieces of automation.

Listing 23.1 Path-like access to SQL objects

```
$server = "localhost"
$instance = "default"
$dbname = "AdventureWorks"
$tblname = "HumanResources.Employee"

$path="SQLSERVER:\SQL\$server\$instance\Databases\$dbname\Tables\$tblname"
If(Test-Path $path)
{
  Get-Item $path
}
```

The more you use the SQL provider the more you'll want to become familiar with the paths that exist in the provider if you're planning to do any work in SQL Server with PowerShell.

23.4 Examples of using the SQL Server provider

Let's get some objects and see what you can do with this tool. Listing 2 shows a function that prepares the provider for use in the various versions of SQL Server; this function is reused throughout this chapter. It includes an example of using the provider to get information from SQL Server. The listing uses code from http://mng.bz/4sXz to load the assemblies so the function is reusable.

Listing 23.2 Function to load the SQL Server provider

```
function Load-SQLSnapins
{
   [CmdletBinding()]
   Param()

    $ErrorActionPreference = "Stop"

    $sqlpsreg="HKLM:\SOFTWARE\Microsoft\PowerShell\1\ShellIds\
      ➥ Microsoft.SqlServer.Management.PowerShell.sqlps"

    if (Get-ChildItem $sqlpsreg -ErrorAction "SilentlyContinue")
    {
        throw "SQL Server Provider for Windows PowerShell is not installed."
```

```
    }
    else
    {
        $item = Get-ItemProperty $sqlpsreg
        $sqlpsPath = [System.IO.Path]::GetDirectoryName($item.Path)
    }

    Set-Variable -scope Global -name SqlServerMaximumChildItems -Value 0
    Set-Variable -scope Global -name SqlServerConnectionTimeout -Value 30
    Set-Variable -scope Global -name SqlServerIncludeSystemObjects -Value
     $false
    Set-Variable -scope Global -name SqlServerMaximumTabCompletion -Value
     1000

    Push-Location
    cd $sqlpsPath

   if (!(Get-PSSnapin -Name SQLServerCmdletSnapin100 `
-ErrorAction SilentlyContinue))
   {
      Add-PSSnapin SQLServerCmdletSnapin100
      Write-Verbose "Loading SQLServerCmdletSnapin100..."
   }
   else
   {
      Write-Verbose "SQLServerCmdletSnapin100 already loaded"
   }

   if (!(Get-PSSnapin -Name SqlServerProviderSnapin100 `
-ErrorAction SilentlyContinue))
   {
      Add-PSSnapin SqlServerProviderSnapin100
      Write-Verbose "Loading SqlServerProviderSnapin100..."
   }
   else
   {
      Write-Verbose "SqlServerProviderSnapin100 already loaded"
   }

    Update-TypeData -PrependPath SQLProvider.Types.ps1xml
    update-FormatData -prependpath SQLProvider.Format.ps1xml
    Pop-Location
}

<#
namespaces based on
http://msdn.microsoft.com/en-ca/library/ms182491(v=sql.105).aspx

SQL2005
root\Microsoft\SqlServer\ComputerManagement"

SQL2008
root\Microsoft\SqlServer\ComputerManagement10"

SQL2012
\\.\root\Microsoft\SqlServer\ComputerManagement11\instance_name
```

```
#>
function Prepare-SQLProvider
{
   [CmdletBinding()]
   Param()
   $namespace = "root\Microsoft\SqlServer\ComputerManagement"
   if ((Get-WmiObject -Namespace $namespace -Class SqlService `
-ErrorAction SilentlyContinue)
   )
   {
      Write-Verbose "Running SQL Server 2005"
      #load Snapins
      Load-SQLSnapins
   }
   elseif ((Get-WmiObject -Namespace "$($namespace)10" -Class SqlService `
-ErrorAction SilentlyContinue))
   {
      Write-Verbose "Running SQL Server 2008/R2"
      #load Snapins
      Load-SQLSnapins
   }
   elseif ((Get-WmiObject -Namespace "$($namespace)11" -Class SqlService `
-ErrorAction SilentlyContinue))
   {
      Write-Verbose "Running SQL Server 2012"
      Write-Verbose "Loading SQLPS Module ... "
      Import-Module SQLPS
   }
}
```

Listing 3 shows how to get a list of the database names on your server. It's simple if you think of your SQL Server like a file system. For each file in a file system, properties give information about that file. Similarly, in the SQL Server provider you can access your databases like you do files in a directory.

Listing 23.3 Displaying a list of database names from SQL Server

```
Prepare-SQLProvider
cd SQLSERVER:\SQL\localhost\default
cd Databases
Get-Childitem | Select Name
```

The example in listing 4 takes you a little further into the hierarchy to get a list of tables. This isn't much harder than the previous example, because the Tables folder is another level in the hierarchy. The path is similar to SQLSERVER:\SQL\localhost\default\Databases\AdventureWorks\Tables. You can either use `Get-ChildItem` to get the tables or you can use `Where-Object` to filter them by property. In this case, you need to use the `Where-Object` because the SQL Server provider doesn't have support for filters. Figures 23.1 and 23.2 show the output from listing 4.

Listing 23.4 Getting a list of tables

```
Prepare-SQLProvider
CD SQLSERVER:\SQL\localhost\default\Databases\AdventureWorks\Tables
Get-ChildItem | Select DisplayName
Get-ChildItem | Where-Object { $_.DisplayName -match "HumanResources[.]" |
 Select DisplayName
```

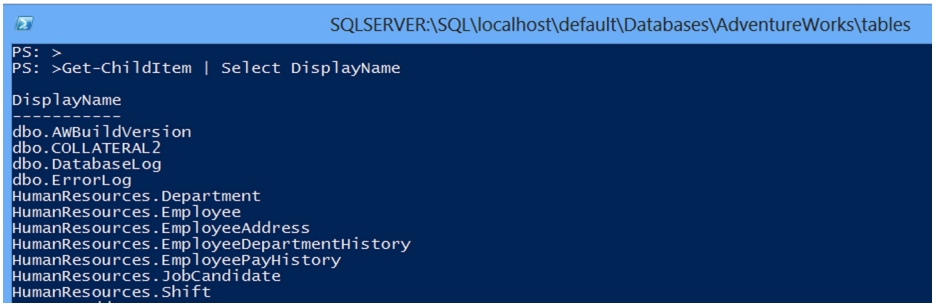


Figure 23.1 Output of getting a list of tables

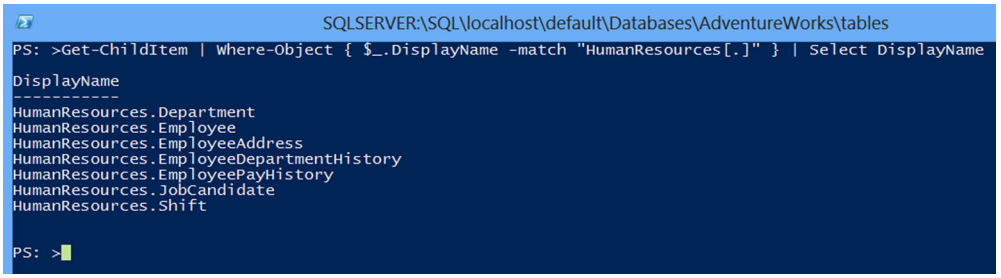


Figure 23.2 Output of the second command with the `Where-Object` clause

Last but not least, when you aren't in the mood or can't use the provider to get information but you need to use some of the objects it provides you can take advantage of the fact that the return objects are SMO-based. You use a server object to get some properties, or when you need access to the server object later in your code you can use `Get-Item` and the provider path to the server to get a server object. This is illustrated in the following listing. Figure 23.3 shows the output.

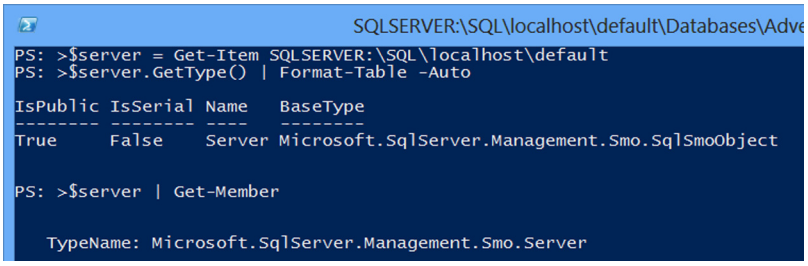


Figure 23.3 Using `Get-Item` to get an SMO server object

Listing 23.5 Getting a server object using the SQL Server provider

```
Prepare-SQLProvider
$server = Get-Item SQLSERVER:\SQL\localhost\default
$server.GetType() | Format-Table -Auto
$server | Get-Member
```

Notice in figure 23.3 that you see the type: the server object is a `Microsoft.SqlServer.Management.Smo.SqlSmoObject`. More specifically, in the second statement it's a `Microsoft.SqlServer.Management.Smo.Server` object. You can use this approach with databases, tables, and stored procedures to get and manipulate objects, all in a path to the object.

23.5 Getting a count of databases in an instance

The next listing uses the SQL Server provider to get a count of databases using functions to load the provider for whichever version of SQL Server is installed.

Listing 23.6 Get-DatabaseCounts function

```
function Get-DatabaseCounts
{
    [CmdletBinding()]
    Param(
       [Parameter(Position=0,Mandatory=$true)]
       [alias("server")]
       [string]$serverName,

       [Parameter(Position=1,Mandatory=$true)]
       [alias("instance")]
       [string]$instanceName
    )

   $results = @()
   (Get-Item SQLSERVER:\SQL\$serverName\$instanceName).Databases |
   Foreach-Object {
       $db = $_

       $db.Tables |
       Foreach-Object {
          $table = $_

          $hash = @{
             "Database"       = $db.Name
             "Schema"         = $table.Schema
             "Table"          = $table.Name
             "RowCount"       = $table.RowCount
             "Replicated"     = $table.Replicated
          }

          $item = New-Object PSObject -Property  $hash
          $results += $item

       }
   }
```

```
    $results
}

Prepare-SQLProvider -Verbose
Get-DatabaseCounts -server "localhost" -instance "DEFAULT" | Out-GridView
```

This listing shows the count of databases in the localhost\DEFAULT instance of SQL Server using the `Get-DatabaseCounts` function.

23.6 Finding a table in many databases

This use case is a common one when you're dealing with upgrades to a database or when you're deploying new code that relies on a new table that was created during development. There are different ways to find a table in the midst of many databases. Listing 7 shows a function that uses the provider to find the table, and listing 8 still uses the provider but with a script.

Listing 23.7 Finding the existence of a table in many databases

```
Function Get-SQLTableInDB {
    [CmdletBinding()]
    Param(
       [Parameter(Position=0,Mandatory=$true)]
       [alias("server")]
       [string]$serverName,

       [Parameter(Position=1,Mandatory=$true)]
       [alias("instance")]
       [string]$instanceName,

       [Parameter(Position=2,Mandatory=$true)]
       [alias("table")]
       [string]$tableName
    )

    (Get-Item SQLSERVER:\SQL\$serverName\$instanceName).Databases |
     Foreach-Object {
       $db = $_

       $db.Tables |
       Foreach-Object {

          $sqltable = $_
          If($tableName -eq $($sqltable.Name)) {
             Return $db.Name
          }
       }
     }
}

Prepare-SQLProvider
Get-SQLTableInDatabases -server "localhost" -instance "DEFAULT" `
-table "Table1"
```

Listing 23.8 Finding the existence of a table in many databases using the provider

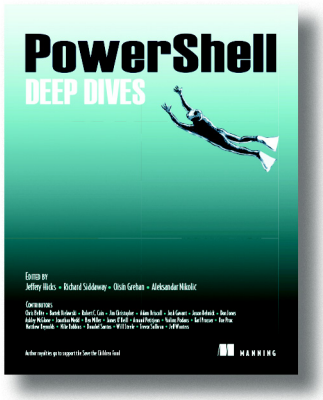
```
Prepare-SQLProvider

$servername = "localhost"
$instance = "default"
$tableName = "backupset"
$schema = "dbo"

$instpath = "SQL\$servername\$instance\Databases"
foreach($db in (Get-ChildItem SQLSERVER:\SQL\$instpath)) {
 $dbname = $db.Name
 if(!(Test-Path SQLSERVER:\$instpath\$dbname\Tables\$schema`.$tableName))
 {
   Write-Output $db.Name
 }
}
```

23.7 Summary

In this chapter you've seen how to get the SQL Server PowerShell provider for 2008/R2 and how to add it to your PowerShell session. The SQL Server provider for PowerShell is provided as a snap-in and is loaded with the `Add-PSSnapin` command; you access the structure of SQL Server using a path structure. You can add the provider to any Windows machine by downloading the PowerShell objects in the SQL Server Feature Packs.

Whether you're retrieving objects individually or detecting their existence a path structure provides a powerful way to use PowerShell and SQL Server together. This is just the tip of the iceberg when it comes to what you can do with the provider and how it all works, but I hope you caught the vision of where you can take it.

Chapter 25 discusses SMO and how to use objects in SQL Server with SMO; that chapter is a great companion to what you learned here. SQL Server 2012 wasn't covered in this chapter, but the concepts apply to the SQL Server 2012 provider; it's just loaded as a module (SQLPS) instead of a snap-in. Now, go execute some PowerShell!



Here's your chance to learn from the best in the business. *PowerShell Deep Dives* is a trove of essential techniques, practical guidance, and the expert insights you earn only through years of experience. Editors Jeffery Hicks, Richard Siddaway, Oisin Grehan, and Aleksandar Nikolic hand-picked the 28 chapters in the book's four parts: Administration, Scripting, Development, and Platforms.

PowerShell has permanently changed Windows administration. This powerful scripting and automation tool allows you to control virtually every aspect of Windows and most Microsoft servers like IIS and SQL Server.

PowerShell Deep Dives is a trove of essential techniques and practical guidance. It is rich with insights from experts who won them through years of experience. The book's 28 chapters, grouped in four parts (Administration, Scripting, Development, and Platforms), were hand-picked by four section editors: Jeffery Hicks, Richard Siddaway, Oisín Grehan, and Aleksandar Nikolic.

Whether you're just getting started with PowerShell or you already use it daily, you'll find yourself returning to this book over and over.

What's inside:

- Managing systems through a keyhole
- The Ten Commandments of PowerShell scripting
- Scalable scripting for large datasets
- Adding automatic remoting
- Provisioning web servers and websites automatically to IIS 8
- And 23 more fantastic chapters

IIS Administration

IIS has many configuration settings. Scripting provides a way to set up multiple IIS instances with identical configurations. This chapter provides an excellent introduction to IIS administration via PowerShell. It also gives a good practical example – standing up a web farm of four web servers. You're shown how to perform the individual tasks and then combine the code to produce a script you can reuse as many times as required.

Chapter 27 from *PowerShell Deep Dives* edited by Jeffrey Hicks, Richard Siddaway, Oisin Grehan, and Aleksander Nikolic

Provisioning IIS web servers and sites with PowerShell

The following scenario is common if you're an administrative web master, and here's how it was delivered to me: "Deploy a highly available web farm (four servers) with a couple of websites, including certificates, for a new secure shopping site. Make sure to enable graphical remote management for IIS Manager so that other admins and developers can make changes; and, by the way, did we mention we're moving to Windows Server 2012 Core?" (See figure 27.1.)

This isn't a complicated project, thanks to the support of PowerShell and the Internet Information Services (IIS) cmdlets, but you may encounter tricky spots and gotchas along the way.

Initially I solved this problem by using PowerShell interactively to complete the required tasks. As a smart and lazy admin, I saved the commands to a script so that in the future I could automate similar deployments without all the typing. I even turned some of the tasks into advanced functions so that other admins could accomplish some of the trickier stuff.

In this chapter you'll see how I interactively solved this deployment scenario, and I'll also show you how to automate it. The entire process from beginning to end involves these tasks:

- Deploy IIS to the Windows Server 2012 Core remote servers.
- Prepare the remote servers with website files and certificates.
- Enable remote-management support for the graphical IIS Manager.
- Create a load-balanced web farm.

- Create a secure load-balanced website using Secure Sockets Layer (SSL).
- Automate the process.

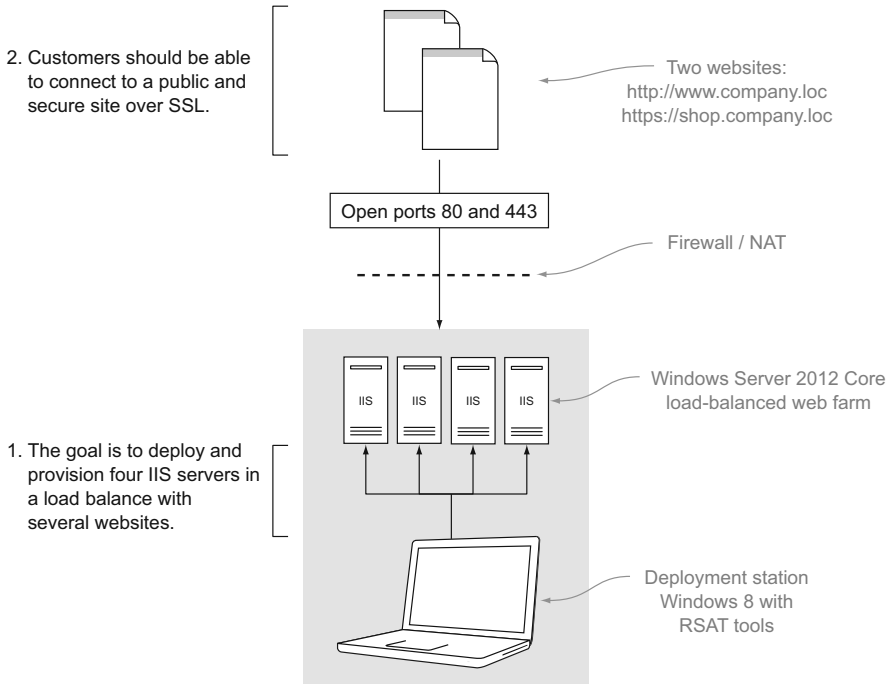


Figure 27.1 The deployment goal of a web farm with multiple websites

Setting up the lab environment

I created a lab environment to write this chapter. If you want to follow along you can create a similar environment.

