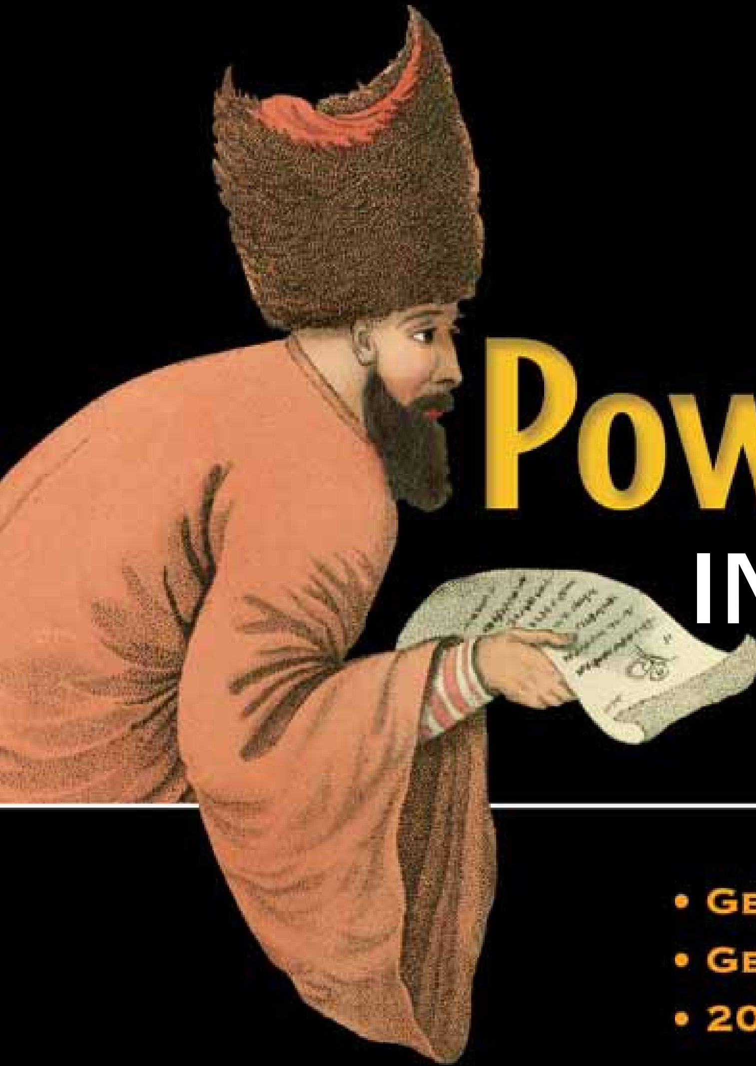
Richard Siddaway



PowerShell IN PRACTICE

- GET GOING
- GET SAVVY
- 205 PRACTICAL TECHNIQUES

SAMPLE CHAPTER

PowerShell in Action

by Richard Siddaway

Chapter 8

Copyright 2010 Manning Publications

brief contents

PART 1 GETTING STARTED WITH POWERSHELL 1

1 ■ PowerShell fundamentals 3
2 ■ Learning PowerShell 30
3 ■ PowerShell toolkit 63
4 ■ Automating administration 92

PART 2 WORKING WITH PEOPLE 121

5 ■ User accounts 123
6 ■ Mailboxes 159
7 ■ Desktop 188

PART 3 WORKING WITH SERVERS 221

8 ■ Windows servers 223
9 ■ DNS 257
10 ■ Active Directory structure 287
11 ■ Active Directory topology 321
12 ■ Exchange Server 2007 and 2010 352
13 ■ IIS 7 383
14 ■ SQL Server 414
15 ■ PowerShell innovations 448

Part 3

Working with servers

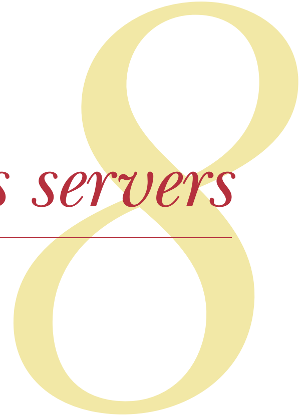
In the previous two parts of the book, we learned how to use PowerShell. We've taken that knowledge and applied it to performing administration tasks that directly impact the user community. In part 3, we'll turn our attention to working with Windows servers and the applications that run on our servers.

Desktop administration tends to be about performing the same acts on many machines. Server administration is about performing a wider range of actions on a smaller number of machines. The other major difference is that an individual desktop machine having problems doesn't impact the business (unless it belongs to a senior manager, of course). A server that's having problems could stop a business-critical application and have a major impact on the business. Server administration has a bigger impact on the user community than desktop administration, as it affects the whole user community rather than an individual.

Server administration should be taken to mean remote administration. The servers are usually in a data center that may be at the other end of the building or the other end of town. In some cases, the data center may be in a different part of the country or even a different country. We shouldn't assume that we can gain physical access to the servers. Work remotely and embrace automation. It leaves more time for the fun things such as investigating what we can administer with PowerShell and how we can perform those actions remotely.

We'll start our look at servers in chapter 8 by understanding how to administer the basic aspects of a Windows server, including the filesystem, services, processes, the Registry and the event logs. While reading this chapter, keep in mind that parts of the material we covered in chapter 7 can be applied to servers also.

DNS, AD structure, AD topology, Exchange 2007, IIS 7, and SQL Server are covered in chapters 9 to 14. In each chapter, we'll look at how to administer the individual application from an "on-server" and remote viewpoint. In many cases, the same tools allow both. Some applications, such as IIS 7 and SQL Server, supply multiple methods of administration by PowerShell. We'll examine the options and make recommendations as to the best tools to use in particular circumstances. Chapter 15 brings together some of the innovations in the PowerShell world and how these can be applied.



Windows servers

This chapter covers

- Services and Processes
- Administering the filesystem
- Working with the Registry
- Managing Event Logs

This chapter opens the third and major part of the book, where we look at how we can administer our Windows servers, and the applications they host, by using PowerShell.

> **REMEMBER** Many of the scripts and tasks that were covered in chapter 7 also need to be performed on servers. Likewise, some of the material in this chapter can be applied to desktop machines. The two chapters form a bridge between the desktop and server aspects of administrator's activities.