Although I'm using Windows Server 2012 Core, this deployment solution also works on Windows Server 2008 R2, with or without a graphical desktop. I use some of the newer networking commands from Server 2012 for the Domain Name System (DNS) settings, but if you're using Windows Server 2008 R2 you can work around that with the GUI. I also use the dynamic module-loading feature in PowerShell v3; if you're using PowerShell v2 I'll warn you when you need to import a module.

These are the items that I set up in advance:

Deployment station—Windows 8 Pro running Remote Server Administration Tools (RSAT). I'll use local RSAT cmdlets in this chapter.

Four Windows Server 2012 Core servers—Each server is assigned an IP address and is a member of the domain, although this is not required for middle-tier web servers. You can set up the IP address through SConfig.cmd or the networking cmdlets.

Remoting—This feature is enabled for all Windows Server 2012 products; you'll need to enable it if you're using Windows Server 2008 R2. (This is a requirement.)

Script execution—This should be enabled on the servers.

SSL certificate—For a lab environment you can use a self-signed certificate, but for production use a good web server certificate or even an Extended Validation (EV) certificate. I created a certificate in Active Directory Certificate Services (AD CS) and exported it to a Personal Information Exchange (.pfx) file.

Let's get started and deploy the web servers and websites.

27.1 Rapid IIS deployment

To begin the deployment we'll use PowerShell Remoting to connect to the remote servers. Some tasks won't be completed over remoting, so store a list of computer names in a variable that you can pipe to commands.

1. Gather the computer names of the future web servers and store them to a variable, `$Servers`, using one of the following options.

   If the servers are members of the domain, use the Active Directory cmdlet `Get-ADComputer`:

```
PS> $Servers = Get-ADComputer -Filter "name -like 's*'" |
  ➥ Select-Object -ExpandProperty name
```

NOTE If you're using PowerShell v2, be sure to import the `Active Directory` module first.

   You can also get the list from a CSV or TXT file:

```
PS> $Servers = Import-Csv c:\servers.csv |
  ➥ Select-Object -ExpandProperty ComputerName

PS> $Servers = Get-Content c:\servers.txt
```

2. Create a PowerShell remote session to the servers. Store the sessions in a variable `$Sessions` for easy access later:

```
PS> $Sessions = New-PSSession -ComputerName $Servers
```

3. Determine what software is needed to support all of the tasks for this project.

   The remote servers require the following roles and features for this deployment solution, but you can add to the list if you need additional components to support your websites:

   - *Web Server (IIS) (`web-server`)*—The primary role for a web server. This installs the components of IIS and creates the default website.
   - ASP.NET *(`web-asp-net`)*—Provides support for ASP.NET websites.
   - *Network Load Balancing (NLB)*—I'm using Microsoft's built-in layer-3 NLB software. You can substitute your own hardware load balancer or Microsoft's layer-7 Application Request Routing (ARR) balancer. ARR has cmdlets for easy management and is one of my favorite products. ARR also includes

additional features beyond load balancing but requires greater in-depth knowledge, so I'm sticking with the straightforward, built-in, and useful Microsoft NLB.

– *Management Service* (`Web-Mgmt-Service`)—Required component for remote management of IIS with IIS Manager.

4 Install the required components on the remote servers with `Invoke-Command`:

```
PS> Invoke-Command -Session $Sessions {Install-WindowsFeature
  ➥ web-server,web-asp-net,NLB,Web-Mgmt-Service}
```

PowerShell v2 notes

If you're using PowerShell v2 on Server 2008 R2 you'll need to import the `Server Manager` module first:

```
Invoke-Command -Session $Sessions {Import-Module ServerManager}
```

Also I'm using the new `Install-WindowsFeature` cmdlet. In PowerShell v2 use the `Add-WindowsFeature` cmdlet.

Installing the software components to all four servers, as shown in figure 27.2, takes only a few minutes (5 minutes to be exact).

The IIS installation process creates the default website automatically. Let's test this default website on each server before continuing with the next task.

Testing ensures that the web server is functioning properly and reduces future troubleshooting if something goes wrong:

5 Use the `$Servers` variable to pipe the server names to Internet Explorer:

```
PS> $Servers | ForEach-Object {Start-Process iexplore http://$_}
```

Four separate browsers automatically launch and test the default website on each individual server.

With the initial software deployment completed the next task is to deploy (copy) the website files and certificate out to the servers. PowerShell makes this a snap.

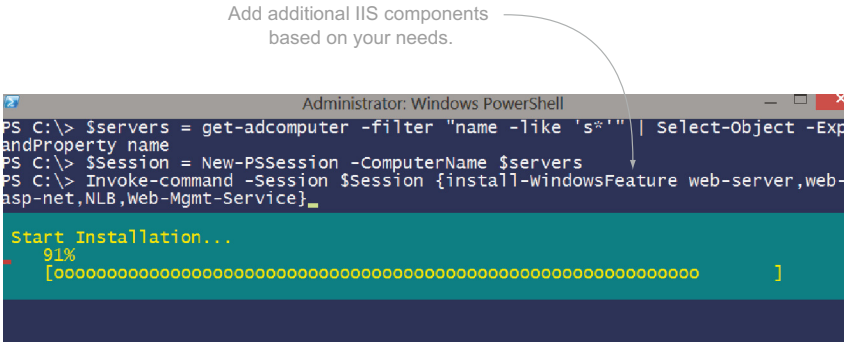


Figure 27.2 Performing a rapid install of the required software on multiple servers

27.2 Transferring website files and certificates

IIS supports storing your website files and applications on a central share from a clustered file server. Some organizations, such as small companies, don't have this capability, so we'll copy the websites from a central location (my computer) out to the individual web servers. Because these web servers will be load-balanced, each server needs to have the same files.

DEPLOYING THE DEFAULT WEBSITE

1. Copy the new default website to each web server's c:\inetpub\wwwroot path:

```
PS> $Servers | ForEach-Object {Copy-Item -Path c:\sites\www\*.*
  ➥ -Destination "\\$_\c$\inetpub\wwwroot"}
```

2. Test the default website after the file transfer (see figure 27.3):

```
PS> $Servers | ForEach-Object {Start-Process iexplore http://$_}
```

With the default website successfully deployed we can focus on the new secure shopping site.

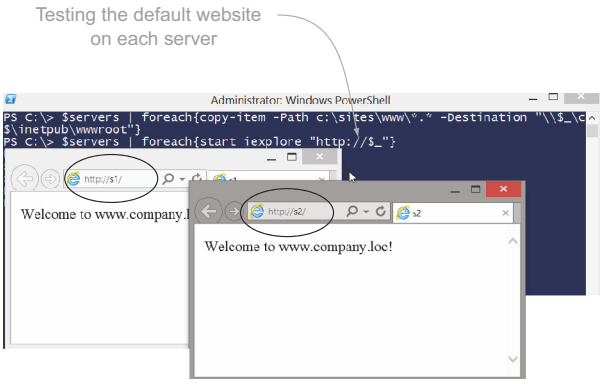


Figure 27.3 Successful deployment of the default website to multiple web servers

DEPLOYING THE SHOPPING WEBSITE

Most of the websites that you'll copy out to the web servers won't be in the default path (InetPub). I prefer to use a directory called sites, with each website in its own folder:

1. Create the folder structure on the remote servers (C:\sites\shopping), and then copy the new website:

```
PS> Invoke-Command -Session $Sessions {New-Item -Path c:\sites\shopping
  ➥ -ItemType directory -Force}
PS> $Servers | ForEach-Object {Copy-Item -Path c:\sites\shopping\*.*
  ➥ -Destination "\\$_\c$\sites\shopping"}
```

2. Generate a certificate for SSL for the secure shopping site.

   (I previously generated and stored a trusted certificate on my local Windows 8 computer in c:\sites\certpfx.)

3 Copy the certificate to the remote servers, and then use CertUtil.exe to import the certificate:

```
PS> $Servers | ForEach-Object {Copy-Item -Path c:\sites\certpfx\*.*
  ➥ -Destination "\\$_\c$"}
PS> Invoke-Command -Session $Sessions {certutil -p P@ssw0rd
  ➥ -importpfx c:\company.loc.pfx}
```

I sent the password in clear text because PowerShell Remoting is secure and encrypted. I wouldn't do this in a script. The certificate imports successfully, as shown in figure 27.4.

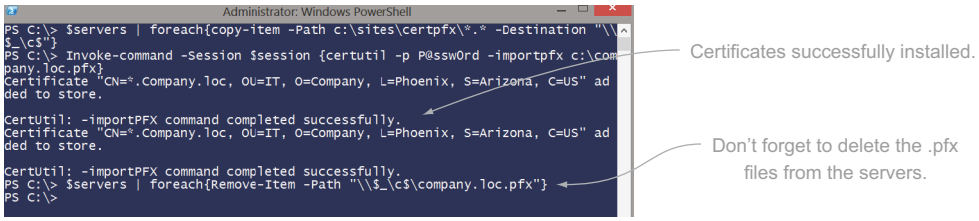


Figure 27.4 Deploying and installing a certificate for SSL

4 Remove (delete) the .pfx file from the remote servers:

```
PS> $Servers | ForEach-Object {Remove-Item -Path
   "\\$_\c$\company.loc.pfx"}
```

The website files are copied to the remote servers and each server has the certificate for the secure site. Before you finish creating and configuring the secure site you need to enable IIS remote management so that the websites can be managed using IIS Manager.

27.3 Enabling remote management for IIS Manager

IIS remote management adds the capability of managing websites on remote servers from IIS Manager. It's best to enable and configure this feature using IIS Manager run locally on each server; it's not a friendly feature to enable through the command line or on Windows Server 2012 Core. In addition, we need to replace the temporary, self-signed certificate, which is assigned to remote management.

Let's break this into two steps: enabling the service and replacing the certificate.

ENABLING THE SERVICE

1 Enable the remote management service in the registry, and then start the Web Management Service (WMSVC).

WMSVC has a startup type of `Manual`, so change the startup to `Automatic` before starting the service:

```
PS> Invoke-Command -Session $Sessions {Set-ItemProperty
  ➥ -Path HKLM:\SOFTWARE\Microsoft\WebManagement\Server
  ➥ -Name EnableRemoteManagement -Value 1}
PS> Invoke-Command -Session $Sessions {Set-Service wmsvc
  ➥ -StartupType Automatic}
PS> Invoke-Command -Session $Sessions {Start-Service wmsvc}
```

Figure 27.5 illustrates the successful start of WMSVC on the remote computers.

At this point you can connect IIS Manager to the remote computers, but you can't use IIS Manager to manage and change the certificates for the remote service.

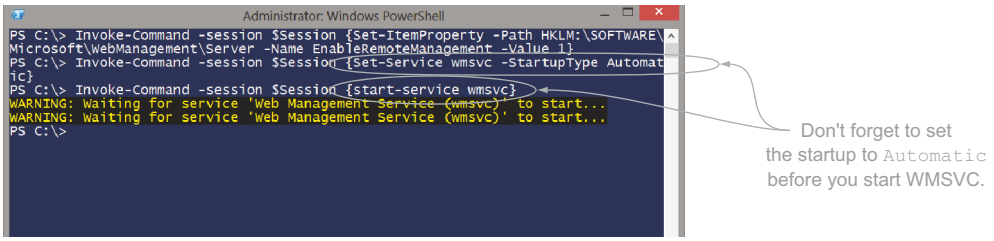


Figure 27.5 Enabling IIS remote management and starting WMSVC

REPLACING THE CERTIFICATE

The IIS remote management service uses port 8172 and binds a temporary certificate to "all unassigned" IP addresses. You need to change this binding, and this is where things get a little strange. To remove the old SSL binding for port 8172 and add a new one you need to access the `IIS:` provider. Because PowerShell cmdlets and this provider don't work together as well as they could, extra steps are required to complete the process:

1. Get the thumbprint of the trusted certificate that you imported previously and store it to a variable (`$cert`).

   Perform this step over PowerShell Remoting so that the variable can be used for later commands:

   ```
   PS> Invoke-Command -Session $Sessions {$cert = Get-ChildItem
     ➥ -Path Cert:\LocalMachine\My | where {$_.subject -like "*company*"} |
     ➥ Select-Object -ExpandProperty Thumbprint}
   ```

2. Access the IIS: drive.

   When IIS is installed, a module called `WebAdministration` is added, which includes cmdlets and an IIS: provider. To ensure that the provider is loaded, import the `WebAdministration` module:

   ```
   PS> Invoke-Command -Session $Sessions {Import-Module WebAdministration}
   PS> Invoke-command -Session $Sessions {cd IIS:\SslBindings}
   ```

   Bindings are stored in IIS:\SslBindings as path items.

3. Remove the binding that contains the temporary certificate:

   ```
   PS> Invoke-command -Session $Sessions {Remove-Item -Path
     ➥ IIS:\SslBindings \0.0.0.0!8172}
   ```

NOTE Usually IIS binding information is entered and displayed as `IPaddress :port:hostname`, as in `*:80:*`, but PowerShell interprets the colon (`:`) as a path indicator. When using the cmdlets to work with bindings for IIS, replace the colon with an exclamation mark (`!`), as in `*!80!*`.

4 Create a new binding that uses the new trusted certificate.

Use the `Get-Item` command to retrieve the correct certificate based on the thumbprint stored in `$cert`. The certificate is piped to `New-Item`, which creates the new binding for all IP addresses on port 8172:

```
PS> Invoke-Command -Session $Sessions {Get-Item
  ➥ -Path "cert:\localmachine\my\$cert" |
  ➥ New-Item -Path IIS:\SslBindings\0.0.0.0!8172}
```

5 Start IIS Manager (`PS> Start inetmgr`), and create connections to the remote servers as shown in figure 27.6.

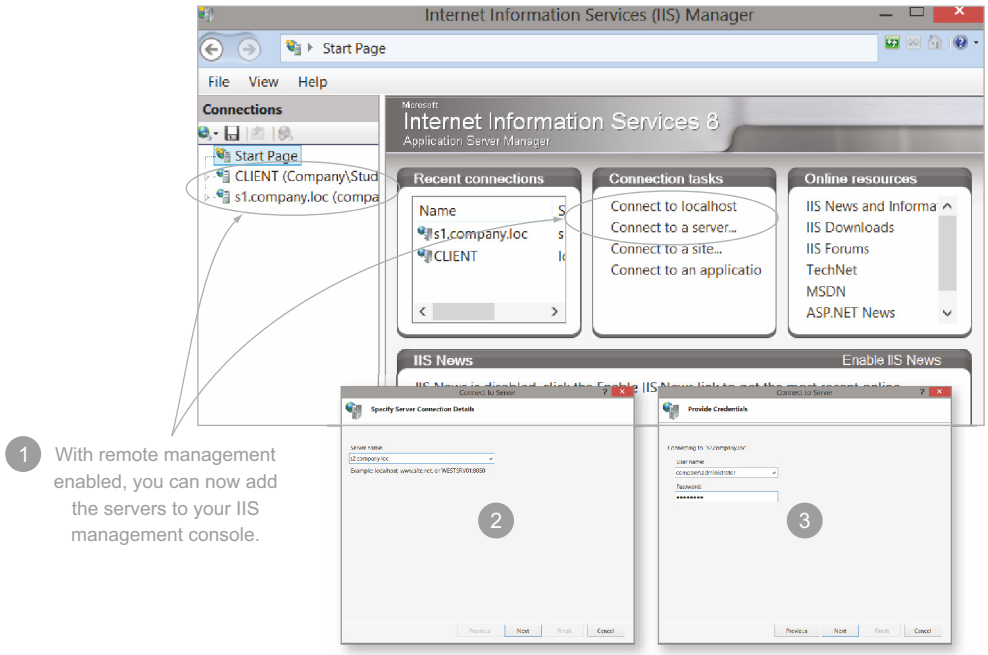


Figure 27.6 Adding the remote servers to IIS Manager

With the remote management capabilities of IIS enabled we can finish off our deployment and provisioning web server project with two final tasks: building the web farm and creating a new secure website. Let's start with the web farm.

27.4 Creating a load-balanced web farm

For many companies a hardware load balancer that provides high availability is the only choice for their web farms; it's fast, efficient, and provides certificate management. Not everyone can afford (or even needs) this level of performance, so other options are available. My favorite is the layer-7 load balancer for IIS from Microsoft called Application Request Routing (ARR). It's free, an excellent product, can be downloaded from www.iis.net, has cmdlets for management, includes many more

features in addition to load balancing, and, did I mention, it's free. ARR performs load balancing using URL rewrite. Because URL rewrite is complex and requires in-depth knowledge of ARR I chose to use the built-in Microsoft NLB for this example deployment situation. NLB works well and doesn't require the additional installation and knowledge overhead to make a great solution.

For this task I'm using the cmdlets from the `NLB` module on my Windows 8 computer. Alternatively you could issue these commands over PowerShell Remoting:

1. Create the load balance on server `S1` with the `New-NlbCluster` cmdlet, and create a cluster IP address for the default website:

```
PS> New-NlbCluster -HostName s1 -InterfaceName Ethernet -ClusterName web
  ➥ -ClusterPrimaryIP 192.168.3.200 -SubnetMask 255.255.255.0
  ➥ -OperationMode Multicast
```

2. Add another address with the `Add-NlbClusterVip` cmdlet:

```
PS> Get-NlbCluster -HostName s1 | Add-NlbClusterVip -IP 192.168.3.201
  ➥ -SubnetMask 255.255.255.0
```

   You'll use this additional cluster IP address for the secure website that you'll create in the next section.

3. Add the second server (`S2`) as a node in the load balance with the `Get-NlbCluster` cmdlet:

```
PS> Get-NlbCluster -HostName s1 | Add-NlbClusterNode -NewNodeName s2
  ➥ -NewNodeInterface Ethernet
```

4. Repeat step 3 for the other two servers in this scenario.

   The return information from the `Get-NlbCluster` cmdlet informs you if you have any problems converging the load balance.

5. Launch the graphical Network Load Balancing Manager (on a Windows 8 computer) from the Administrative Tools to verify the status (see figure 27.7).
6. Test the load balance with full name resolution.

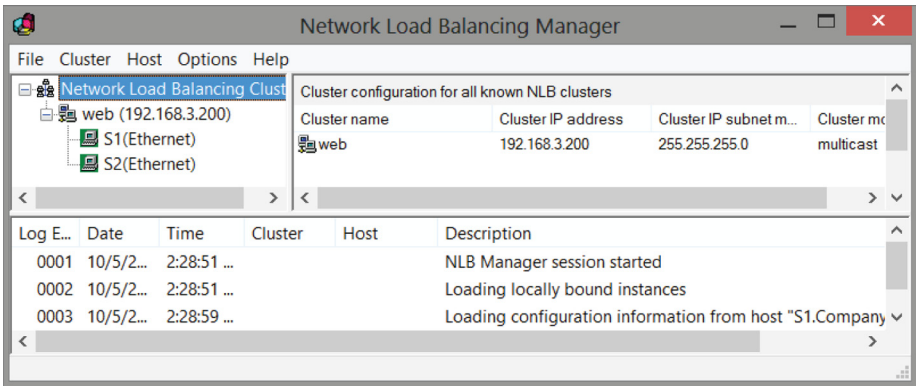


Figure 27.7 Verifying the load balance in the Network Load Balancing Manager

Create a www record in DNS that points to the cluster IP address, and then launch a browser using the new address:

```
PS> Add-DnsServerResourceRecordA -Name www -ZoneName company.loc
  ➥ -IPv4Address 192.168.3.200 -ComputerName DC.company.loc
PS> Start-Process iexplore http://www.company.loc
```

Finally, after all this work, it's time for the final task: creating a new and secure website for the web farm. Let's make a website!

27.5 Creating an SSL website

To make a new website on the remote servers use the IIS (web) cmdlets from the `WebAdministration` module. Remember that we already copied the files for this new website to the location c:\sites\shopping:

1. Create an application pool for the new website with the `New-WebAppPool` cmdlet:

```
PS> Invoke-Command -Session $Sessions
{New-WebAppPool -Name Shopping-Pool}
```

Figure 27.8 shows the graphical version of creating a pool in IIS Manager.

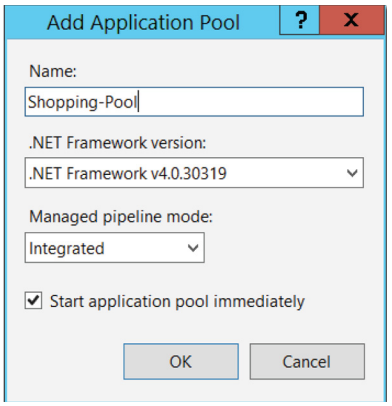


Figure 27.8 Creating a pool in IIS Manager

The new application pool is created with default settings for items such as the recycle times and identity. This is a good time to add your own application pool commands to alter those defaults, if desired. (See the sidebar for an example.)

Changing the application pool identity

Usually, for application pools of `ApplicationPoolIdentity`, the default identity is sufficient as a restricted identity. In cases where multiple customers have websites located on the same server (multitenant), isolating each pool with its own identity provides unique security for every customer. To set the pool identity IIS uses a number representing the identity. The default value is `4`, but if you want to have isolation you can create individual accounts and assign those accounts to each pool as in the following example:

```
LocalSystem = 0
LocalService = 1
NetworkService = 2
SpecificUser = 3
ApplicationPoolIdentity = 4
PS> Invoke-Command -Session $Sessions {Set-ItemProperty
  ➥ -Path IIS:\AppPools\MyTest -Name processmodel.identityType -Value 3}
```

```
PS> Invoke-Command -Session $Sessions {Set-ItemProperty
  ➥ -Path IIS:\AppPools\MyTest -Name processmodel.username
  ➥ -Value Administrator}
PS> Invoke-Command -Session $Sessions {Set-ItemProperty
  ➥ -Path IIS:\AppPools\MyTest -Name processmodel.password -Value
            P@ssw0rd}
```

2 Create a new website named `Shopping`.

After you create the application pool the `New-Website` cmdlet does the rest of the work:

```
PS> Invoke-Command -Session $Sessions {New-Website -Name Shopping
  ➥ -HostHeader shop.company.loc -PhysicalPath C:\sites\shopping
  ➥ -ApplicationPool Shopping-Pool -Port 443 -ssl -SslFlags 0}
```

The website has a host header of `shop.company.loc` and points to the physical location of the website files. The new site is assigned to the correct application pool and a binding on port 443 is set. The `-SslFlags` tells the website to use a normal certificate.

3 Create another SSL binding for the new site.

The process is the same as discussed previously, but the binding is for all IP addresses on port 443:

```
PS> Invoke-Command -Session $Sessions {$cert=Get-ChildItem
  ➥ -Path Cert:\LocalMachine\My | where {$_.subject -like "*company*"} |
  ➥ Select-Object -ExpandProperty Thumbprint}
PS> Invoke-Command -Session $Sessions {Import-Module WebAdministration}
PS> Invoke-Command -Session $Sessions {Get-Item
  ➥ -Path "cert:\localmachine\my\$cert" | New-Item -Path
  ➥ IIS:\SslBindings\0.0.0.0!443!Shop.company.loc}
```

As shown in figure 27.9, the new binding is successfully created on all remote servers.

4 Test the new website.

Add a DNS record that points to the cluster IP address previously defined for the website and then launch a browser using the address:

```
PS> Add-DnsServerResourceRecordA -Name shop -ZoneName company.loc
  ➥ -IPv4Address 192.168.3.201 -ComputerName DC.company.loc
PS> Start-Process iexplore https://shop.company.loc
```

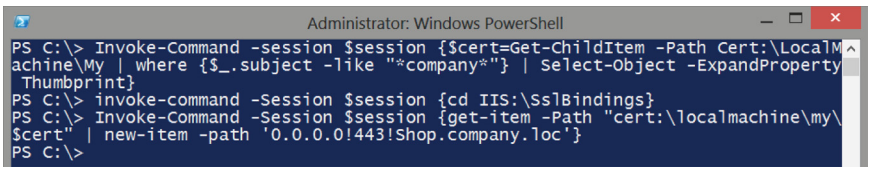


Figure 27.9 Successful creation of the new SSL binding

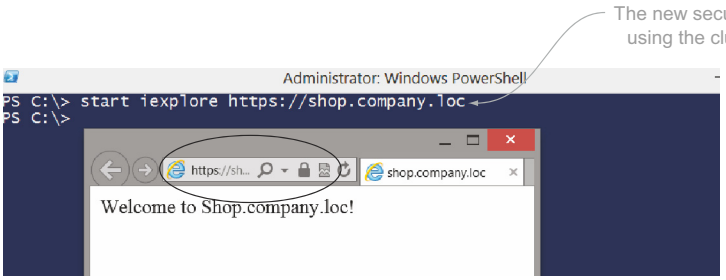


Figure 27.10 Successful test of the new website using SSL

As shown in figure 27.10, the new website successfully passes the test using the trusted certificate over SSL.

Total time for this project, using PowerShell interactively, is approximately 30 minutes. Storing these commands in a .ps1 file helps me script future deployment projects. Why do all that typing again? I wrote the tricky tasks, such as enabling remote management, as advanced functions so that other admins have the tools they need without all the hassle. I increased my value to the company and managed to get a little more time on the beach.

27.6 Automating the process

Automating the deployment process is as simple as sticking the commands in a script file, but I went further and built in more flexibility. PowerShell Remoting and the `Invoke-Command` cmdlet make life easy. For example, have you ever tried the switch option for `Invoke-Command -FilePath`? This switch option eliminates the need to copy scripts to remote computers before executing them. You write a script that performs the tasks as if it were running on the local computer. To send that script to your remote computers use `Invoke-Command`.

In this section I'll first show you the script that does the hard work, and then I'll show you how I call and use the script. The only changes from the commands you've already seen are the following:

- I removed all of the `Invoke-Command` cmdlets.
- I changed how the certificate password is passed to the script. I don't want the password hardcoded in the script, so I used a PowerShell v3 feature to pass a variable to the script with `$Using:CertPassword`.
- I left out the NLB commands, in case you already have a load-balance solution, but you can always add them.

Here's the script, which I named `Deploy-WebServer.ps1`.

Listing 27.1 Deploy-WebServer.ps1

```
Install-WindowsFeature web-server,Web-Mgmt-Service

Set-ItemProperty -Path HKLM:\SOFTWARE\Microsoft\WebManagement\Server `
-Name EnableRemoteManagement -Value 1

Set-Service wmsvc -StartupType Automatic
Start-Service wmsvc

certutil -p $Using:certPassword -importpfx c:\Wildcard.company.loc.pfx
Remove-Item -Path c:\Wildcard.company.loc.pfx

Import-module -Name WebAdministration
$cert = Get-ChildItem -Path Cert:\LocalMachine\My |
where {$_.subject -like "*company*"} |
Select-Object -ExpandProperty Thumbprint

Remove-Item -Path IIS:\SslBindings\0.0.0.0!8172

Get-Item -Path "cert:\localmachine\my\$cert" |
New-Item -Path IIS:\SslBindings\0.0.0.0!8172

New-WebBinding -Name "Default Website" -Protocol https

Get-Item -Path "cert:\localmachine\my\$cert" |
New-Item -Path IIS:\SslBindings\0.0.0.0!443
```

Installs required components

Enables remote management

Removes the certificate file

Creates new SSL binding

To use the Deploy-WebServer.ps1 script I run interactive commands to set up the remoting connections and set a few variables. Then I call the deployment script with a single `Invoke-Command` cmdlet:

1. Build a remote session to the computers that will become web servers.

   Put the server names in a variable—you'll need that later, so don't cheat and make this a one-liner:

   ```
   PS> $Servers='server1','server2', 'server3'
   PS> $Sessions=New-PSSession -ComputerName $Servers
   ```

2. Set a variable to contain the password to install the certificate.

   This information is passed over the remoting session encrypted:

   ```
   PS> $CertPassword="P@ssw0rd"
   ```

3. Interactively copy the website files and certificates to the remote servers:

   ```
   PS> $servers | ForEach-Object{New-Item -Path \\$_\C$\inetpub\wwwroot
     ➥ -ItemType Directory -Force}
   PS> $servers | ForEach-Object{Copy-Item -Path c:\sites\www\*.*
     ➥ -Destination \\$_\C$\inetpub\wwwroot -Force}
   PS> $servers | ForEach-Object{Copy-Item -Path c:\sites\CertPFX\*.*
     ➥ -Destination \\$_\C$\ -Force}
   ```

   If you put these commands in the Deploy-WebServer.ps1 file you'll run into a double-hop issue—the remote computers connecting to another remote server to get the files.

NOTE If you copy files to Windows Server 2012 Core you'll first need to install the `FS-FileServer` role to access the `C$` share.

4 Run the deployment script using the `-FilePath` parameter:

```
PS> Invoke-Command -Session $Sessions -FilePath C:\scripts\deploy-
  ➥ WebServer.ps1
```

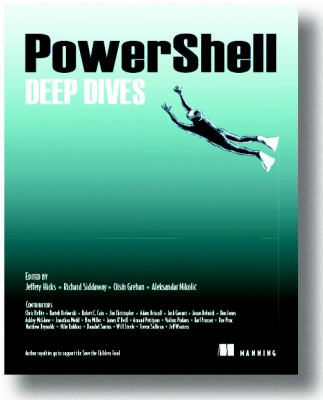
All the target servers now have a web server, website, and certificates installed and are ready for action!

27.7 Summary

This chapter covered the deployment of multiple web servers with multiple websites, which included building a web farm and installing certificates for SSL. The concepts and tactics demonstrated here could easily be applied to other roles, features, and products, such as SharePoint web servers and Client Access Server (CAS) arrays for Microsoft Exchange. I gleaned the following takeaways during this real-life project:

- I can use PowerShell interactively to solve each task, even for a more complicated deployment.
- There may not be specific cmdlets for every situation, such as enabling the remote management of IIS, but there are ways around those issues.
- PowerShell Remoting must be enabled to permit these larger-scale management solutions. While it's the default for Windows Server 2012, you need to enable it now even if you're not at that version yet.

Thanks to PowerShell I get an amazing amount of work done quickly and without traveling to a cold data center. If you have any questions about the script or commands I discussed in this chapter visit the forums at http://www.powershell.org, and I'll be happy to help!



Here's your chance to learn from the best in the business. *PowerShell Deep Dives* is a trove of essential techniques, practical guidance, and the expert insights you earn only through years of experience. Editors Jeffery Hicks, Richard Siddaway, Oisin Grehan, and Aleksandar Nikoli? hand-picked the 28 chapters in the book's four parts: Administration, Scripting, Development, and Platforms.

PowerShell has permanently changed Windows administration. This powerful scripting and automation tool allows you to control virtually every aspect of Windows and most Microsoft servers like IIS and SQL Server. *PowerShell Deep Dives* is a trove of essential techniques and practical guidance. It is rich with insights from experts who won them through years of experience. The book's 28 chapters, grouped in four parts (Administration, Scripting, Development, and Platforms), were hand-picked by four section editors: Jeffery Hicks, Richard Siddaway, Oisín Grehan, and Aleksandar Nikolic.

Whether you're just getting started with PowerShell or you already use it daily, you'll find yourself returning to this book over and over.

What's inside:

- Managing systems through a keyhole
- The Ten Commandments of PowerShell scripting
- Scalable scripting for large datasets
- Adding automatic remoting
- Provisioning web servers and websites automatically to IIS 8
- And 23 more fantastic chapters

AD Administration

I've probably done more PowerShell scripting against Active Directory than any other target. AD management, user management in particular, is often the first target for automation in most organizations. This chapter introduces that topic and gives an example of one of PowerShell's strengths, namely that there's often more than one way to accomplish a task. If you can't get access to the AD cmdlets this chapter shows how to perform the common user administration tasks through scripting.

Chapter 5 from *PowerShell in Practice* by Richard Siddaway

User accounts

This chapter covers

- Automating AD user accounts
- Searching Active Directory
- Creating and modifying group memberships
- Group nesting

"Working with users" is the title of the middle part of this book. Anyone who thought "It would be a nice job but for the users" should be ashamed, very ashamed. Write out 100 times "I mustn't say things like that again." Better still, create a PowerShell script to write it out. There'll be a test.

A large part of administration comes back to users, directly or indirectly. In this chapter, we'll be automating the administration of user accounts. Why do we want to do this? Look back at my example from chapter 4. Do you want to set up 7,000+ users in a few weeks? Automation all the way.

The other reason for automating user account management is consistency. When working as a consultant, I've seen Active Directory implementations where the names are created in every combination you can think of. First name first; sur-

name first; various combinations of commas and spaces between the name parts. Commas should be avoided if possible, as they have to be allowed for in the script; otherwise the user account won't be found. The rest of the account information is just as inconsistent, with missing or wrong telephone numbers, addresses, and so on. Consistency makes things easier to administer. Be consistent. How do you do that? Automation all the way. Another thing we need to consider is groups. Allocating permissions by groups is best practice in a Windows environment, so we need to know how to create and modify groups.

The chapter will start with a look at the options we have for working with user accounts and groups. In this chapter, most of the scripts will be presented in two variations in order to provide the maximum flexibility. After explaining which options will be used, we'll look at how we work with local users and groups, including creation and modification.