In the introduction to this part, I made the point that server administration should be viewed as an activity to be performed remotely. PowerShell has a number of ways of supplying the capability to perform remote administration:

- Some cmdlets have a remote capability.
- Scripts can use WMI or .NET to access remote systems.
- PowerShell v2 brings a remoting capability based on the *Windows Remote Management* service.
- Some PowerShell providers, for example the one in SQL Server 2008, provide access to remote machines.

The techniques that apply in a given situation will be highlighted, as will any known issues with the remote capability. In all cases, firewalls can block access via these remoting technologies. Make sure that the firewall on the remote computer doesn't block the required protocols.

In this chapter we'll start by looking at how we'll administer our Windows server. This will create the foundation for later chapters, when we look at the applications hosted on the servers.

8.1 Automating server administration

One of my main areas of work is as an IT architect. I spend my time designing and implementing systems to solve business problems. One of the first things I have to do when presented with a new problem is to determine the customer's requirements. The same concept applies to server administration. What are our requirements? We can't decide how we'll automate our administration until we know what we'll administer.

When we think about administering the server itself rather than the applications hosted on it, we tend to arrive at the following suspects:

- Services
- Processes
- Filesystem
- Registry
- Event logs

The system configuration section from chapter 7 is also applicable to servers. Many of the scripts presented in that section are WMI-based and are therefore directly applicable to remote administration. Anything to save you from running around so much.

Services and processes define what's running on the server. These, in many cases, are the applications we'll be dealing with in subsequent chapters. At this stage we're concerned with the basics; are they running and how do we manage them?

The filesystem is an essential part of any server. We need to be able to work with files and the folder hierarchy irrespective of the applications in use. The registry is a repository of configuration information that we have to be able to access. We'll see the registry provider in action. Now you can do a `dir` through the registry. Awesome!

Event logs are where we find the diagnostic information we need when things go wrong. Reading the data in the event logs is good, but we also need to be able to write to the event logs and even create our own logs.

We need to consider one more thing before we dive into all the fun stuff with PowerShell. There's a new kid on the block in the Windows world: Windows Server Core.

8.1.1 Server Core

Server Core is an install option for Windows Server 2008 (and R2). It's a stripped down version without a GUI, which may seem an odd option for a Windows system. The command prompt is the only way to locally manage a Server Core machine, though GUI tools can be used from a remote machine. Once chosen, the only way to revert to a GUI-based version of Windows is to reinstall. Likewise, a GUI-based version of Windows can't be converted to a Server Core version without reinstalling.

One of the real benefits of running Server Core is the reduced number of services, leading to a reduced patching requirement. Anything that reduces patching is a benefit to an administrator.

There are a number of roles for which Server Core is ideal, including domain controllers, DNS servers, or file and print servers. One thing that's missing in the original Windows Server 2008 release is the .NET framework. This means that ASP.NET can't be used, and also PowerShell can't be installed in the normal way. (Dmitry Sotnikov has shown how it's possible to install PowerShell v2 on a Server Core machine: http://dmitrysotnikov.wordpress.com/2008/05/15/powershell-on-server-core/.)

> **NOTE** Sotnikov's method for installing PowerShell on Server Core isn't a supported or recommended approach for production machines. It should be viewed as a proof of concept. Many aspects of a Server Core installation can be managed remotely via WMI.

In Windows Server 2008 R2, a subset of the .NET framework can be installed on Server Core, meaning that PowerShell is available as an optional feature. If you're using Server Core in your environment consider upgrading to R2 if at all possible.

The first things we need to look at are services and processes. In other words, what's running on our systems and how can we manage them.

8.2 Services and processes

Services and processes together make up the applications running on our systems. Some services such as Exchange or SQL Server provide applications; others provide the background functionality we need for a healthy, working system. Knowing what's running, and possibly more importantly, what should be running, gives us a powerful administrative handle on our systems.

Some of the processes we have running are from applications we've explicitly started. PowerShell can be used to manage the processes we have running, including the creation of new processes. There's a mesh of dependencies between services on a Windows system. We could use the GUI tool to trace these, but it's easier with PowerShell as we'll see. We can even display the information graphically, which can make the dependencies more obvious.

TECHNIQUE 61 Service health check

One common troubleshooting scenario is that a service has stopped for some reason. It's not necessarily the primary service that's causing the problem, but a service on which that service is dependent that's having problems. Big fleas have little fleas upon their back to bite 'em and little fleas have...

PROBLEM

We need to view the services installed on the system to determine which services are running and if any of their dependent services haven't started.

SOLUTION

PowerShell provides a `Get-Service` cmdlet that we can use to investigate the status of our services. In PowerShell v1, this script can only be run against the local system. If we want to work with remote systems, we need to use WMI. PowerShell v2 adds a `computername` parameter so we could modify the script to incorporate the dealing with services on remote computers.

> **NOTE** PowerShell v2 also adds `DependentServices` and `RequiredServices` parameters. These will only show the status of the dependent or required service. The status of the parent service isn't shown. The script could be modified to work with these parameters if required.

The script in listing 8.1 starts by using `Get-Service` in its default mode to generate a list of services installed on the local machine ❶. The list is sorted by the display name of the service. This is the name that's shown in the Services administration tool. The display name isn't necessarily the same as the service name, as can be seen by running `Get-Service`, or by looking at the examples in table 8.1.

Name	Display name
Afd	Ancilliary Function Driver for Winsock
Tdx	NetIO Legacy TDI Support Driver
NSI	Network Store Interface Service

Table 8.1 Sample service names and display names.

The services are piped into a `foreach` cmdlet that performs the bulk of the work in the script. An initial check on the status of the service is used to determine how the information is displayed. A service that's stopped ❷ will be displayed with red text on a white background, whereas a running service ❸ will be displayed in the normal colors normally used by PowerShell. Note how `` `n `` is used to force the display to be on a new line in the `Write-Host`.

> **NOTE** The colors can be easily changed depending on the colors used on your machine. The list of allowable colors can be seen in the help file for `Write-Host`.

After displaying the status of the service, we perform a `Get-Service` on the individual services and expand the property that holds the services that are depended on ❹. Another `foreach` is used to display whether these services are stopped ❺ or running ❻.