Working with Active Directory users and groups occupies the bulk of the chapter. We start at the logical place by creating a user account. One of the major differences between working locally and working with Active Directory is that with the latter, we're often working with multiple users simultaneously. This will be illustrated by looking at how we can create users in bulk. Not quite on the scale of 7,000 at a time, but we could scale if required. Having created our users, we need to think about modifications to various attributes together with how we move the account to a different Organizational Unit (OU). During the move, the account may need to be disabled. This is a common scenario for dealing with people leaving the organization.

We often need to search Active Directory to find a particular user or possibly to find accounts or passwords that are about to expire. One common need is to discover a user's last logon time. This can be useful for checking who's still active on our directory. I recently checked an AD installation where there were several hundred accounts that hadn't been used for over six months. The disposal of old accounts can, and should, be automated.

The final section of the chapter deals with Active Directory groups. After a group has been created, we'll definitely need to modify its membership and may need to change its scope—the last type of group. We complete the section by answering two questions: "Who's in this particular group?" and "What groups is this user in?" These are questions that can't be easily answered by using the GUI tools.

In order to perform these tasks, we need to use ADSI, which is the primary interface for working with Active Directory, as we saw in chapter 3. There are a few options to consider regarding the exact way we accomplish this before we start creating scripts.

5.1 Automating user account management

Before the release of Windows Server 2008 R2, we'd work with user accounts via ADSI, as we saw in chapter 4. This can be performed in a number of ways, including:

- `[ADSI]` type accelerator
- `System.DirectoryServices` .NET classes

- `System.DirectoryServices.AccountManagement` .NET classes
- Quest AD cmdlets (the nouns all start with QAD)

Windows Server 2008 R2 introduced a module containing Active Directory cmdlets (see section 5.1.2).

> **POWERSHELL DILEMMA** This illustrates the dilemma that many new PowerShell users face. "I've found three different ways of performing this task: which one should I use?" The short-term answer, especially if you're new to PowerShell, is whichever one you feel most comfortable with. In the longer term, investigate the possibilities, pick one, and stick with it. One slight problem is that sometimes you need to use multiple methods to cover all eventualities.

ADSI can be used to access AD LDS, previously known as ADAM, via PowerShell in a similar way to Active Directory. The only major change is the way you connect to the directory service. The code to get a directory entry for an Active Directory user is:

```
$user = [ADSI]"LDAP://cn=Richard,cn=Users,dc=Manticore,dc=org"
```

To connect to an AD LDS or ADAM instance, this changes to:

```
$user = [ADSI]
"LDAP://server_name:port/cn=Richard,cn=Users,dc=Manticore,dc=org"
```

If the AD LDS/ADAM instance is on the local machine, this becomes:

```
$user = [ADSI]
"LDAP://localhost:389/cn=Richard,cn=Users,dc=Manticore,dc=org"
```

5.1.1 Microsoft AD cmdlets

When a Windows Server 2008 R2 domain controller is created, a module of Active Directory cmdlets is installed. Modules are covered in more detail in chapter 15 and appendix B. This module can also be installed on Windows Server 2008 R2 servers or Windows 7 machines (using the RSAT download). The module isn't loaded by PowerShell by default. We use:

```
Import-Module ActiveDirectory
```

The Microsoft AD cmdlets work in a slightly different manner, in that they access a web service running on the domain controller. This performs the actions against Active Directory. The web service is available for installation on Windows Server 2008 or Windows Server 2003 domain controllers, but we'll need a Windows Server 2008 R2 or Windows 7 machine to install and run the cmdlets. The PowerShell v2 remoting capabilities can be used to set up proxy functions for these cmdlets on any machine running PowerShell v2. This technique is described in chapter 13.

A similar approach is taken with Exchange 2010, in that remote access is provided by a web service. These two systems are examples of a "fan-in" administrative model, in that many administrators can connect to the same machine to perform their jobs. Contrast this with the approach we'll see with IIS in chapter 13, where one administrator can work on multiple machines. PowerShell provides many ways to remotely

administer our systems. The Active Directory cmdlets interacting with a web service is just one example. The need to install something on the domain controller may be viewed as a negative, in which case the Quest cmdlets could be used, as they only need to be installed on the machine used for administration.

5.1.2 Recommendations

When working with Active Directory and PowerShell, we have two main choices: use scripts or use the AD cmdlets from Microsoft or Quest. My preference is to use the cmdlets, but I realize that they aren't available in some tightly controlled environments. I'll concentrate on scripting so that the chapter is applicable to as many people as possible. Even if you use the cmdlets, understanding how to script the task will aid your understanding of the subject.

I don't fully recommend the `System.DirectoryServices.AccountManagement` .NET classes for use with Active Directory for three reasons. First, you need to have installed .NET 3.5, which not everyone can do. Second, the functionality has some gaps; for instance there's no capability to set the `description` attribute (this seems to be a common failing on the .NET classes for working with Active Directory). Finally, the syntax is odd compared to the standard ADSI syntax many people already know. I'll show examples using these classes because there's some useful functionality and because it's new with little documentation.

For local users and groups, the `System.DirectoryServices.AccountManagement` .NET classes are excellent and will be used in the following scripts. Variant scripts using `[ADSI]` will be shown for those users who don't have .NET 3.5 available. For Active Directory-based users, the `[ADSI]` accelerator will mainly be used, with the Microsoft or Quest AD cmdlets used as a variant.

> **NOTE** I won't be providing variant scripts in all the remaining chapters of the book, just where I think there's value in showing two approaches.

First up on the automation express is local users and groups.

5.2 Local users and groups

Enterprises use Active Directory to manage users and groups. But they still need to manage local user accounts. This could be because the machine isn't a domain member (for example if it's in a perimeter network).

> **NOTE** If performing this on Windows Vista or Windows Server 2008, PowerShell needs to be started with elevated privileges—it needs to be started using Run as Administrator. On Windows XP or Windows Server 2003, you must be logged on with an account with Administrator privileges.

As stated earlier, we'll be using the `System.DirectoryServices.AccountManagement` .NET classes in these examples. You must have .NET 3.5 loaded to use this namespace. If it's not possible to use this version of .NET then the scripts shown under the variation headings can be used.

COMPUTER NAMES In the example scripts dealing with local users and groups, the machine name is always `pcrs2`. You'll need to change this in your environment.

Compared to Active Directory, there are a limited number of tasks we'd want to perform against local users. The tasks condense to creation and modification activities against users and groups. We need to create users before we can modify them, so that's where we'll start.

TECHNIQUE 1 User creation

Creating user accounts is the first step in working with users. In this case, we're creating an account on the local machine. Ideally, we're looking for a method that'll work when run locally or against a remote machine. We can achieve this by using the following approach.

PROBLEM

We need to create a local user account on a Windows machine.

SOLUTION

Creating a user account is a common administrative activity and is illustrated in listing 5.1. If it's not possible to use this .NET class, use the variant presented in listing 5.2. Start by loading the `System.DirectoryServices.AccountManagement` assembly as shown in listing 5.1 (see ❶). PowerShell doesn't automatically load all .NET assemblies, so we need to perform that chore. If an assembly will be used often, put the `load` statement into your profile. Nothing bad happens if you do perform the `load` statement multiple times.

The `[void]` statement is new. All it does is suppress the messages as the assembly loads. If you want to see the messages, remove it. I've used the full name of the assembly (obtained via `Resolve-Assembly` in PowerShell Community Extensions), as some of the other load mechanisms are in the process of being removed. In PowerShell v2 we could use:

```
Add-Type -AssemblyName System.DirectoryServices.AccountManagement
```

as an alternative load mechanism. This avoids the need to use the deprecated .NET method.

Listing 5.1 Creating a local user account

```
[void][reflection.assembly]::Load(
"System.DirectoryServices.AccountManagement, Version=3.5.0.0,
     Culture=neutral, PublicKeyToken=b77a5c561934e089")

$password = Read-Host "Password" -AsSecureString
$cred = New-Object -TypeName System.Management.Automation.PSCredential
-ArgumentList "userid", $password

$ctype = [System.DirectoryServices.AccountManagement.ContextType]::Machine
$context = New-Object
```

❶ Load the assembly

❷ Create the password

❸ Set the context

```
-TypeName System.DirectoryServices.AccountManagement.PrincipalContext
-ArgumentList $ctype, "pcrs2"

$usr = New-Object -TypeName                                    <-- 4 Create user
System.DirectoryServices.AccountManagement.UserPrincipal
-ArgumentList $context

$usr.SamAccountName = "Newuser1"                               <-- 5 Set properties
$usr.SetPassword($cred.GetNetworkCredential().Password)
$usr.DisplayName = "New User"
$usr.Enabled = $true
$usr.ExpirePasswordNow()

$usr.Save()                                                    <-- 6 Save
```

The next job after loading the assembly is to generate a password for the new account ❷. The method presented here avoids having the password in the script (good security) and doesn't show its value on screen as it is input. Using `Read-Host` with the `-AsSecureString` option means we get prompted for the password, and when we type it, asterisks (`*`) are echoed back on screen rather than the actual characters. The string we've typed in is encrypted and can't be accessed directly:

```
PS> $password = Read-Host "Password" -AsSecureString
Password: ********
PS> $password
System.Security.SecureString
PS>
```

There's a slight issue with this technique. You can't use the secure string directly as a password in a user account. We resolve this by creating a PowerShell credential as the next step. `Userid` is a placeholder for the account name in the credential. Any string will do.

PowerShell needs to know where to create the account using the `PrincipalContext` class ❸. This takes a context type, in this case `Machine` (the local SAM store), and the name of the machine. If the name is null then the local machine is assumed.

MACHINE NAME The machine name will need to be changed for your environment.

The `UserPrincipal` class is used to create an empty user account in the data store we set in the context ❹. We can then start to set the properties of the user account ❺ as shown. `SamAccountName` and `DisplayName` should be self explanatory. `$usr.Enabled = $true` means that the account is enabled and ready to use; `$usr.ExpirePasswordNow()` indicates that the user must change the password at first logon.

TRUE OR FALSE The PowerShell automatic variables `$true` and `$false` are used to define Boolean values—true or false. They're of type `System.Boolean`. One thing to explicitly note is that `$true` isn't the same as "true" and `$false` isn't the same as "false." Remember to use the Booleans, not the strings.

Setting the password value is interesting, as it uses the `SetPassword` method with the password from the credential we created earlier:

```
$usr.SetPassword($cred.GetNetworkCredential().Password)
```

The last action is to write the new user account back to the local data store ❻ using the `Save()` method.

DISCUSSION

This may seem like a lot of code, especially when compared to the WinNT method presented next, but everything before we create the user object ❹ (in listing 5.1) could be created once and used many times. One property that can't be set using this approach is the description. If you want to use this, consider the WinNT approach presented in listing 5.2. We'll be using ADSI via the `[ADSI]` accelerator in this example. ADSI can connect to a number of account data stores, including Active Directory using the LDAP provider, which we'll see in later sections, and the WinNT provider for connecting to the local account database. If you've been in IT long enough to remember scripting against Windows NT, you'll remember using WinNT. No prizes for guessing where the name comes from.

> **WINNT AND ACTIVE DIRECTORY** The WinNT provider can be used to connect to LDAP directories such as Active Directory, but it has much reduced capability compared to the LDAP provider.

WinNT and LDAP are case sensitive. Remembering this will make debugging scripts much faster.

Listing 5.2 Creating a local user account using WinNT

```
$computer = "pcrs2"                          ❶ Computer name
$sam = [ADSI]"WinNT://$computer"             ❷ Link to SAM
$usr = $sam.Create("User", "Newuser2")       ❸ Create user
$usr.SetPassword("Passw0rd!")
$usr.SetInfo()

$usr.Fullname = "New User2"                  ❹ Set fullname
$usr.SetInfo()
$usr.Description = "New user from WinNT"     ❺ Set Description
$usr.SetInfo()
$usr.PasswordExpired = 1                     ❻ Force password change
$usr.setInfo()
```

Set a variable to the computer name (change for your environment) ❶. We then need to bind to the local Security Account Manager (SAM) database using the WinNT ADSI provider ❷. Setting the computer name in a variable isn't strictly necessary, but it makes things easier if you want to change the script to accept parameters.

The `Create()` method is used to create a user object ❸. The first parameter tells the system to create a user, and the second parameter is the account name. Unless you want the account to be disabled, you must set the password at this point.

PASSWORD WARNING I deliberately wrote this script with the password in the script to show how obvious it is. Imagine a scenario where you create a set of new accounts. If someone finds the password, you could have a security breach. As an alternative, you could leave the account disabled until required.

`SetInfo()` is used to save the account information back to the database. There are a few other attributes we want to set. The full name defaults to the account name (login ID), so changing it to the user's name will make finding the account easier ❹. Using the .NET method, we couldn't set the description, but it can be done quite easily with this method ❺. The last setting is to force the users to change their passwords when they log on for the first time ❻. If `PasswordExpired` is set to one, the password change is enforced. A value of 0 for `PasswordExpired` means that users don't have to change their passwords.

It's not strictly necessary to use `SetInfo()` after every change. The attribute changes could be rolled up by a single call to `SetInfo()`.

One way or another, we've created our user. Now we have to think about creating a group for the user account.

TECHNIQUE 2 Group creation

Working with groups is much more efficient than working with individual accounts. You need to give a set of users access to a resource. Put them in a group and assign the permissions to the group. Before we can do that, we need to create the group.

PROBLEM

We need to create a local group on a Windows computer.

SOLUTION

Continuing our exploration of `System.DirectoryServices.AccountManagement`, we use the `GroupPrincipal` class to create a group in listing 5.3. After loading the assembly ❶ (in listing 5.3) we set the context to the local machine ❷. This time we're creating a group, so we need to set the group scope to local ❸. This code can be modified to work at the domain level by changing the context to domain and the group scope to the appropriate value. Examples of using these .NET classes on Active Directory accounts will be given later.

Listing 5.3 Create a local group

```
[void][reflection.assembly]::Load(
"System.DirectoryServices.AccountManagement,
Version=3.5.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089")                ❶ Load assembly

$ctype =
[System.DirectoryServices.AccountManagement.ContextType]
::Machine

$context = New-Object
-TypeName System.DirectoryServices.
AccountManagement.PrincipalContext
```

```
-ArgumentList $ctype, "pcrs2"                    ← 2 Set context

$gtype = [System.DirectoryServices.AccountManagement.GroupScope]::Local
                                                          Set group scope 3
$grp = New-Object
-TypeName System.DirectoryServices.
AccountManagement.GroupPrincipal
-ArgumentList $context, "lclgrp01"                ← 4 Create group

$grp.IsSecurityGroup = $true                      5 Set properties
$grp.GroupScope = $gtype                          ↑
$grp.Save()                                       6 Save
```

The `GroupPrincipal` class is used to create the group using the context and a group name of `lclgrp01` ❹. Set the group scope (type of group) ❺, save the changes ❻, and we're done. We may not need to explicitly set the fact that it's a security group for local groups by using `$grp.IsSecurityGroup = $true`, but it's useful when working with Active Directory groups.

DISCUSSION

Using ADSI is just as easy, as shown in listing 5.4

Listing 5.4 Create a local group with WinNT

```
$computer = "pcrs2"                               1 Connect to machine
$sam = [ADSI]"WinNT://$computer"                  ↑
$grp = $sam.Create("Group", "lclgrp02")           ← 2 Create group
$grp.SetInfo()
                                                  3 Set properties
$grp.description = "New test group"               ↑
$grp.SetInfo()                                    4 Save
```

After connecting to the local SAM database ❶, we use the `Create()` method ❷. The parameters indicate that we're creating a group and the group name. This approach allows us to set the description ❸, and we save the new group ❹ to the database.

Note that we also did a `SetInfo()` immediately after creation. As we'll see when working with Active Directory, saving is needed so we can actually work with the object. Groups need members, so now we'll look at how to add members into a group.

TECHNIQUE 3 Group membership

Groups by themselves don't do anything. We need to add members to make them useful. We should remove members from the group when they don't need to be in there anymore. We all clean up group membership—don't we?

PROBLEM

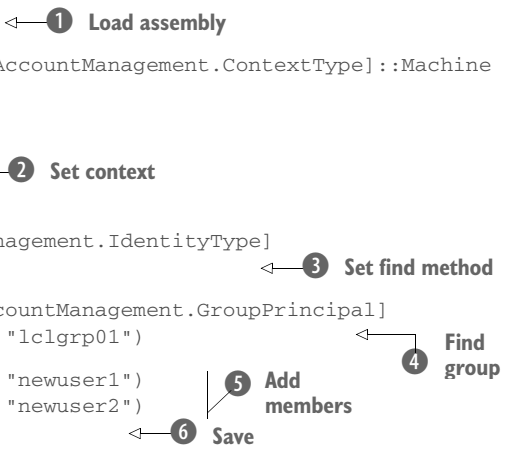
We need to add a new member to a local group.

SOLUTION

The `GroupPrincipal` class contains methods for modifying the membership of a group. By now, you should see a pattern emerging for how these scripts work. Listing 5.5 demonstrates this pattern. Load the assembly ❶, set the context to the local machine ❷, set the method to find the group ❸, and then find the `GroupPrincipal` ❹.

Listing 5.5 Modify local group membership

```
[void][reflection.assembly]::Load(
"System.DirectoryServices.AccountManagement,
Version=3.5.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089")

$ctype = [System.DirectoryServices.AccountManagement.ContextType]::Machine
$context = New-Object
-TypeName System.DirectoryServices.
AccountManagement.PrincipalContext
-ArgumentList $ctype, "pcrs2"

$idtype =
[System.DirectoryServices.AccountManagement.IdentityType]
::SamAccountName

$grp = [System.DirectoryServices.AccountManagement.GroupPrincipal]
::FindByIdentity($context, $idtype, "lclgrp01")

$grp.Members.Add($context, $idtype, "newuser1")
$grp.Members.Add($context, $idtype, "newuser2")
$grp.Save()

## remove group members
#$grp.Members.Remove($context, $idtype, "newuser1")
#$grp.Save()
```



The group has a collection of members and we can use the `Add()` method to modify the membership ❺. Note that we have to give the context and how we're identifying the user as well as the user account to add. A final `Save()` ❻ and our changes are written back to disk.

To remove group members, we use the `Remove()` method instead of the `Add()` method.

DISCUSSION

Using ADSI to modify group membership is equally straightforward, as seen in listing 5.6. We get objects to represent the users and group and use the group's `Add()` method to add members. Note that we have to give the path to the user, which will be something like `WinNT://pcrs2/newuser1`. We could input the path directly to `Add()`. Removing users is equally direct: we use the `Remove()` method as shown.

Listing 5.6 Modify local group membership with WinNT

```
$grp = [ADSI]"WinNT://pcrs2/lclgrp02"
$user = [ADSI]"WinNT://pcrs2/newuser1"
$grp.Add($user.Path)
$grp.SetInfo()

$user2 = [ADSI]"WinNT://pcrs2/newuser2"
$grp.Add($user2.Path)
$grp.SetInfo()

#$grp.Remove($user2.Path)
#$grp.SetInfo()
```

That's all we're going to look at as far as local users and groups are concerned. Automating local accounts gains us some efficiency improvements, but it's not the whole picture. Enterprises will be using Active Directory to manage the vast majority of their user accounts. With accounts numbering in the hundreds, if not thousands, it's in the automation of Active Directory management that we'll really see some benefit.

5.3 Active Directory users

Active Directory is the foundation of administration in a modern Windows environment. I've given an example of the mass creation of user accounts and the savings that automating that process brought. It's time to start looking at the automation of Active Directory user account management in detail. Though I can't cover all eventualities in a single chapter, the examples here will form a solid start to building automation into your environment.

The majority of the scripts deal with a single object, but in listing 5.11 I show how to create users in bulk. In listing 5.23 where we discuss changing group membership, there's a technique for dealing with all of the users in a particular OU. This technique can be used in the other scripts as appropriate to enable them for bulk processing. In this section, I'll use an ADSI-based script as the primary method, with the Quest and Microsoft AD cmdlets as secondary methods. If it's possible to use these cmdlets in your organization, I recommend you do so.

> **DOMAIN NAMES** In these scripts I'm working in my test domain. You need to change this for your environment, so you must change the LDAP connectivity strings of the form LDAP://OU=*England,dc=manticore,dc=org* to match your domain.

TECHNIQUE 4 User creation

Any work on user accounts must start with creating that user account

PROBLEM

A user account has to be created in Active Directory.

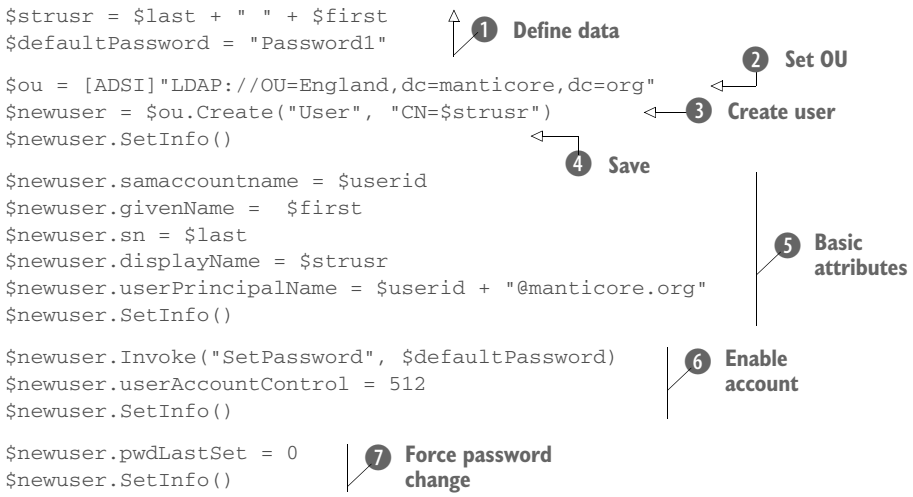
SOLUTION

Using ADSI, the solution has similarities to that presented in listing 5.2. We start by creating the data, such as the name and user ID that we'll use to create the account ❶ (in listing 5.7). I create the fullname (`$struser`) from the first and last names, as each will be required later. I've deliberately set the password in the script rather than explain how to use a secure string again. Use the technique shown in listing 5.1 if you want to mask the password. I've included a version of listing 5.7 in the code download file that uses the password-masking technique—look for listing 5.7s.

Listing 5.7 Creating a single user

```
$first = "Joshua"
$last = "TETLEY"
$userid = "jtetl"
```

❶ Define data



The next steps are to define the OU where we'll create the user ❷ then perform the creation ❸. The new user account should be immediately saved ❹. This ensures that later processing occurs without error.

The attributes concerned with the user's name are set ❺. Surname is `sn` and `givenName` is the first name. The display name is what's shown as the full name in Active Directory Users and Computers (ADUC). The attribute `cn` is the name shown in AD Users and Computers when the OU is browsed. `cn` is also used to identify the user when we're creating a directory entry for modification. See listing 5.11.

When first created, an Active Directory account is disabled by default. We need to set a password and set the `useraccountcontrol` attribute to 512 (normal user) to enable the account ❻. `useraccountcontrol` flags are detailed in appendix D. The final process is to set the `pwdLastset` attribute to zero ❼. This forces the user to change the password at next logon.

DISCUSSION

That was a fairly lengthy script for creating users. PowerShell cmdlets give a much better experience than scripting, as we'll see in listings 5.8 and 5.8a. There are three separate examples here to illustrate different methods of handling passwords:

- In the first example in listing 5.8 ❶, `$null` password is specified. No password is set and the account is disabled unless it's requested to be enabled. A password has to be supplied before the account can be enabled.
- In the second example ❷, no password is specified. No password is set and the account is left in a disabled state. Again, a password is required before the account can be enabled.
- In the final example ❸, a user password is specified. The password is set and the account is enabled via the `Enable-ADAccount` cmdlet.

Note that the Microsoft AD cmdlets all use a prefix on AD for the noun.

Listing 5.8 Creating a single user by Microsoft cmdlets

```
New-ADUser -Name "DARWIN Charles" -SamAccountName "CDarwin" `      (1)
-GivenName "Charles" -Surname "DARWIN" `
-Path 'ou=england,dc=manticore,dc=org' -DisplayName "DARWIN Charles" `
-AccountPassword $null -CannotChangePassword $false `
-ChangePasswordAtLogon $true  -UserPrincipalName "CDarwin@manticore.org"

New-ADUser -Name "NEWTON Isaac" -SamAccountName "INewton" `      (2)
-GivenName "Isaac" -Surname "NEWTON" `
-Path 'ou=england,dc=manticore,dc=org' -DisplayName "NEWTON Isaac" `
-AccountPassword (Read-Host -AsSecureString "AccountPassword") `
-CannotChangePassword $false  -ChangePasswordAtLogon $true `
-UserPrincipalName "INewton@manticore.org"

New-ADUser -Name "SORBY Henry" -SamAccountName "HSorby" `      (3)
-GivenName "Henry" -Surname "SORBY" `
-Path 'ou=england,dc=manticore,dc=org' -DisplayName "SORBY Henry"  `
-AccountPassword (Read-Host -AsSecureString "AccountPassword")`
-CannotChangePassword $false  -ChangePasswordAtLogon $true  `
-UserPrincipalName "HSorby@manticore.org"

 Enable-ADAccount -Identity HSorby        (3)
```

Using the Quest cmdlets is similar. Some of the parameters are slightly different—for example, `Path` and `ParentContainer` respectively for the OU in which the user is created.

Listing 5.8a Creating a single user with the Quest cmdlets

```
New-QADUser -Name "SMITH Samuel" -FirstName "Samuel" -LastName "SMITH"
-DisplayName "SMITH Samuel" -SamAccountName ssmith
-UserPassword "Password1" -UserPrincipalName "ssmith@manticore.org"
-ParentContainer "ou=England,dc=manticore,dc=org"

Set-QADUser -Identity "manticore\ssmith"
-ObjectAttributes @{useraccountcontrol=512; pwdLastSet=0}
```

The `New-QADUser` cmdlet is used to create user accounts. *New* is used for cmdlets that create objects. Comparing listing 5.8 with listing 5.7: the similarities are obvious. The `-Name` parameter corresponds to `cn` used to create the user in listing 5.7. Note we need to create the user, then set the `useraccountcontrol` and force the password change by using `Set-QADUser`. These attributes can't be set when creating the account. It just doesn't work.

Creating a single user may be slightly more efficient in PowerShell, especially using the cmdlets. We really gain from automating the bulk creation of user accounts.

TECHNIQUE 5 User creation (bulk)

We've seen how to create users one by one. The full benefit of automation is achieved by creating users in bulk. In order to get the most from these techniques, you may want to change your procedures for new joiners to the organization. Get all the new user information in one go and create them using listing 5.9 or 5.10. One or two runs

per week and they're all done. It's much more efficient than single-user creation in dribs and drabs.

PROBLEM

We need to create a lot of users at one time.

SOLUTION

Our solution is an adaptation of listing 5.7. Take a moment to compare listing 5.9 with listing 5.7 and you'll see that the content of the `foreach` loop is a modified version of listing 5.7.

> **FOREACH ALIAS** Foreach as used here is an alias for `Foreach-Object` as shown in listing 5.10a.

We start by reading a CSV file called pms.csv and passing the contents into a `Foreach_object` (in listing 5.9) ❶. The CSV file contains three columns with headers of last, first, and userid. The great advantage of using a CSV file is that we can refer to the column headers in the rest of our script as properties of the pipeline object.

Listing 5.9 Creating users in bulk

```
Import-csv pms.csv | foreach {
    $strusr = $_.Last.ToUpper() + " " + $_.First

    $ou = [ADSI]"LDAP://OU=England,dc=manticore,dc=org"
    $newuser = $ou.Create("user","cn=$strusr")
    $newuser.SetInfo()

    $newuser.samaccountname = $_.userid
    $newuser.givenName =  $_.first
    $newuser.sn = $_.last
    $newuser.displayName = $strusr
    $newuser.userPrincipalName = $_.userid + "@manticore.org"
    $newuser.SetInfo()

    $newuser.Invoke("SetPassword", "Password1")
    $newuser.userAccountControl = 512
    $newuser.SetInfo()

    $newuser.pwdLastSet = 0
    $newuser.SetInfo()

    Write-Host "Created Account for: "  $newuser.Displayname
}
```

❶ Read CSV file into loop
❷ Create user
❸ Set OU
❹ Save user
❺ Completion message

We can create the contents of the `$struser` variable by using `$_.last.ToUpper()` and `$_.first` (remember PowerShell isn't case sensitive) ❷. `ToUpper()` is a string-handling method that converts all characters to uppercase. `$_` refers to the object coming down the pipeline. In this case, the object is a line from the CSV file. The column headers are properties, so `$_.first` means the contents of the column named `first` in the current row. This is real processing power.

I've hard coded an OU for this batch of users ❸. In a typical organization, you may be creating users in a number of OUs, so this could become another column in the

CSV file. After we create the user ❹, we proceed to complete the attributes as before. The only difference is that we're reading them from the pipeline object rather than coding them into the script. The last line of the script ❺ writes out a message to state that creation is complete.

By adding a couple of commands and changing the way we get the data, we've turned a script to create a single user into one that can create many. The time and effort to go from listing 5.7 to 5.9 is minimal. The administrative effort that will be saved is huge and will easily pay back the investment.

DISCUSSION

In listing 5.8, we had a script to create a single user with the AD cmdlets. This is can also be turned into a bulk creation script, as in listing 5.10.

Listing 5.10 Creating users in bulk with Microsoft cmdlets

```
Import-Csv -Path users2.csv | foreach {
New-ADUser -Name "$($_.Given) $($_.Surname)" `
-SamAccountName $_.Id -GivenName $_.Given `
-Surname $_.Surname `-Path 'ou=england,dc=manticore,dc=org' `

 -DisplayName "$($_.Given) $($_.Surname)" `
 -AccountPassword $null -CannotChangePassword $false `
 -ChangePasswordAtLogon $true  `
 -UserPrincipalName "$($_.Id)@manticore.org"
 }
```

In this case, I've used the parameters as subexpressions for variety. It's not usually necessary to do this, but is worth demonstrating. We have seen how to create a user account with the Microsoft cmdlets. Listing 5.10a shows how we can perform the same action with the Quest cmdlets.

Listing 5.10a Creating users in bulk with Quest cmdlets

```
Import-Csv pres.csv | ForEach-Object {
$name = $_.last.ToUpper() + " " + $_.first
$upn = $_.userid + "@manticore.org"
New-QADUser -Name $name -FirstName $_.first -LastName $_.last.ToUpper()
 -DisplayName $name -SamAccountName $_.userid -UserPassword "Password1"
 -UserPrincipalName $upn  -ParentContainer "ou=USA,dc=manticore,dc=org"

Set-QADUser -Identity $upn
-ObjectAttributes @{useraccountcontrol=512; pwdLastSet=0}
}
```

These examples follow the same format as listing 5.9. We take a CSV file and pass it into a `foreach` loop. The data for the parameters is read from the pipeline as before. After looking at listing 5.10, there are no real differences in this one as to how we handle the data apart from the fact that I create a variable for the UPN. This is so that it can be used in both cmdlets. It's more efficient to only create it once. I haven't specifically written out a message, because the two cmdlets automatically create messages.

The names in the CSV files are those of English scientists, British prime ministers, and US presidents respectively, in case you were wondering. Unfortunately, things never remain the same in IT, so we have to tear ourselves away from PowerShell Space Invaders and modify some users. An admin's work is never done.

TECHNIQUE 6 User modification

After creating a user account, it's more than probable that we'll need to make modifications. People move departments; telephone numbers change; even names can change. We may want to increase security by restricting most users to being able to log on only during business hours.

Active Directory can hold a lot of information about your organization. If you keep the information up to date and accessible then you can leverage the investment in Active Directory and you don't need a separate phone book system, for instance.

PROBLEM

We have to make modifications to one or more user accounts in Active Directory.

SOLUTION

Using ADSI, we retrieve a directory entry for the user account we need to modify and set the appropriate properties. This is one of the longest scripts we'll see, but as we break it down, you'll see that it's not as bad as it looks. I've organized the script to match the tabs on the user properties in ADUC.

> **SCRIPT USAGE** I don't expect this script to be used in its entirety. In normal use, I'd expect a few attributes to be changed rather than a bulk change like this. It's more efficient to present all the changes in one script. Then you can choose which attributes you need to modify.

In listing 5.11, we start by getting a directory entry for the user ❶. This is the part that will change in your organization. If you're making the same change to lots of users, put them into a CSV file and use a `foreach` loop in a similar manner to listing 5.9.

Listing 5.11 Modifying user attributes

```
$user = [ADSI]
"LDAP://CN=CHURCHILL Winston,OU=England,DC=Manticore,DC=org"
$user.Initials = "S"
$user.Description = "British PM"
$user.physicalDeliveryOfficeName = "10 Downing Street"
$user.TelephoneNumber = "01207101010"
$user.mail = "wsc@manticore.org"
$user.wwwHomePage = "http://www.number10.com"
$user.SetInfo()

$user.streetAddress = "10 Downing Street"
$user.postOfficeBox = "P.O. 10"
$user.l = "London"
$user.St = "England"
```

❶ Get user
❷ Start of General tab
❸ Office
❹ Email
❺ Start of Address
❻ PO Box
❼ City
❸ State/province

```
$user.postalCode = "L10 9WS"
$user.c = "GB"
$user.SetInfo()

$comp = "comp1,comp2"
[byte[]]$hours = @(0,0,0,0,255,3,0,255,3,0,255,3,0,255,3,0,255,3,0,0,0)

$user.logonhours.value = $hours
$user.userWorkstations = $comp
$user.SetInfo()

$user.profilepath = \\server1\usrprofiles\wsc
$user.scriptPath = "mylogon.vbs"
$user.homeDrive = "S:"
$user.homeDirectory = "\\server2\home\wsc"
$user.SetInfo()

$user.homePhone = "01207101010"
$user.Pager = "01207101011"
$user.Mobile = "01207101012"
$user.facsimileTelephoneNumber = "01207101014"
$user.ipPhone = "01207101015"
$user.Info = "This is made up data"
$user.SetInfo()

$user.Title = "Prime Minister"
$user.Department = "Government" "
$user.Company = "Britain""
$user.Manager = "CN=WELLESLEY Arthur,OU=England,DC=Manticore,DC=org" "

$user.SetInfo()
```


The first tab that we need to deal with is the General tab ❷. This holds the name information, which can be modified as shown. Usually the attributes we use in ADSI match those shown in ADUC. I've annotated those that are different such as office ❸ and email address ❹. I've used `SetInfo()` after each tab's worth of changes to ensure that they're written back. If you cut and paste the script, it's less likely the `SetInfo()` will be forgotten.