Listing 8.1 Service health check

```
Get-Service |Sort -property DisplayName  | foreach{          ❶ List services
    If ($_.Status -eq "Stopped") {
       Write-Host "`n $($_.DisplayName) is $($_.Status)" `    ❷ Display stopped service
            -foregroundcolor Red -backGroundColor White}
    Else {Write-Host "`n $($_.DisplayName) is $($_.Status)" }   ❸ Display running service

    Get-Service $_.Name |
        Select -ExpandProperty ServicesDependedOn | foreach{    ❹ List required services
    If ($_.Status -eq "Stopped") {
         Write-Host "`t is dependent on $($_.DisplayName) `
             which is $($_.Status)" `
             -foregroundcolor Red -backGroundColor White }     ❺ Display stopped
    Else { Write-Host "`t is dependent on $($_.DisplayName) `
        which is $($_.Status)" }                               ❻ Display running
      }
}
```

DISCUSSION

Exchange Server 2007 has a cmdlet that produces similar output for the Exchange services. Variants could be written that just tested the SQL Server services, for instance.

One interesting variation is to display the results graphically. Netmap is a tool that can be used to create and view network graphs. PowerShell scripts to work with it can be downloaded from http://dougfinke.com/blog/?p=465. As an example, we could look at the DHCP client service. The services that the DHCP client is dependent on are listed in table 8.1. But as figure 8.1 shows, there's a further layer of dependencies in that the Tdx and NSI services both have dependent services. This diagram could be extended to include all services, but the level of detail wouldn't lend itself to being easily reproduced.

Now that we know what services are on our systems, how do we go about managing them?

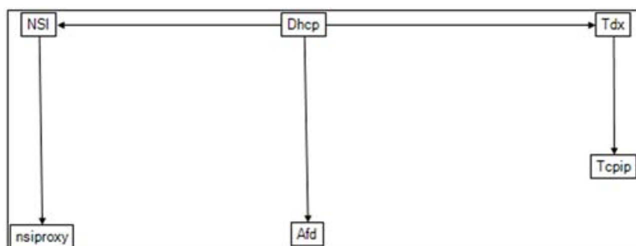


Figure 8.1 Hierarchy of services on which the DHCP client service is dependent. There are three primary dependencies, two of which have a further dependent service. All of the services in the hierarchy must be working for DHCP to work.

TECHNIQUE 62 Managing services

There are a number of cmdlets available to manage services:

- `Get-Service`
- `New-Service`
- `Restart-Service`
- `Resume-Service`
- `Set-Service`
- `Start-Service`
- `Stop-Service`
- `Suspend-Service`

The cmdlet names are self-describing in terms of functionality. In PowerShell v1, the service cmdlets are restricted to the local machine. This is extended in version 2, in that `Get-Service` and `Set-Service` have a `-computername` parameter added so we can work with remote systems. It's not possible to use the `ReStart-`, `Resume-`, `Start-`, `Stop-`, or `Suspend-Service` cmdlets against a remote machine. We can perform these actions using WMI or the `Set-Service` cmdlet. `Set-Service` can work with the service startup type, but `Get-Service` can't display the startup type because the .NET object doesn't incorporate the startup type.

PROBLEM

Many of our major applications run as services, for example SQL Server and Exchange. How can we quickly, and easily, determine which services are running if the users report problems accessing the systems?

SOLUTION

PowerShell and WMI, where necessary, enable us to view and manage the services on our systems. The first thing we usually want to know is whether our services are running. In PowerShell, whenever we have to retrieve information, the verb to use is *Get*. We're dealing with services, so our command in listing 8.2 becomes `Get-Service` ❶. The standard output is shown in the truncated listing and consists of the service name, display name, and its status in terms of whether it's running. With PowerShell vv2, we can interrogate the services on remote machines by using a `-ComputerName` parameter. Alternatively, we can use the `Win32_Service` WMI class. The cmdlets that have a `-ComputerName` parameter use the local credentials and don't allow them to be changed. `Get-WmiObject` does allow the use of alternate credentials.

The PowerShell pipeline enables us to take the results of our discovery ❷. In this case, we want all of the services whose name starts with the letter S and we can use `Where-Object` to limit our output to those services that are actually running. We could take this a step further and pipe these results in to a `Stop-Service` cmdlet. We find all of the running services whose names start with S and stop them.

> **NOT ME** I often include this in PowerShell introductory sessions. Strangely enough, none of the people who encourage me to try this in the demonstration are willing to perform the same action on their machines.

This leads to a reminder about one of PowerShell's failsafe mechanisms—the `-whatif` parameter ❸. We can see what the results of our actions would be if we actually

performed them by using -whatif, as in listing 8.2. It's a good idea to test these types of actions using -whatif.

Listing 8.2 Managing services

```
PS> Get-Service                                    ❶ Get-Service

Status   Name               DisplayName
------   ----               -----------
Running  AEADIFilters       Andrea ADI Filters Service
Running  AeLookupSvc        Application Experience
Stopped  ALG                Application Layer Gateway Service
Running  AppHostSvc         Application Host Helper Service
Running  Appinfo            Application Information
Stopped  AppMgmt            Application Management

Listing truncated to save space
                                                    ❷ Filter Get-Service
Get-Service s* | where {$_.Status -eq "Running"}

Get-Service s* | where {$_.Status -eq "Running"} |  ❸ Stop-Service
Stop-Service -whatif

$s1 = Get-WmiObject -class Win32_Service -Filter "Name='W32Time'"
$s1.stopservice()                                   ❹ WMI and services
```

DISCUSSION

WMI can also be used to manage services. But be aware that the *-Service cmdlets don't use WMI to perform their functions. We can use WMI directly to manage services ❹. Using the Win32_Service class will return information about all services. The -Filter parameter can be used to restrict the action to a single service as shown. Get-Member can be used to view the list of properties and methods available when using this WMI class:

```
$s1 | Get-Member
```

There's a method that will stop the service. A -whatif parameter isn't available when using the StopService() method in WMI. The service will be stopped immediately. WMI gives us the ability to administer services on remote machines. In PowerShell v2, some of this functionality is available through the Set-Service cmdlet.

TECHNIQUE 63 Managing processes

Processes are the other major area we have to look at in this section. We're often dealing with applications such as Microsoft Word when we think of processes. All processes consume resources, usually thought of as CPU cycles and memory (physical and virtual). As administrators, we need to be aware of the resources that applications are consuming. It could be that an application is taking more resources than it should, which is having an adverse effect on other processes.