Moving on to the Address tab ❺, we find simple data such as the PO Box ❻ as well as number of catches. The City field on ADUC we have to treat as `l` (for location) ❼, and state\province becomes `st` ❽. Setting the country requires the use of the two-character ISO code in the `c` attribute ❾. In this case, `GB` is the ISO code for the United Kingdom, even though Great Britain is only part of the UK!

> **TIP** If you can't remember the ISO code for a particular country or aren't sure what to use, use ADUC to set the country by name on one user and ADSIEdit to check what code has been entered. With Windows Server 2008 ADUC, use the Attribute tab to view the data.

On the Account tab ⓬, we can also set the workstations a user can log on to ⓭ as well as the hours of the day he can log on. We need to create an array of workstation names ❿ and use this to set the attribute. The logon hours attribute is more complicated, in that we have to create an array of bytes as shown ⓫. Three bytes represent a day (start-

ing at Sunday) and each bit represents a one-hour time span. All zeros means the user isn't allowed to log on, and if all values are set to 255 (default) the user can log on 24x7. In the case shown, the user is restricted to logon times of Monday to Friday 8 a.m. to 6 p.m. If you want to use this, I recommend setting up one user in ADUC and copying the resultant values. This is definitely the quickest way to get it right.

The Profile tab ⓮ is for setting logon scripts and home drives as shown. The only difficulty here is the attribute names, as I've annotated, especially the scriptpath ⓯ which supplies the logon script to be run for the user. The local path ⓰ refers to the drive to be mapped to a user's home area and the connect attribute ⓱ supplies the UNC path to the user's home area. When you're setting telephone numbers on the Telephones tab ⓲, remember that the numbers are input as strings rather than numbers.

The final tab I'll deal with is the Organization tab ⓳. The attribute names match the ADUC fields as shown. Note that the Manger entry must be given the AD distinguished name as its input. The Direct Reports field is automatically backfilled from the Manager settings on other users. You can't set it directly.

DISCUSSION

I haven't given a full alternative using the cmdlets in this section. We can use the Microsoft cmdlets like this:

```
Get-ADUser -Identity hsorby | Set-ADUser -Department Geology
Get-ADUser -Identity hsorby -Properties Department

Get-ADUser -Identity hsorby -Properties *
```

The most efficient way to perform bulk changes is to use `Get-ADuser` to return the users in which we're interested and then pipe them into `Set-AdUser`. This way we can easily test which users are affected. The change can be examined with `Get-ADUser`. When we use `Get-ADUser`, we normally only get a small subset of properties returned. We can generate more data by explicitly stating which properties we want returned.

With the Quest cmdlets, we'd use the `Set-QADUser` cmdlet and use either one of the predefined parameters or the `-ObjectAttributes` parameter as shown in listing 5.10a.

TECHNIQUE 7 Finding users

We've seen how to create and modify user accounts in Active Directory. One of the other tasks we need to perform frequently is searching for particular users. No, not under the desk, but in Active Directory. In this section, we'll look at searching for an individual user, disabled accounts, and accounts that are locked out. You'll see other searches that look at logon times and account expiration later in the chapter.

Searching Active Directory requires the use of LDAP filters. They're explained in appendix D.

> **DELETED USER ACCOUNTS** Searching for deleted user accounts will be covered in chapter 10

We'll start with searching for a single user.

PROBLEM

We need to search Active Directory for specific users or accounts that are disabled or locked out.

SOLUTION

We can use the `System.DirectoryServices.DirectorySearcher` class to perform our search. In PowerShell v2, this can be shortened slightly by using `[ADSISEARCHER]`. Using `System.DirectoryServices.DirectorySearcher` makes searching faster and simpler compared to previous scripting options. We need to start by creating a variable with the name of the user to search for ❶ (in listing 5.12). We can search on other attributes, as we'll see later. We want to search the whole Active Directory, because we can't remember where we put this user. We can use `GetDomain()` to determine the current domain ❷. Using this method makes our script portable across domains. We then get a directory entry ❸ for the domain.

Listing 5.12 Searching for a user account

```
$struser = "BOSCH Herbert"      ❶ Set user

$dom          =
     [System.DirectoryServices.ActiveDirectory.Domain]::GetCurrentDomain()  ❷
$root = $dom.GetDirectoryEntry()

$search = [System.DirectoryServices.DirectorySearcher]$root      ❹
$search.Filter = "(cn=$struser)"                                 ❺ Set filter
$result = $search.FindOne()                                      ❻ Run search
                                                                 ❸ Root entry
if ($result -ne $null)      ❼
{
  $result.properties.distinguishedname
}
else {Write-Host $struser " Does not exist"}
```

Creating a search as shown ❹ will set the domain as the root of the search—we search the whole domain. We're looking for a particular user, so we need to set an LDAP filter for that user ❺. The `cn` attribute holds the name of the user account in Active Directory. It's possible to search on most attributes.

> **PAGE SIZE AND TIMEOUT** There's a limit on the number of results that will be returned from an LDAP search. The default limit is 1,000. If your results will exceed this number, add the line `$search.PageSize = 1000` after the filter. This will cause the results to be returned in batches (pages) of 1,000. When using the cmdlets, use the `PageSize` and `SizeLimit` parameters to control the return of data.
>
> There's a timeout of 120 seconds on the server side, at which point the server will return only the results found up to that point. The default client-side timeout is infinite.

When we run this search, we only expect a single result, so we use `FindOne()` ❻. As we'll see later, if we expect multiple results to be returned, we use `FindAll()`. Interestingly, `FindOne()` does a `FindAll()` and returns only the first result. If you've per-

formed Active Directory searches using VBScript in the past, note that we don't need to use an ADO recordset.

We perform a final check to see if we actually have a result ❼ and then we can display the distinguished name of the user. This will tell us where the user is hiding.

DISCUSSION

Using the cmdlets is even simpler. The Microsoft cmdlets give us:

```
Get-ADUser -Identity hsorby
```

And the Quest cmdlets produce:

```
$struser = "BOSCH Herbert"
Get-QADUser -ldapFilter "(cn=$struser)"
```

We could make this one line by putting the name into the `-Identity` parameter. The cmdlet automatically produces output, including the distinguished name, which minimizes the amount of code we need.

Our search script can be easily modified so that we can search for different things. Two examples are searching for disabled accounts and locked-out accounts, as shown in listing 5.13.

Listing 5.13 Disabled user accounts

```
$dom = [System.DirectoryServices.ActiveDirectory.Domain]
::GetCurrentDomain()
$root = $dom.GetDirectoryEntry()

$search = [System.DirectoryServices.DirectorySearcher]$root

$search.Filter =  "(&(objectclass=user)(objectcategory=user)
(useraccountcontrol:1.2.840.113556.1.4.803:=2))"
$result = $search.FindAll()

foreach ($user in $result)
{
  $user.properties.distinguishedname
}
```

❶ Search filter

❷ Find all disabled accounts

We create the search so that we're searching the whole domain again. The main difference in this script is the search filter ❶. Our LDAP filter will find user accounts. We need the `objectclass` and the `objectcategory`, as computer accounts also use the user class! The last part of the filter is where we look at the `useraccountcontrol` attribute and perform a bitwise `AND` on it with the value 2 (account disabled). The syntax looks bad, but just think of it as a long-winded way of saying "bitwise". The only part we need to think about changing is the final value, which is what we're searching for. The possible values for `useraccountcontrol` are listed in appendix D.

In case there's more than one disabled account, we use `FindAll()` to return multiple results ❷, which we can then display.

I'm almost embarrassed to present the cmdlet equivalents as they are so short. We'll start with the Microsoft cmdlet:

```
Search-ADAccount -AccountDisabled -UsersOnly |
select Name, distinguishedName
```

The Quest version is even shorter:

```
Get-QADUser -Disabled
```

It doesn't get any easier than that! The cmdlet also displays the results. What more can you ask for? Well, it doesn't make the tea for one...

Moving on, users and passwords don't mix. Users seem to take great delight in forgetting passwords and locking themselves out of Active Directory, usually on a Monday morning when they've just got back from vacation. Eventually, they may get around to ringing the help desk and you can check to see if they're locked out. Alternatively, you can use listing 5.14 to find the locked-out accounts.

Listing 5.14 Locked user accounts

```
Add-Type -AssemblyName System.DirectoryServices.AccountManagement      ❶

$ctype =
[System.DirectoryServices.AccountManagement.ContextType]::Domain
$context = New-Object -TypeName
     System.DirectoryServices.AccountManagement.PrincipalContext
-ArgumentList $ctype, "manticore.org", "DC=Manticore,DC=org"      ❷

$date = (Get-Date).AddDays(-1)      ❸

$mtype =
[System.DirectoryServices.AccountManagement.MatchType]
::GreaterThan                                                      ❹

$results =
[System.DirectoryServices.AccountManagement.UserPrincipal]
::FindByLockoutTime($context, $date, $mtype)                       ❺
if($results -ne $null){
  foreach ($result in $results){$result.distinguishedname}
}
else{Write-Host "No users locked out"}
```

`System.DirectoryServices.AccountManagement` from .NET 3.5 has a nice method, `FindByLockoutTime()`, which we can use to find locked accounts. In addition, we can see how to use these classes in a domain environment. As usual, we start by loading the .NET assembly ❶. In this case, I've used `Add-Type` from PowerShell v2. In PowerShell v1 you can use the `load` command from listing 5.1. The context in this case is a domain rather than a single machine. `ContextType` is set to `Domain` as shown, and the `PrincipalContext` is set to the name of the domain ❷. The arguments are the context type we created in ❶; the name of the domain and container we're working with, respectively. The container defined by the LDAP distinguished name of the domain.

The lockout time on the user accounts will be compared to a value we create ❸. We use `Get-Date` to retrieve the current date and use the `AddDays()` method to set the date back, in this case by one day. We're adding a negative number. There isn't a method to subtract days, so we fall back on this slightly inelegant approach. We'll be

searching for accounts locked out in the last 24 hours. By varying this value, we can control how far back we look for locked-out accounts.

The comparison operator for our search is provided by the `MatchType` ❹. In this case we're looking for values greater than the reference value—lockouts that have occurred since the reference time. The search is performed by the `FindByLockoutTime()` method with the context, reference date, and operator as parameters ❺. The usual check on the results and displaying the distinguished names completes the script. This is the easiest method to script for searching for locked-out accounts that I've found.

If you want a super easy way of finding locked-out accounts, it doesn't get much easier than using the AD cmdlets. The Microsoft cmdlet syntax is:

```
Search-ADAccount -LockedOut
```

and the syntax for the Quest cmdlet is very similar:

```
Get-QADUser -Locked
```

These will retrieve all locked-out accounts in the domain.

We've looked at searching for disabled accounts; we should now look at how to enable or disable them.

TECHNIQUE 8 Enabling and disabling accounts

Listing 5.4 showed how to disable or enable a local user account. This script shows how to perform the same action on an Active Directory account.

PROBLEM

We need to disable or enable an Active Directory account.

SOLUTION

An Active Directory user account can be disabled by modifying the `useraccountcontrol` attribute, as shown in listing 5.15. This is the domain equivalent of listing 5.1 in that it toggles between enabled/disabled—it'll enable a disabled account and vice versa. We use ADSI to connect to the relevant account, retrieve the `useraccountcontrol` attribute, perform a bitwise exclusive `OR` on it, and write it back. The bitwise exclusive `OR` will toggle the disabled bit to the opposite value; that is it will disable the account if enabled and enable if disabled.

Listing 5.15 Disabling Active Directory user accounts

```
$user = [ADSI]"LDAP://CN=BOSCH Herbert,OU=Austria,DC=Manticore,DC=org"
$oldflag = $user.useraccountcontrol.value
$newflag = $oldflag -bxor 2
$user.useraccountcontrol = $newflag
$user.SetInfo()
```

DISCUSSION

The AD cmdlets provide specific commands to disable and enable user accounts:

```
Disable-ADAccount -Identity HSorby
Enable-ADAccount -Identity HSorby
```

```
Disable-QADUser -Identity "CN=BOSCH Herbert,OU=Austria,DC=Manticore,DC=org"
Enable-QADUser -Identity "CN=BOSCH Herbert,OU=Austria,DC=Manticore,DC=org"
```

All we need is to pass the identity of the user to the cmdlet and it does the rest. I can type this faster than opening the GUI tools, especially if I know the user ID so I can use `domain\userid` as the identity with the Quest cmdlets. (See appendix D for an explanation of the differences between the two sets of cmdlets when handling identities.)

One problem that you may find is disabling an account and moving it to a holding OU pending deletion. We've seen how to disable it, and we'll now turn to the move.

TECHNIQUE 9 Moving accounts

One method of organizing users in Active Directory is to have OUs based on department or location. This can enable us to apply specific group policies to those users. If the users move to a different location or department, we need to move the account to the correct OU so they receive the correct settings. When people leave the organization, their user accounts should be deleted. Many organizations will have an OU specifically for accounts that are to be deleted, so the accounts have to be moved into the correct OU.

PROBLEM

A user account has to be moved to another OU.

SOLUTION

The `[ADSI]` accelerator gives us access to a `MoveTo` method, but we have to remember that it's on the base object, so we need to include .psbase in PowerShell v1. In v2, this isn't an issue, as it has been made visible. Listing 5.16 demonstrates how we use the `MoveTo()` method to move a user account into a new OU.

Listing 5.16 Moving Active Directory user accounts

```
$newou = [ADSI]"LDAP://OU=ToBeDeleted,DC=Manticore,DC=org"
$user = [ADSI]"LDAP://CN=SMITH Samuel,OU=England,DC=Manticore,DC=org"

$user.psbase.MoveTo($newou)
```

Using the `[ADSI]` type accelerator, we set variables to the user and target OU. If you were to perform `$user | get-member`, you wouldn't see any methods on the object apart from two conversion methods. But by using `$user.psbase | get-member`, we drop into the underlying object as discussed in chapter 2. There we can see a `MoveTo()` method that will do just what we want. We call the method with the target OU as a parameter and the user is whisked off to his new home. If we have to move a number of accounts from an OU, we can modify the script to read the OU contents and then perform a move on the selected accounts.

> **WITHIN A DOMAIN ONLY** The techniques in this section only work within a domain; they can't be used for cross-domain moves.

DISCUSSION

The AD cmdlets don't provide a cmdlet to explicitly move users between OUs, but we can use the generic cmdlets for moving AD objects. All we need to provide is the

identity of the user and target OU. Using the Microsoft cmdlet we can perform a move like this:

```
Move-ADObject
-Identity "CN=HUXLEY Thomas,ou=starking,dc=manticore,dc=org"
-TargetPath "ou=england,dc=manticore,dc=org"
```

The Quest cmdlet is similar, but notice the parameter is called `NewParentContainer` rather than `TargetPath`. There are just enough differences like this to get confusing if you use both sets of cmdlets on a regular basis:

```
Move-QADObject
-Identity "CN=SMITH Samuel,OU=England,DC=Manticore,DC=org"
-NewParentContainer "OU=ToBeDeleted,DC=Manticore,DC=org"
```

These cmdlets also work with groups and computer accounts. When we're not creating, moving, or modifying user accounts, someone is bound to ask for information such as the last time Richard logged on to the domain.

TECHNIQUE 10 Last logon time

Finding the last logon time for a user isn't straightforward. When Active Directory was introduced with Windows 2000, an attribute called `lastlogon` was made available. This is stored on a domain controller by domain controller basis. Each domain controller stores the date and time it last authenticated that user. The attribute isn't replicated.

Windows 2003 introduced another attribute called `lastlogontimestamp`. It does replicate between domain controllers, but it's only updated if the user hasn't logged on to that domain controller for more than a week. The value can easily become more than a week out of date. This attribute is really of use for determining if a user hasn't logged on for a significant period, for example finding all of the users who haven't logged on for a month or more.

PROBLEM

Determine the last time a user logged on to the domain.

SOLUTION

As discussed, in listing 5.17 we'll use the `lastlogon` and `lastlogontimestamp` attributes to find when a user last logged on to the domain. By using `System.DirectoryServices.ActiveDirectory.Domain` we can retrieve information about the current domain ❶. This includes a list of the domain controllers ❷ in the domain. By looping through this list, we can check each domain controller in turn for the last logon information. This wouldn't be practical in a domain with many domain controllers, so the list of domain controllers to check could be manually created.

Listing 5.17 Last logon times

```
$dom = [System.DirectoryServices.ActiveDirectory.Domain]
::GetCurrentDomain()                                          ❶

foreach ($dc in $dom.domaincontrollers) {      ❷ Iterate through domain controllers

$ldapstr = "LDAP://" + $dc.Name + "/cn=richard,cn=users,dc=manticore,dc=org"   ❸
```

```
$user = [ADSI]$ldapstr                    ← 4 Get user
"`nDomain Controller:   $($dc.Name)"                    5
"Name:                  {0}" -f  $($user.name)                  6

$ll = $user.lastlogon.value
$log = [datetime]$user.ConvertLargeIntegerToInt64($ll)
$lastlog = $log.AddYears(1600)

"Last Logon:            {0:F}" -f  $($lastlog)     ← 7 Last logon

$ll = $user.lastlogontimestamp.value
$log = [datetime]$user.ConvertLargeIntegerToInt64($ll)
$lastlog = $log.AddYears(1600)                                  8 Last logon
                                                                  timestamp
"Last Logon Timestamp: {0:F}" -f  $($lastlog)          ←

}
```

The LDAP string we use to connect is slightly modified to include the fully qualified domain name of the domain controller ❸. Note the use of the + symbol for string concatenation. Previously we've performed a serverless binding and not worried about which domain controller we connected to. Using the LDAP string, we connect to the designated domain controller and access the user account ❹ stored on that machine.

We can now print the required information starting with the domain controller ❺ name. We're substituting into the string, but need to use the `$()` to ensure the name is evaluated before substitution; otherwise the name of the object would be output! The `` `n `` before the domain controller is a special character that forces a new line. Special characters are detailed in appendix A.

The `name` ❻, `lastlogon` ❼, and `lastlogontimestamp` ❽ are displayed using the string formatting operator `-f`. The fields within the string are enclosed in `{}` and substituted by the variables to the right of the `-f` operator in turn. The two logon times are stored in ticks (10,000th of a second, counting from January 1, 1600). We need to convert the number that's stored in Active Directory into a 64-bit integer and then into a date.

When we use `$log = [datetime]$user.ConvertLargeIntegerToInt64($ll)` to create the date it starts counting from 0 AD so the date is 1,600 years too low. We need to add 1,600 years to the resultant date to make it match the calendar.

In listing 5.18 we use the `FromFileTime()` method of the datetime class which automatically performs this addition. A simple example illustrates how it works.

```
PS> $d = Get-Date
PS> $d

25 March 2010 21:36:47

PS> $d.Ticks
634051498076096000
PS> [datetime]::FromFileTime($d.Ticks)

25 March 3610 21:36:47
```

We get the date and save it to a variable. The date and number of ticks can be viewed. When we convert the number of ticks back to a date the 1600 years is automatically added.

DISCUSSION

Using the cmdlets is a little simpler, but we still need to query multiple domain controllers, as shown in listing 5.18.

Listing 5.18 Last logon times using Microsoft cmdlets

```
Get-ADDomainController -Filter *| foreach {
    $server = $_.Name
    $user = Get-ADUser -Identity Richard `
      -Properties lastlogon, lastlogondate, lastlogontimestamp `
      -Server $($server)

   $t1 = [Int64]::Parse($($user.lastLogon))
   $d1 = [DateTime]::FromFileTime($t1)

   $t2 = [Int64]::Parse($($user.lastLogontimestamp))
   $d2 = [DateTime]::FromFileTime(t2)

   Add-Member -InputObject $($user) -MemberType Noteproperty `
    -Name "DCName" -Value $($server) -PassThru -Force |
    Format-Table DCName, `
    @{Name="LastLogonTime"; Expression={$($d1)}},`
    lastlogondate, `
    @{Name="LastLogonTimeStamp"; Expression={$d2}
}}
```

`Get-ADDomainController` will only return a single domain controller by choice. This can be overridden by specifying `*` in the filter parameter. Each domain controller is queried for the last logon time information. Note that `lastlogondate` is new in Windows Server 2008 R2. I'm using `Add-Member` to add the domain controller name as a new property on the user object. This enables us to see to which domain controller the information relates. Note how we have to work to retrieve the date from the Int64 that's held in Active Directory.

The Quest solution is similar to listing 5.18 in that we connect to the domain ❶ (in listing 5.19) and loop through the domain controllers ❷ as before. We print the domain controller name ❸ and then connect to the domain controller of interest ❹. `$null` is used to suppress the informational messages regarding the connection. The user information is retrieved and displayed ❺. The date creation is handled automatically ❻. We then disconnect from the domain controller.

Listing 5.19 Last logon times using Quest cmdlets

```
$dom = [System.DirectoryServices.ActiveDirectory.Domain]
::GetCurrentDomain()
foreach ($dc in $dom.domaincontrollers) {

"`nDomain Controller:   $($dc.Name)"

$null = Connect-QADService -Service $dc.Name

Get-QADUser -Identity 'manticore\Richard' |
Select-Object name, lastlogon, lastlogontimestamp | Format-List

Disconnect-QADService
}
```

❶ ❷ Iterate through domain controllers

❸

❹

❺ Get user

❻ Disconnect

These solutions aren't satisfactory because we have to query a number of domain controllers to get an exact time. But if we only need an approximate last logon time, using the `lastlogontimestamp` is a simpler option.

In addition to knowing when users last logged on, we may need to know when their passwords or, in the case of temporary staff, their accounts are going to expire.

TECHNIQUE 11 Password expiration

The default maximum password age is 42 days and is controlled by domain-level group policy. This is often altered to meet an organization's particular needs. Users will often forget that passwords need changing, especially mobile users who're rarely in the office. It can often save administrative effort to remind them that their passwords will need changing ahead of time. It's usually possible to change a password when connected by VPN, but not if the password has already expired. Prompting users to change passwords ahead of time can solve the problem before it arrives.

PROBLEM

We need to find the users whose passwords will expire within a given time frame.

SOLUTION

This involves searching the domain, so we return to our search script and modify the LDAP filter to check the `pwdlastset` attribute. The expiration date for the password isn't stored directly. The date the password was last set is stored in the `pwdlastset` attribute. Unfortunately, this isn't directly accessible because it's a COM large integer, like the logon times we saw in the previous example. We need to convert some dates into the correct format and use them in our search filter, as in listing 5.20.

Listing 5.20 Password expiration check

```
$now = (Get-Date).ToFileTime()                                  # 1 Set current date
$end = ((Get-Date).Adddays(-42)).ToFileTime()                   # 2 Set time period of interest

$dom = [System.DirectoryServices.ActiveDirectory.Domain]
::GetCurrentDomain()

$root = $dom.GetDirectoryEntry()                                # 3 Get current domain

$filt = "(&(objectcategory=Person)" +
"(objectclass=user)" +
"(pwdlastset>=$end)(pwdlastset<=$now))"

$search = [System.DirectoryServices.DirectorySearcher]$root     # 4 Create searcher
$search.Filter = $filt
$results = $search.FindAll()                                    # 5 Set filter

foreach ($result in $results){                                  # 6 Display results
    $result.properties.distinguishedname
}
```

Start by using the current date (`Get-Date`) and convert it into the correct format using the `ToFileTime()` method ❶. If we assume that we have a 42-day maximum password age then all passwords should've been reset at least 42 days ago. We need to decide how many days ago we want to check for password reset. If you're looking at passwords that

will expire in the next 10 days, we're interested in those set 32 days ago, and so forth. As I'm using a test domain, I had to force some of this, so my example shows a date of 42 days in the past—in other words, all password changes ❷. You'll need to set this value depending on your password policy and how far ahead you want look.

We get the current domain root ❸ and create a directory searcher ❹, as we've seen previously. The filter ❺ is interesting in that we need the `objectcategory` and `objectclass` to restrict the search to users. Leave off the `objectcategory` and you'll get computer accounts as well.

> **COMPUTER PASSWORDS** Computer passwords set themselves—don't try to change them manually.

We check the `pwdlastset` attribute for accounts that fall between our chosen dates using `FindAll()` and display the results ❻. We're using a `DirectorySearcher` object so you don't have access to the full property list. We can use the distinguished name to access a `DirectoryEntry` object and list full names, and so on. We could even send the user an email (PowerShell v2 has a `Send-MailMessage` cmdlet or we can script it).

DISCUSSION

A similar result can be achieved using the cmdlets:

```
$now = (Get-Date).ToFileTime()
$end = ((Get-Date).AddDays(-42)).ToFileTime()

$filt = "(&(objectcategory=Person)" +
"(objectclass=user)(pwdlastset>=$end)" +
"(pwdlastset<=$now))"

Get-ADUser -LDAPFilter $filt

$now = (Get-Date).ToFileTime()
$end = ((Get-Date).Adddays(-42)).ToFileTime()

$filt = "(&(objectcategory=Person)" +
"(objectclass=user)(pwdlastset>=$end)" +
"(pwdlastset<=$now))"

Get-QADUser -ldapFilter $filt
```

We set the start and end dates of our search and use the same LDAP filter as earlier. We get the same result, but with less code.

Temporary workers are often given accounts with an expiration date. Searching for these is similar to searching for expiring passwords.

TECHNIQUE 12 Account expiration

This is another search scenario, except this time we'll be using the `accountexpires` attribute. One big plus of creating search scripts in this way is that the only real change is the LDAP filter. The body of the script remains the same.

PROBLEM

We need to know which accounts will expire within a given time frame.

SOLUTION

Modifying our LDAP filter to use the `accountexpires` attribute enables us to find accounts that will expire within a certain number of days, as shown in listing 5.21. This is a variation on the password expiration script we saw previously. Set the start and end dates of our search ❶. In this case, we're interested in accounts that will expire in the next 60 days. Get the current domain root and create a searcher ❷. The search filter is simpler in that we're looking at the user object class and we want to find accounts where the `accountexpires` attribute falls between our two given dates ❸. We use `FindAll()` because we expect multiple results and we display the results ❹ as previously.

Listing 5.21 Account expiration check

```
$now = (Get-Date).ToFileTime()
$end = ((Get-Date).Adddays(60)).ToFileTime()      <-- 1 Set dates

$dom = [System.DirectoryServices.ActiveDirectory.Domain]
::GetCurrentDomain()

$root = $dom.GetDirectoryEntry()

$search = [System.DirectoryServices.DirectorySearcher]$root   <-- 2 Create searcher
$filt = "(&(objectclass=user)" +
"(accountexpires<=$end)" +
"(accountexpires>=$now))"

$search.Filter = $filt      <-- 3 Search filter

foreach ($result in $results){
    $result.properties.distinguishedname      <-- 4 Display results
}
```

DISCUSSION

Using the cmdlets is easy. All we need to do is define the end date of our search. Using the Microsoft cmdlet, we have this syntax:

```
Search-ADAccount -AccountExpiring `
-TimeSpan 60.00:00:00 -UsersOnly |
Format-Table Name, Distinguishedname
```

The Quest cmdlet has a simpler syntax:

```
Get-QADUser -AccountExpiresBefore $((Get-Date).AddDays(60))
```

With the Microsoft cmdlets, we use a `TimeSpan` to look 60 days ahead. We use the `-UsersOnly` parameter to only give us user accounts. The Quest cmdlet only has to be given the date that's 60 days ahead.

This completes our look at user accounts in Active Directory. You've seen a lot of material in this section that should cover most of your needs for automating the administration of user accounts. The scripts are easily modifiable, especially the search and modification scripts. They can all easily be adapted to accept parameters or to read from a file using the examples already given. I'm going to round off the chapter with a look at Active Directory groups.

5.4 Active Directory groups

Active Directory groups are manipulated in a similar manner to the local groups we've already seen. We have the alternative of using cmdlets in this case. We'll look at creating and modifying groups, and finish the section by discovering how to display nested group memberships from the perspective of a group and a user-something you definitely can't do in the GUI.

TECHNIQUE 13 Group creation

Group creation is similar to creating local groups.

PROBLEM

We need to create an Active Directory group.

SOLUTION

The group can be created using ADSI in a similar manner to creating a user in Active Directory, as shown in listing 5.22. There are a number of group types available in Active Directory. We start by creating constants that define the available types and scopes of groups ❶ (in listing 5.22). We bind to the OU where we'll create the group ❷. The group type and scope are combined at the bit level using a binary or operation ❸. I deliberately made this a universal group so that it's obvious that this works. The default group is a global security group. The group is created ❹ and immediately saved.

Listing 5.22 Creating Active Directory group

```
$global = 0x00000002
$domainlocal = 0x00000004
$security = 0x80000000
$universal = 0x00000008

$ou = [ADSI]"LDAP://ou=All Groups,dc=manticore,dc=org"
$grouptype = $security -bor $universal

$newgroup = $ou.Create("Group", "cn=UKPMs")
$newgroup.SetInfo()

$newgroup.GroupType = $grouptype
$newgroup.samAccountname = "UKPMs"
$newgroup.SetInfo()
```

❶ Set constants
❷
❸ Set group type
❹
❺
❻ Set samAccountname

Processing is completed by setting the group type ❺ and a `samaccountname` ❻. We need `samaccountname` or a random one is generated. A final `SetInfo()` writes everything back to the database.

DISCUSSION

If we use the cmdlets, we need to supply the information shown. The code matches the script, but each cmdlet is only one line of code. We start with the Microsoft cmdlet, `New-ADGroup`, and then look at the Quest cmdlet. `New-QADGroup`:

```
New-ADGroup -Name "English Scientists" -SamAccountName EngSci `
 -GroupCategory Security -GroupScope Global `
-DisplayName "English Scientists" `
 -Path "OU=England,dc=manticore,dc=org" `
```

```
-Description "Members of this group are English Scientists"

New-QADGroup -Name "USPres" -SamAccountName "USPres" `
-GroupType "Security" -GroupScope "Universal" `
-ParentContainer "ou=All Groups,dc=manticore,dc=org"
```

After creating our group, we need to populate it with members.

TECHNIQUE 14 Changing membership

Managing group membership will be a mixture of manual and automated procedures. I hate to say it, but not everything can be automated. If you can use the cmdlets, they're ideal for adding single users to a group. If you're creating a group with a number of users that can be identified to an LDAP search, then use the following script as a guide. It could just as easily be searching on a department or location. If the users are scattered across your Active Directory, then collect their names into a CSV file and modify the script to read the file and add the users to a group.

Group membership can also be set as the user account is created.

PROBLEM

All of the users in an OU need to be put into a group.

SOLUTION

An LDAP search filter is used to find all of the user accounts in a given OU, and we can use that information to add the users to the group, as in listing 5.23. We start by creating a directory entry ❶ (in listing 5.23) for the group. A `directorysearcher` ❷ is created to find all of the users in the OU. Note that we set the root of the search to the OU. There's no need to search the whole directory when we know the users are in a single OU.

Listing 5.23 Changing Active Directory group membership

```
$group = [ADSI]"LDAP://cn=UKPMs,ou=All Groups,dc=manticore,dc=org"   ❶ Group
$root = [ADSI]"LDAP://ou=England,dc=manticore,dc=org"
$search = [System.DirectoryServices.DirectorySearcher]$root
$search.Filter = "(&(objectclass=user)(objectcategory=user))"        ❷ Search for users
$result = $search.FindAll()

foreach ($user in $result)                                           ❸ Loop through results
{
  $group.Add("LDAP://" + $user.properties.distinguishedname)         ❹ Add user
  $group.SetInfo()                                                   ❺ Save

$message = $user.properties.distinguishedname +
 " added to group " + $group.cn
Write-Host $message                                                  ❻ Message
}
```

We loop through our results ❸ and use the `Add()` method of the group to add ❹ the user into the group. We're constructing the AD path for the user, which is the input parameter the method expects. `$user.properties.distinguishedname` is used to access the `distinguished` name property because we're dealing with a `directorysearcher` resultset rather than a user object.

As usual, we use `SetInfo()` to write ❺ the information back to disk. The script finishes by writing a message ❻ to say the user has been created. If we wanted to remove users from a group, we could use the `Remove()` method instead of `Add()`.

DISCUSSION

We can use the cmdlets in a number of ways to solve this problem. One solution is to search on an attribute and pipe the results into the cmdlet we use to add a group member:

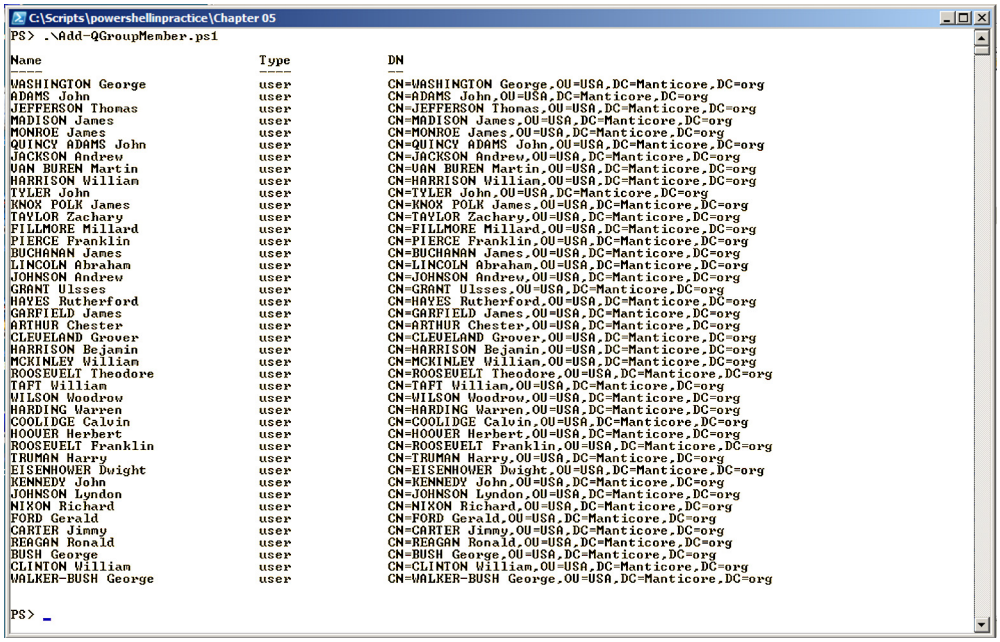
```
Get-ADUser -Filter {Title -eq "Scientist"} `
-SearchBase "OU=England,dc=manticore,dc=org" | foreach {
  Add-ADGroupMember -Identity EngSci -Members $($_.DistinguishedName) }
```

Quest has analagous cmdlets:

```
Get-QADUser -SearchRoot "ou=USA,dc=manticore,dc=org" |
ForEach-Object {Add-QADGroupMember
-Identity "CN=USPres,OU=All Groups,DC=Manticore,DC=org"
-Member $_.distinguishedname }
```

Use `Get-QADUser` (equivalent to a directory searcher) pointed at the OU with the users. Pipe the results into a `foreach` where we use `Add-QADGroupMember` to add the user to the group. The `-Identity` parameter refers to the group, and `-Member` to the user. The cmdlets automatically print the results on screen as shown in figure 5.15.