PROBLEM

One common scenario for an administrator is that the phone rings and an irate voice tells you that the system is running slow, which is stopping everyone from working

properly. There are numerous possible reasons that could cause this slowdown. One area that has to be investigated is how the processes are using CPU and memory.

SOLUTION

`Get-Process` is at the heart of any understanding of resource usage by processes. We can use it to view the CPU and memory usage by process. `Get-Process` is the cmdlet that we need when we want to investigate processes. In PowerShell v2, it gets a `-ComputerName` parameter so that we can look at the processes on remote machines. In PowerShell v1, we can use the `Win32_Process` WMI class to achieve the same goal as `Get-WMIObject` can access remote machines. The default output of `Get-Process` in listing 8.3 ❶ shows a number of useful pieces of information, including:

- CPU usage
- Physical memory usage-both paged and nonpaged
- Virtual memory usage
- Working set size
- Number of handles

We can restrict the data by only asking for particular processes, either filtering by name or by property, as in listing 8.3. `Get-Process` returns a large number of properties. They can be viewed by using `Get-Member` ❷. The default display is to show the processes sorted by name.

Listing 8.3 Managing processes

```
PS> Get-Process                                    ❶ List processes

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     27       1      368       1624    11             496 AEADISRV
    112       3     1096       3972    35            1148 Ati2evxx
Listing truncated for brevity

    574      31    45636      96404   319   540.84   4932 WINWORD
    611      17    27872      32336   174    10.94   4384 wlcomm
   1105      71   116916     103724   373   224.16   3696 wlmail
     37       2      768       2456    22            3100 XAudio

Get-Process | Get-Member                           ❷ Process properties

Get-Process | sort cpu -Descending | Select -First 5
Get-Process | Sort PM -Descending | Select -First 5     ❸ Sorting
Get-Process | Sort WS -Descending | Select -First 5

Get-Process | Select Name, `                       ❹ Calculated display
@{Name="CPU(s)";
Expression={if ($_.CPU -ne $()) {$_.CPU.ToString("N")}}}, `
@{Name="Mem";Expression={($_.PM+$_.WS)/1kb}}, `
@{Name="VM(M)";Expression={[int]($_.VM/1mb)}}, Handles |
Sort Mem -Descending | Select -First 5 |
Format-Table Handles, Mem, "VM(M)", Name -AutoSize

Get-Process notepad | Stop-Process                 ❺ Kill process
```

DISCUSSION

In many cases, we're more interested in the processes that are consuming most of a particular resource, usually CPU or memory. PowerShell has a `Sort-Object` cmdlet. We can sort by various properties and select the top five, or whatever seems suitable, and just display those processes ❸. This is a useful technique, but the results shown by sorting by Paged Memory (PM) and Working Set (WS) for instance will be slightly different. What we really need is a combined measure that will show those processes that take the most memory overall.

We can't do this directly. But by starting with `Get-Process` ❹ we can calculate what we need. The bulk of the work is performed in the `Select-Object` cmdlet. In addition to selecting the process name and handles, we calculate three fields. The CPU(s) field gives a neater display in that CPU usage is rounded to two decimal places. Note how the `if` statement is used in the `Expression` script block. The Mem field simply adds the WS and the PM, with the result returned as kilobytes. As a contrast, the virtual memory is returned as an integer number of megabytes.

We can sort on calculated fields. In this case the combined memory calculation. The first five results are selected and `Format-Table` is used to control the order of display.

> **ONE LINE** In listing 8.3 ❹ is a single line of PowerShell code! It's split for clarity of display purposes.

The thought process that has brought us to this long line of code is typical of the way that we as administrators produce scripts. We start with the output of the cmdlets, then start sorting and selecting specific properties to refine our results. If we can't achieve the results we need, we have to get more complicated and start calculating fields. This approach was described in more detail in chapter 4.

If we find a process that's taking too big a share of the resources, we may need to stop it. We can use `Stop-Process` and ensure we identify the process correctly. There's a big opportunity for mistakes here. A better solution is to make sure we get the correct process and pipe the results into `Stop-Process` ❺. This reduces the chances of a mistake being made.

TECHNIQUE 64 Launching processes

The final activity involving processes we need to consider is how we can create them. Processes are created by the OS, applications, and services. We may need to start our own processes either because we need to start an application or as part of the activity involved in solving a problem.

PROBLEM

We need to start an application on a local or remote system. This application may need to start at a specific time (use Windows scheduler to run PowerShell) or could be used for diagnostic purposes.

SOLUTION

WMI provides two methods we can exploit to solve this problem. The first is usable in PowerShell v1 or v2, but the second method is only applicable to version 2. We also

have a direct method using Start-Process, which is only available in version 2 and only works on the local computer, as shown in listing 8.4.

Listing 8.4 Creating processes

```
$c = [WMIClass]"Win32_Process"                          ❶ Version 1 or 2
$c.Create("notepad.exe")

Invoke-WmiMethod -Class Win32_process -Name Create      ❷ Version 2
-ArgumentList notepad.exe

Start-Process notepad                                   ❸ Version 2, local only
```

DISCUSSION

The object of the exercise in this example is to start a process that allows us to start an instance of Notepad. Notepad is a good test bed when experimenting with processes. It won't damage your system, unless you start hundreds of instances. It's possible to make a mistake, such that you create an infinite loop that continually spawns processes. You'll only do this once and become very paranoid about loops afterward. Yes, this is experience talking but it was a long, long time ago. Honest!

Our first example uses the [WMIClass] accelerator ❶ we discussed in chapter 3. This enables us to create an instance on a WMI class, in this case the Win32_Process class. Unfortunately, this is a two-step process, as we then have to use the Create() method to actually create the process. We need to give the application that'll run in the process as a parameter. If the application path isn't known to Windows, we need to give the full path. This approach can be used on remote computers. We amend the first line to include the computer name and use the full WMI path to the class.

```
$c = [WMIClass]"\\computer1\root\cimv2:Win32_Process"
```

The Create() method is used in exactly the same way.

PowerShell v2 brings a slightly easier to use option in the form of the Invoke-WmiMethod cmdlet ❷. We need to supply the WMI class, the method, and the arguments as shown. This cmdlet has a -ComputerName parameter so that we can work with remote machines:

```
Invoke-WmiMethod -Computername computer1 -Class Win32_process
-Name Create -ArgumentList notepad.exe
```

Start-Process ❸ can be used on the local machine to create a process for an application.

In addition to starting processes, we need to think about terminating processes. We can use WMI, though Remove-WMIObject is only available in PowerShell v2:

```
Get-WmiObject -Class win32_process -Filter "Name='calc.exe'" |
Remove-WmiObject
```

Other alternatives include using the Stop-Process cmdlet on the local machine or using the kill method on the process object:

```
Stop-Process -Name notepad
Get-Process notepad | Stop-Process

$p = Get-Process notepad
$p.Kill()
```

The `Kill()` method is the one I'd recommend least. Ideally, you should use PowerShell v2 and the `Stop-Process` cmdlet so that the `-whatif` and `-confirm` parameters are available.

The combination of PowerShell and WMI is the recommended method for working with remote processes. The remoting capabilities in PowerShell v2 could be used if available. This concludes our work with services and processes. The next major activity we need to consider is administering the filesystem.

8.3 Filesystem

Administrators always seem to be tinkering around in the filesystem. It's where most of us probably started as administrators. Even if we're working with server-based applications such as Exchange or SQL Server, we still need to interact with the filesystem. PowerShell gives us a number of tools for working with the filesystem. We don't have the space to cover all of the possible scenarios in this section. We could probably fill another book with examples of just working with the filesystem. The examples in this section will provide a starting point for further experimentation, and will cover the most common administrative tasks regarding the filesystem.

PowerShell has the concept of *providers* for working with data stores. The filesystem provides the model for providers. Though PowerShell treats the filesystem as just another provider, in reality the other providers often don't supply the same level of functionality as the filesystem provider. Table 8.2 lists the cmdlets that can be used to work with files.

Table 8.2 Cmdlets for working with files

Cmdlet	Synopsis
`Clear-Item`	Deletes the contents of an item, but doesn't delete the item.
`Copy-Item`	Copies an item from one location to another.
`Get-ChildItem`	Gets the items and child items in one or more specified locations.
`Get-Item`	Gets the item at the specified location.
`Invoke-Item`	Performs the default action on the specified item.
`Move-Item`	Moves an item from one location to another.
`New-Item`	Creates a new item.
`Remove-Item`	Deletes the specified items.
`Rename-Item`	Renames an item.
`Set-Item`	Changes the value of an item to the value specified in the command.

In the filesystem provider, a file or folder is regarded as an item, and it's the provider's task to provide the interface to the `*-Item` and `*-ItemProperty` cmdlets. These cmdlets together with a number of others are known as the *core commands* and theoretically should be available in all providers. The full list of core commands can be found by typing:

```
Get-Help about_Core_Commands
```

In addition to being able to work with the files themselves, we need to be able to access the contents of the files, which we can using the cmdlets in table 8.3.

Table 8.3 Cmdlets for working with file content

Cmdlet	Synopsis
`Add-Content`	Adds content to the specified items, such as adding words to a file.
`Clear-Content`	Deletes the contents of a item, such as deleting the text from a file, but doesn't delete the item.
`Get-Content`	Gets the content of the item at the specified location.
`Set-Content`	Writes or replaces the content in an item with new content.

In addition to the cmdlets listed in the table, we have other ways to work with text type files including:

- `Export-Csv`
- `Import-Csv`
- `Out-File`

XML files have been deliberately left until chapter 13. We'll see these cmdlets in action in the rest of this section. The starting point for the filesystem though has to be folders.

TECHNIQUE 65 Creating folders

In the Windows filesystem, folders are used as a method of grouping and organizing files into a treelike structure. Before we can work with files, we need to consider how we'll organize our files and how we'll create the folders we'll need. Examples of creating folders and files are also given in section 2.4.4 dealing with `for`, `do`, and `while` loops, and section 2.4.5 dealing with functions.

PROBLEM

We need a method of creating one or more folders.

SOLUTION

The cmdlet we need to perform this task can be discovered by remembering the verb-noun syntax of PowerShell. If we want to create something, the verb to use is New. In this case, we want to create a folder in the filesystem and folders are counted as items in a provider. This means we need the `New-Item` cmdlet, shown in listing 8.5.

Listing 8.5 Creating folders

```
New-Item -Name TestFolder -Path c:\scripts -ItemType Directory
```

DISCUSSION

`New-Item` only requires three pieces of information in order to create a folder. The name of the folder, the path to the folder (remember the standard limits on path length in Windows), and the fact that we're creating a folder. In the filesystem provider, the `-ItemType` parameter can only accept values of `Directory` or `File`. PowerShell will prompt for a type if you don't supply one.

> **NOTE** There are two aliases available when creating folders—`md` and `mkdir`. They're used in the same way as the cmd.exe commands.

Having created our folder, we now need something to put in it.

TECHNIQUE 66 Creating files

The files on our systems tend to fall into one of two categories. One possibility is that they're application-specific, such as a file created by Word or a SQL Server database file. On the other hand, they may be a text file that we can work with directly in our PowerShell scripts. This assumes that we have the permissions required to work with the particular files in question!

PROBLEM

Our computer systems are never static. This is where some of the challenge and fun comes into being an administrator. Applications take more resources over time. Disks fill up. New applications are introduced. These activities all involve changes to the system.

One thing we need to be aware of is change over time. This can only be recognized if we have a record of previous states. We can view the current state of our processes and services, but to save the previous state, we need to write the data into a file that we can access at a later date. How can we save this information using PowerShell?

SOLUTION

PowerShell provides several ways to create and write to a file, as shown in listing 8.6. The best one to use depends on circumstances. One of the great strengths of PowerShell is that there are multiple ways to achieve a goal. This can also be perceived as a weakness, especially by someone setting out to learn PowerShell. The best advice I can give is to look at the alternatives and settle on which works best in your particular circumstances. This may well involve experimentation. Never be afraid to experiment—that's why you can use the same commands from the prompt as in your scripts!