After creating our groups and populating them with users, we may need to change the scope of the group.



```
C:\Scripts\powershellinpractice\Chapter 05
PS> .\Add-QGroupMember.ps1

Name                          Type            DN
----                          ----            --
WASHINGTON George             user            CN=WASHINGTON George,OU=USA,DC=Manticore,DC=org
ADAMS John                    user            CN=ADAMS John,OU=USA,DC=Manticore,DC=org
JEFFERSON Thomas              user            CN=JEFFERSON Thomas,OU=USA,DC=Manticore,DC=org
MADISON James                 user            CN=MADISON James,OU=USA,DC=Manticore,DC=org
MONROE James                  user            CN=MONROE James,OU=USA,DC=Manticore,DC=org
QUINCY ADAMS John             user            CN=QUINCY ADAMS John,OU=USA,DC=Manticore,DC=org
JACKSON Andrew                user            CN=JACKSON Andrew,OU=USA,DC=Manticore,DC=org
VAN BUREN Martin              user            CN=VAN BUREN Martin,OU=USA,DC=Manticore,DC=org
HARRISON William              user            CN=HARRISON William,OU=USA,DC=Manticore,DC=org
TYLER John                    user            CN=TYLER John,OU=USA,DC=Manticore,DC=org
KNOX POLK James               user            CN=KNOX POLK James,OU=USA,DC=Manticore,DC=org
TAYLOR Zachary                user            CN=TAYLOR Zachary,OU=USA,DC=Manticore,DC=org
FILLMORE Millard              user            CN=FILLMORE Millard,OU=USA,DC=Manticore,DC=org
PIERCE Franklin               user            CN=PIERCE Franklin,OU=USA,DC=Manticore,DC=org
BUCHANAN James                user            CN=BUCHANAN James,OU=USA,DC=Manticore,DC=org
LINCOLN Abraham               user            CN=LINCOLN Abraham,OU=USA,DC=Manticore,DC=org
JOHNSON Andrew                user            CN=JOHNSON Andrew,OU=USA,DC=Manticore,DC=org
GRANT Ulsses                  user            CN=GRANT Ulsses,OU=USA,DC=Manticore,DC=org
HAYES Rutherford              user            CN=HAYES Rutherford,OU=USA,DC=Manticore,DC=org
GARFIELD James                user            CN=GARFIELD James,OU=USA,DC=Manticore,DC=org
ARTHUR Chester                user            CN=ARTHUR Chester,OU=USA,DC=Manticore,DC=org
CLEVELAND Grover              user            CN=CLEVELAND Grover,OU=USA,DC=Manticore,DC=org
HARRISON Bejanin              user            CN=HARRISON Bejanin,OU=USA,DC=Manticore,DC=org
MCKINLEY William              user            CN=MCKINLEY William,OU=USA,DC=Manticore,DC=org
ROOSEVELT Theodore            user            CN=ROOSEVELT Theodore,OU=USA,DC=Manticore,DC=org
TAFT William                  user            CN=TAFT William,OU=USA,DC=Manticore,DC=org
WILSON Woodrow                user            CN=WILSON Woodrow,OU=USA,DC=Manticore,DC=org
HARDING Warren                user            CN=HARDING Warren,OU=USA,DC=Manticore,DC=org
COOLIDGE Calvin               user            CN=COOLIDGE Calvin,OU=USA,DC=Manticore,DC=org
HOOVER Herbert                user            CN=HOOVER Herbert,OU=USA,DC=Manticore,DC=org
ROOSEVELT Franklin            user            CN=ROOSEVELT Franklin,OU=USA,DC=Manticore,DC=org
TRUMAN Harry                  user            CN=TRUMAN Harry,OU=USA,DC=Manticore,DC=org
EISENHOWER Dwight             user            CN=EISENHOWER Dwight,OU=USA,DC=Manticore,DC=org
KENNEDY John                  user            CN=KENNEDY John,OU=USA,DC=Manticore,DC=org
JOHNSON Lyndon                user            CN=JOHNSON Lyndon,OU=USA,DC=Manticore,DC=org
NIXON Richard                 user            CN=NIXON Richard,OU=USA,DC=Manticore,DC=org
FORD Gerald                   user            CN=FORD Gerald,OU=USA,DC=Manticore,DC=org
CARTER Jimmy                  user            CN=CARTER Jimmy,OU=USA,DC=Manticore,DC=org
REAGAN Ronald                 user            CN=REAGAN Ronald,OU=USA,DC=Manticore,DC=org
BUSH George                   user            CN=BUSH George,OU=USA,DC=Manticore,DC=org
CLINTON William               user            CN=CLINTON William,OU=USA,DC=Manticore,DC=org
WALKER-BUSH George            user            CN=WALKER-BUSH George,OU=USA,DC=Manticore,DC=org


PS> _
```

Figure 5.15 Output when using `Add-QADGroupMember`

TECHNIQUE 15 Changing scope

Groups can be changed from distribution lists to security groups (going the other way, you'll lose the permissions the group has) and the group scope can be changed within the limits given next. Distribution groups don't have their own constant, so just leave out the security group value.

Only some group scope changes are supported:

- Universal to global
- Global to universal
- Domain local to universal
- Universal to domain local

In all cases, the group membership has to support the new scope; for instance a global group can't be changed to a universal group if it's a member of other global groups.

PROBLEM

Our universal group must be changed to a global group.

SOLUTION

The group scope is changed by modifying the `grouptype` attribute, as shown in listing 5.24. The script starts by defining the constants ❶ that we use to create the group. Comparison with listing 5.22 will show them to be the same as used in that script. We have to get a directory entry for the group ❷, and create ❸ and set the new group type ❹. The script finishes by saving the change to disk ❺. The creation of the group type is a binary bit operation as in listing 5.22.

Listing 5.24 Changing Active Directory group scope

```
$global = 0x00000002
$domainlocal = 0x00000004
$security = 0x80000000
$universal = 0x00000008

$group = [ADSI]"LDAP://cn=USPres,ou=All Groups,dc=manticore,dc=org"

$grouptype = $security -bor $global
$group.GroupType = $grouptype
$group.SetInfo()
```

❶ Set constants
Get group ❷
❸ Create group type
❹ Set group type
❺ Save

DISCUSSION

The change can be accomplished by using cmdlets. We define the group together with the new type and scope:

```
# Microsoft
Get-ADGroup -Identity EngSci
Set-ADGroup -Identity EngSci  -GroupScope Universal
Get-ADGroup -Identity EngSci

# Quest
Set-QADGroup -Identity "cn=UKPMs,ou=All Groups,dc=manticore,dc=org"
-GroupScope "Global" -GroupType "Security"
```

We need to consider two final tasks regarding groups to complete our work with Active Directory. One question that will arise is "Which groups is this user a member of?" But before we consider that, we need to be able to find all of the members of a group.

TECHNIQUE 16 Finding group members

Discovering the members of a group can be thought of as two separate problems. We have a problem—the direct group membership—that can be resolved easily. This is the list of members you'd see on the Members tab of the Properties dialog in Active Directory Users and Computers.

The second problem is more complex, in that we want to find all of the members, including those users that are members of a group—members of the group in which we're interested. The group nesting may occur to any number of levels.

PROBLEM

We need to find all the members of a group.

SOLUTION

We solve this problem by creating a function that will list the group members, as shown in listing 5.25. If a member is itself a group, we get the function to call itself using the name of that group. This is known as *recursion*. The primary goal of this section is to resolve the nested group membership. But before we review that script, we'll look at reading the direct membership of a group:

```
$group = [ADSI]"LDAP://cn=UKPMs,ou=All Groups,dc=manticore,dc=org"
$group.member | Sort-Object
```

After retrieving a directory entry object for the group, we can display the members using `$group.member`. Piping this into a sort makes the output more readable.

Listing 5.25 Get nested group membership

```
function resolve-group{
param ($group)
    foreach ($member in $group.member){
        $obj = [ADSI]("LDAP://" + $member)
        $global:members += $obj.distinguishedname
        if ($obj.objectclass[1] -eq 'group'){resolve-group $obj}
    }
}

$global:members = @()
$ldp = "LDAP://cn=International,ou=All Groups,dc=manticore,dc=org"
$group = [ADSI]$ldp

resolve-group $group
$global:members | Sort-Object -Unique
```

4 Loop through members
5 Add to members list
Call function
6
1 Define array
2 Directory entry
3 Call function
7 Display all members

Alternatively, we can use the Microsoft cmdlet:

```
Get-ADGroupMember -Identity EngSci | select Name, distinguishedname
```

The Quest alternative gives us:

```
Get-QADGroupMember -Identity "cn=USPres,ou=All Groups,dc=manticore,dc=org"
```

Discovering the nested group membership is more complicated than retrieving the membership of a single group, as listing 5.25 shows. The script consists of two parts: a function, `resolve-group`, that reads the group membership, and the main part of the script that gets the group and displays the membership. We start the script by creating an empty array (developers will refer to this as *declaring* an array) ❶. The point to note here is the way the variable is defined: `$global:members`. The addition of `global:` to the variable makes it a variable of global scope, meaning that we can access the same variable in the main part of the script and in the function. This will be important.

ADSI is used to get a directory entry ❷ for the group. We then call the `resolve-group` function ❸, passing in the group as a parameter. The `$group` within the function is in a different scope than the `$group` outside the function.

A `foreach` loop is used to read the group membership ❹ from the member property. A directory entry is created for each member ❺ and added to our globally available array. We test the group member, and if it's a group ❻, we call the `resolve-group` function using the member as a parameter.

DISCUSSION

Congratulations! You now understand recursion, as the function will keep calling itself as many times as necessary. As the array we created to hold the membership is global in scope, it can be accessed through the various levels of recursion.

Once the function has finished processing the direct and nested membership, we return to the main part of the script. The contents of the array are sorted and the unique values ❼ are displayed. Using the `-Unique` parameter prevents duplicate entries from being displayed, and means that we don't have to write code to deal with them. This makes the script easier to write and understand.

There's a simpler way to get this information using the Microsoft cmdlet `Get-ADGroupMember`. The `-Recursive` parameter displays nested group membership:

```
Get-ADGroupMember -Identity international -Recursive |
select Name, DistinguishedName
```

The Quest alternative is to use the `-Indirect` parameter:

```
Get-QADGroupMember -Identity 'manticore\international' -Indirect
```

Having mastered recursion in the previous example, we'll use it again to determine all of the groups of which a particular user is a member.

TECHNIQUE 17 Finding a user's group membership

One last Active Directory script and then we're done.

PROBLEM

We need to find all of the groups of which the user is a member.

SOLUTION

The `memberof` attribute holds the groups of which the user is a member. We can recursively check those groups for other groups to determine the full list of groups where the user is a member, as shown in listing 5.26. The processing starts by getting a direc-

tory entry for the user ❶. We use the `memberof` property to find the groups of which the user is a direct member ❷. The group is passed into the function `resolve-membership`, where the distinguished name is written ❸ to screen.

For each of the groups, we get a directory entry ❹ and test to see if it's a member of any groups. If it is, we call the function with the name of each group. ❺ Recursion keeps this script compact. It is a topic that many find difficult but the examples in the book should make it easier to use. Once you have worked through a few scripts of your own you'll be proficient.

Listing 5.26 Get user's group membership

```
function resolve-membership{
param ($group)
    Write-Host $group                                    ❸ Write group

    $group2 = [ADSI]("LDAP://" + $group)                 ❹ Group directory entry
    if ($group2.memberof -ne $null){
        foreach ($group3 in $group2.memberof){           ❺ Call function
                    resolve-membership $group3 }
    }
}

$user = [ADSI]"LDAP://CN=WELLESLEY Arthur,OU=England,DC=Manticore,DC=org"   User directory entry ❶

foreach ($group in $user.memberof){resolve-membership $group}   Call function ❷
```

DISCUSSION

I haven't produced a version using the cmdlets, as there isn't a built-in way to produce this information, and we just replace the `[ADSI]` lines ❹ and ❶ in listing 5.26 with `Get-ADGroup/Get-QADGroup` and `Get-ADUser/Get-QADUser` respectively.

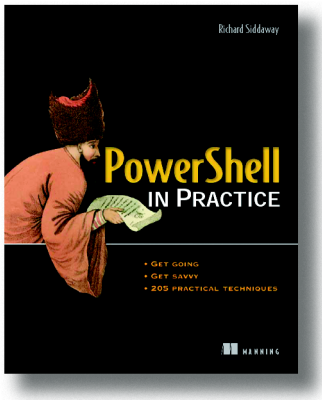
5.5 Summary

Automating Active Directory administration involves working with users and groups or performing searches. We can perform these tasks by scripting based on ADSI or by using the AD cmdlets from either Microsoft or Quest .

Creation and modification scripts follow a pattern of getting a directory object, making changes (or creating a child object), and saving back to the database. Searching has its own pattern of defining the root of the search, defining the search filter, performing the search, and displaying the results.

There's useful functionality in the `System.DirectoryServices.Accountmanagement` classes, though a few holes also exist.

After creating and modifying our user account, it's time to turn our attention to our email system. Email has become a business critical tool, and by combining our mailbox and user account administration techniques, we can automate and streamline our processes.



PowerShell in Practice covers 205 individually tested and ready-to-use techniques, each explained in an easy problem/solution/discussion format. The book has three parts. The first is a quick overview of PowerShell. The second, Working with People, addresses user accounts, mailboxes, and desktop configuration. The third, Working with Servers, covers techniques for DNS, Active Directory, Exchange, IIS, and much more. Along the way, you'll pick up a wealth of ideas from the book's examples: 1-line scripts to full-blown Windows programs.

PowerShell is a powerful scripting language that lets you automate Windows processes you now manage by hand. It will make you a better administrator.

What's inside:

- Basics of PowerShell for sysadmins
- Remotely configuring desktops and Office apps
- 205 practical techniques

This book requires no prior experience with PowerShell.

index

Symbols

Numerics

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

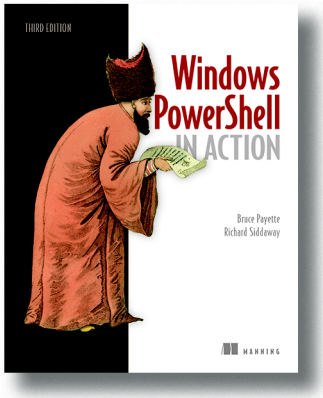
T

U

V

W

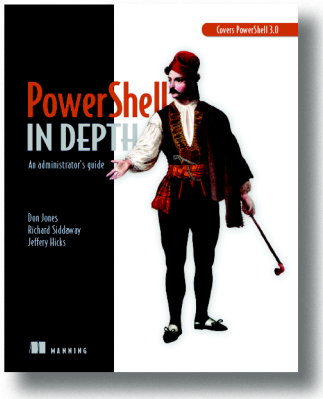
Z



Windows PowerShell in Action, Third Edition

by Bruce Payette and Richard Siddaway

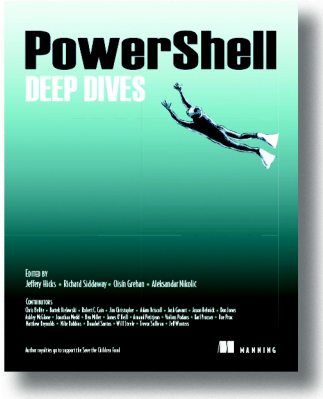
ISBN: 9781633430297
625 pages
$59.99
Fall 2016



PowerShell in Depth, Second Edition

by Don Jones, Jeffrey Hicks, and Richard Siddaway

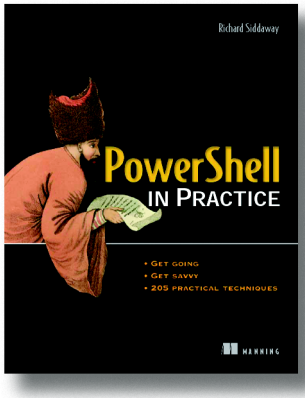
ISBN: 9781617292187
744 pages
$59.99
October 2014



PowerShell Deep Dives

Edited by Jeffery Hicks, Richard Siddaway, Oisin Grehan, and Aleksandar Nikolic

ISBN: 9781617291319
464 pages
$49.99
July 2013



PowerShell in Practice

by Richard Siddaway

ISBN: 9781935182009
584 pages
$49.99
June 2010