Listing 8.6 Creating files

```
New-Item -Name testfile.txt -Path c:\scripts\testfolder -ItemType File
-Value "This is a one line file"                                          ❶

Get-Service | Out-File -FilePath c:\scripts\testfolder\sp.txt          ❷
Get-Process | Out-File -FilePath c:\scripts\testfolder\sp.txt -Append

Get-Process |
Export-Csv -Path c:\scripts\testfolder\testprc.csv -NoTypeInformation    ❸
```

We can use `New-Item` to create a file ❶. We follow the same pattern as when creating a folder, and supply the name, the path, and the type of item we're creating. We also have the option to add content to the file at the time of creation by using the `-value` parameter. If this is omitted, an empty file is created, as in the examples in chapter 2. The `Set-Content` cmdlet can be used to put content into the empty file. If it's used against a file that already has content, that content will be overwritten. This option is good if we need to create an empty file or we have content available to write to the file as one piece.

Our second option involves using `Out-File` ❷. In the first part of the script, we pipe the results from using `Get-Service` into our file. The second part appends the results of `Get-Process`. Note the use of the `-Append` parameter. This forces the data to be appended to the end of the file rather than overwriting the file contents, which is the default behavior. One of the best uses of `Out-File` is where we need to keep appending data to a file. `Add-Content` can also be used to append data to a file.

The final option is to use `Export-Csv` ❸. A text file was produced in the previous two cases. In this example, we're creating a delimited file where the fields are separated by commas. We need to give the path to the file pipe in the data we wish to write into the file. The `-NoTypeInformation` parameter prevents the .NET type information from being written into the first row of the file. We don't usually need this information, so it's best to use this parameter as a matter of routine.

NOTE `Export-Csv` doesn't have the capability to append data to a file.

DISCUSSION

In addition to being able to create and write to files, we need to be able to delete them at the appropriate time. The script in listing 8.7 is used to clean up the contents of my Temp folder on a periodic basis (when I remember to do it).

Listing 8.7 Removing files

```
Remove-Item C:\temp\*.tmp -Recurse
Remove-Item C:\temp\low\*.tmp -Recurse
Remove-Item C:\temp\*.log -Recurse
Remove-Item C:\temp\*.txt -Recurse
Remove-Item C:\temp\*.cvr -Recurse
Remove-Item C:\temp\*.od  -Recurse
Remove-Item C:\temp\*.exe -Recurse
Remove-Item C:\temp\*.dll -Recurse
Remove-Item C:\temp\*.xml -Recurse
Remove-Item C:\temp\*.Hxc -Recurse
```

Note the use of the `-Recurse` parameter. This will recursively follow the subfolder tree within the Temp folder to delete all of the files with the given extension. My scripts of this sort tend to be built up over time, so I end up with repeated calls like this (quick and dirty scripting). If we want to make the script more concise, we can always do this:

```
"tmp", "log", "txt", "cvr", "od", "exe", "dll", "xml", "Hxc" |
foreach {Remove-Item "$env:temp\*.$_" -Recurse}
```

Simply pipe the list of extensions into a `foreach cmdlet` that calls `Remove-Item`. This also showcases the use of an environmental variable `temp`, which gives the path to the temporary folder. We access environmental variables via the `$env` variable, which represents the environment drive in PowerShell.

Once we've written our data into a file, we'll need to read the data in the file.

TECHNIQUE 67 Reading files

Being able to write data into our files is a good thing. It's even better if we can access the data in those files. At this point, we can start to save data for reuse in our scripts, or even for other purposes.

PROBLEM

The data in files, of various formats, has to be read so that we can use it in our scripts.

SOLUTION

In a similar manner to the situation with writing data, PowerShell provides a number of ways to read the data in a file, as shown in listing 8.8. The simplest way to read the data is to use `Invoke-Item` ❶. `Invoke-Item` performs the default action on a file. This will open the file in the application associated with the file type, assuming the file associations have been created in the normal manner. In this case, we're opening an Excel spreadsheet using the Excel application. `Invoke-Item` will open .txt files in Notepad and .csv files in Excel. We can open the file in its default application and manually read the data, or work with it within the application, but we can't work with the data using a PowerShell script using this approach.

Listing 8.8 Reading files

```
Invoke-Item F:\Blog\blogstats.xlsx                          ❶

if (Test-Path c:\scripts\testfolder\testfile.txt){
Get-Content -Path c:\scripts\testfolder\testfile.txt}       ❷

Import-Csv -Path c:\scripts\testfolder\testprc.csv |        ❸
Select Name, PeakPagedMemorySize, PeakWorkingSet,
PeakVirtualMemorySize | Format-Table -AutoSize
```

DISCUSSION

`Get-Content` can be used to read the contents of a file ❷. One source of error in our scripts is that the file we're trying to use isn't actually present. We can avoid this by using `Test-Path` as shown. This cmdlet returns `$true` if it finds the file and `$false` if it doesn't. Performing this test allows us to avoid the error.

The best way to read a .csv file is to use `Import-Csv` ❸. The great thing about this approach is that the field names in the header row in the file can be used in the script after the file has been read. This can either be in a `select` statement as here or referred to as `$_.fieldname` when appropriate; for example in a `foreach` cmdlet. Examples of this will appear throughout the book.

When using `Get-Content` or `Import-Csv`, it's often desirable to read the contents of the file into a variable. Script ❸ would become:

```
$data = Import-Csv -Path c:\scripts\testfolder\testprc.csv
$data | Select Name, PeakPagedMemorySize,
PeakWorkingSet, PeakVirtualMemorySize |
Format-Table -AutoSize
```

We could use the same variable in a number of subsequent statements. This approach is worth adopting if you'll be performing a number of actions on the data. It saves the overhead of rereading the file.

These examples show how we can use pieces of PowerShell functionality to read the contents of a file. One other scenario we need to think about is how we search a set of files to find the one that contains the data in which we're interested.

TECHNIQUE 68 Searching files

There are two basic approaches to searching the contents of files. One approach is to explicitly read each file and loop through each record, testing to determine whether it has the content we need. The second is to wrap all of that functionality into a single command. This is the approach we'll take using the `Select-String` cmdlet.

PROBLEM

We need to find a particular piece of text in one or more files from a given set of files.

SOLUTION

`Select-String` provides a search facility similar to that found in the UNIX `grep` command or Windows `findstr`, but it doesn't have a `recurse` parameter to search subfolders. We again get multiple ways to address this problem, which boil down to using regular expressions or a simple text search, as in listing 8.9.

Listing 8.9 Searching files

```
Select-String -Path $pshome\*.ps1xml -Pattern "EventType" -SimpleMatch     ❶

Select-String -Path "c:\scripts\testfolder\*.txt"     ❷
-Pattern "\s{4}Windows\s" -CaseSensitive
```

DISCUSSION

In the first example, we're using a simple text-matching approach One of the cmdlets introduced in PowerShell v2 is `Get-ComputerRestorePoint`, which returns a list of the available restore points on the system. One of the properties returned is `EventType`, which provides information on the event that triggered the creation of the restore point. While experimenting with the cmdlet, I discovered that though just running the cmdlet provides a text description of the `EntryType`, if I used a `select` or `Format-List` with the cmdlet I got an integer returned as the `EntryType`. With a bit more digging, I discovered that this cmdlet is returning WMI objects and that the change in data being returned is a formatting issue. How could I match the integers with the text?

I knew that the formatting data was held in the PowerShell folder in XML files with an extension of .ps1xml. `Select-String` can be used to search those files for all occurrences of `EntryType` ❶. The data returned includes the file name, the line number within the file, and the contents of the line. In this case I'm using a simple string-matching approach.

NOTE `$pshome` is a built-in PowerShell variable that contains the path to the PowerShell installation folder. On a 32-bit machine, it'll be c:\windows\system32\windowspowershell\v1.0. On a 64-bit machine, the answers will be different depending whether the 64- or 32-bit version of PowerShell is used.

The second approach is to use a regular expression as the pattern we're searching against ❷. I must confess that I don't like using regular expressions. They seem to be one of those arcane pieces of technology that I've never had the time to sit down and master. They're incredibly powerful and if you haven't spent time learning how to use them, I'd definitely recommend it as time well spent.

The path to the files is supplied as is the pattern we want to use for searching. The example shows I'm searching for four spaces—`\s` is a space and `{4}` indicates I want four in a row. After that, we're looking for the word *Windows*, followed by another space. The capitalization in Windows is preserved by using the `-CaseSensitive` parameter.

NOTE More on regular expressions can be found in appendix A.

One of the advantages of using `Select-String` for these searches is that we can search text files and XML files. Before we can perform the search, we need to know where our files are located.

TECHNIQUE 69 Searching for files

Most systems will contain thousands of files. How many times have you thought, "I know I put that information in a file, but where's the file?"

PROBLEM

We need to be able to find a file, or set of files, given the name of the file.

SOLUTION

There's a WMI class, `CIM_DataFile`, that's ideal for solving this problem. `Get-WMIObject` is our way to access this data, as in listing 8.10. We can perform our search based on a number of criteria. In the first example we're using the file name. Note that this doesn't include the extension. Our second example uses the extension as the filter criteria. When using the extension, we don't include the dot—we only use the text part of the extension.

These two examples search the whole machine. We can restrict our search to a particular folder, as shown in the third example. The use of \\ as the delimiter on the path is deliberate and necessary when passing paths into WMI cmdlets. It's possible to combine these filters as required using the WQL syntax.

Listing 8.10 Searching for files

```
Get-WmiObject -Class CIM_DataFile -Filter "FileName='sp'"

Get-WmiObject -Class CIM_DataFile -Filter "Extension='txt'" |
Select Name

Get-WmiObject -Class CIM_DataFile `
-Filter "Path='\\scripts\\testfolder\\'" |
 Select Name
```

DISCUSSION

Alternatively we could use `Get-ChildItem`:

```
Get-ChildItem -Path c:\ -Filter "sp*" -Recurse
Get-ChildItem -Path c:\ -Filter "*.txt" -Recurse
Get-ChildItem -Path c:\scripts | select Name
```

Which should we use? It depends! Compare the output of these two commands:

```
Get-WmiObject -Class CIM_DataFile
-Filter "Path='\\scripts\\strings\\'"  | Get-Member

Get-ChildItem -Path c:\scripts\strings | Get-Member
```

The objects that are returned carry different information. It would be worthwhile to explore the differences to determine which will best meet your needs.

This concludes our look at the filesystem. The examples in this section will provide a firm basis for extending your knowledge of how to administer the filesystem using PowerShell. It's time for us to turn our attention the part of Windows administration that many people would prefer to leave alone. The registry has definitely had bad press. Though it's possible to completely wreck your system by being careless when working with the registry, those administrators who take the time to learn how to do it properly end up with a powerful tool at their disposal.

8.4 Registry

The registry is the fundamental store for configuration information on a Windows system. Many applications follow the .NET approach and utilize XML configuration files, but the registry is still heavily used. A lot of the information in the registry would be useful if we could access it. Traditionally, we've used Regedit.exe to work with the registry. PowerShell provides an alternative in the shape of a registry provider.

> **BE VERY, VERY CAREFUL** This is when I should point out that altering the registry can damage your computer setup, which will necessitate a rebuild. I'm assuming that you won't trash your registry just for giggles, but accidents do happen. So, unless you like rebuilding Windows systems, "Let's be careful in there."

We discussed the concept of providers in the early chapters. A provider is a method of exposing a data store as if it were the filesystem. PowerShell supplies a number of providers that expose various data stores as additional PowerShell drives. A truncated list is shown here:

```
PS> Get-PSDrive

Name       Provider      Root
----       --------      ----
Alias      Alias
C          FileSystem    C:\
D          FileSystem    D:\
Env        Environment
```

```
Function    Function
HKCU        Registry      HKEY_CURRENT_USER
HKLM        Registry      HKEY_LOCAL_MACHINE
```

This shows that two drives, HKCU: and HKLM:, are created by PowerShell. We can access these two parts of the Registry as if they were the filesystem, including the use of standard navigation and manipulation commands. These two parts of the registry are the parts we're most likely to need to access, but there are other parts these drives can't reach. In all, there are six sections to the registry:

- HKEY_CURRENT_USER
- HKEY_LOCAL_MACHINE
- HKEY_CLASSES_ROOT
- HKEY_CURRENT_CONFIG
- HKEY_USERS
- HKEY_PERFORMANCE_DATA

Our first task is to find a method of accessing the whole of the registry.

TECHNIQUE 70 Accessing the registry

Before we jump into the registry itself, it's worth spending a bit of time thinking about location. When all we had to think about was our local disks and a few network drives, life was fairly simple. We knew which drive we were on and where our stuff should be. With PowerShell's ability to create drives pointing to other data stores, things are more complicated, as I found out the hard way.

One of the early meetings of the PowerShell User Group involved me giving a demonstration of PowerShell being used to administer Active Directory. I finished the demonstration with the Active Directory provider from PowerShell Community Extensions (PSCX). Even after a couple of years, doing a `dir` through AD is still a cool demonstration and this was the first time I'd shown it. The demonstration was scripted and worked perfectly.

The night before the User Group meeting, Quest released a beta of its Active Directory cmdlets. Now this was beyond cool. There was no way I could leave them out. I created another demonstration script for the cmdlets and tested it until it worked perfectly. By this time, it was late and without thinking it through, or testing, I joined the two demonstration scripts together.

The talk went well. The first part of the demonstration went well. It reached the join and collapsed. The second part completely failed because I'd left my location in the AD provider and forgotten to swap back to the filesystem to pick up the scripts for the second part. The moral of the story is this: always know where you are and don't change the demo.

We can combine a couple of problems here. How can we keep track of where we are and how can we access the registry?

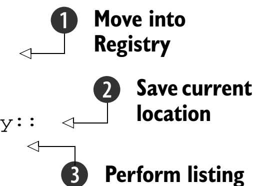
PROBLEM

We know that we can use `cd HKLM:` or `cd HKCU:` to access parts of the registry, but we need to be able to access the whole registry. While we're doing this, it would be a good idea if we could return to our starting point in terms of location.

SOLUTION

We can access the whole registry and keep track of our location in, and among, the various providers by using the `*-Location` cmdlets, as in listing 8.11. The `Set-Location` cmdlet ❶ can be used to change the current location in a provider, such as the filesystem, and between providers or the drives they expose. The `Path` parameter gives the location we want to move to. PowerShell can create aliases of commands to reduce the amount of typing. The alias of `Set-Location` is `cd`, as shown. The use of commands from DOS and Unix shells is one of the things that makes PowerShell easier to learn.

Listing 8.11 Accessing the registry




```
Set-Location -Path Microsoft.PowerShell.Core\registry::     <-- 1 Move into Registry
cd Microsoft.PowerShell.Core\registry::

PS> Push-Location -Path Microsoft.PowerShell.Core\registry::  <-- 2 Save current location
PS> Get-ChildItem                                             <-- 3 Perform listing

    Hive:

SKC  VC Name                           Property
---  -- ----                           --------
  6   0 HKEY_LOCAL_MACHINE             {}
 13   1 HKEY_CURRENT_USER              {Attachment Path}
391   0 HKEY_CLASSES_ROOT              {}
  2   0 HKEY_CURRENT_CONFIG            {}
  6   0 HKEY_USERS                     {}
  0   2 HKEY_PERFORMANCE_DATA          {Global, Costly}

PS> Pop-Location   <-- 4 Return
```

The one odd thing is the path we've constructed to enter the Registry. It's composed of two elements: PSSnapin name and the provider name, followed by two colons, for example `PSSnapin\Provider::`. We can discover the PSSnapin and provider names by using:

```
Get-PSProvider | select Name, PSSnapin
```

PowerShell really can be used to discover how to make PowerShell work.

As I related earlier, not keeping track of location can lead to problems. In the example it wasn't too bad, but a similar lack of awareness once led a colleague to type `del *.*` in the root of the C: drive. Time to rebuild!

PowerShell provides `Push-Location` ❷ to save the current location. The location is pushed onto a stack.

> **NOTE** A *stack* is a store in memory. It's exactly analogous to a stack of plates. We can add a plate to the top of the stack or we can take a plate off the top of the stack. We can't access a plate, or piece of data, that isn't on the top of the stack.

We can use the default stack or we can use the `-StackName` parameter to create, or use, a new stack. `Push-Location` does two jobs, in that we can also use the `-Path` parameter to provide a location to move to (as `Set-Location`) at the same time as putting the current location on the stack.

DISCUSSION

Once we've moved to the desired location, we can perform our tasks, in this case a `Get-ChildItem` to display the registry. ❸ We can return to our starting location by using `Pop-Location` to take the top location off the stack ❹ and set that to be the current location. `Pop-Location` can be used with the default stack or a named stack in the same way as `Push-Location`.

> **RECOMMENDATION** If you're going to be working in other providers extensively, I strongly recommend that you use `Push-Location` and `Pop-Location` in your scripts. This ensures that you always start from a known point and that you can easily return to that point.

Now that we know how to access the registry, it's time to look at reading the data that we find there.

TECHNIQUE 71 Reading registry data

The registry consists of a tree structure of *keys* and *key values*. The PowerShell registry provider treats registry keys in the same way that the filesystem provider treats files—as items. Registry key values are treated as item properties. We need to be aware of this difference and use the appropriate tools.

PROBLEM

The registry provider enables us to access the data stored in the registry as if it were the filesystem. How can we view registry data?

SOLUTION

There is a set of standard cmdlets that work with providers, as shown in listing 8.12. They can be viewed using `Get-Help about_providers`. We start by using `Push-Location` to store our current location and to move to the registry entry we want to view ❶. This task could also have been achieved using `Set-Location` either in one pass or by stepping through the structure of the registry. If a step-by-step approach is used, we can use `Get-ChildItem` at each step to examine the entries. Beware: this can lead to distractions if you spot something interesting!

Listing 8.12 Reading registry data

```
PS> Push-Location `
-Path HKLM:\software\microsoft\powershell\1\shellids     ❶ Move to key
PS> Get-ChildItem                                        ❷ Display key

    Hive: HKEY_LOCAL_MACHINE\software\microsoft\powershell\1\shellids

SKC  VC Name                           Property
---  -- ----                           --------
  0   2 Microsoft.PowerShell           {ExecutionPolicy, Path}

PS> Get-ItemProperty Microsoft.PowerShell    ❸ Display key values

PSPath : Microsoft.PowerShell.Core\Registry::
HKEY_LOCAL_MACHINE\software\microsoft\
powershell\1\shellids\Microsoft.PowerShell
```

```
PSParentPath    : Microsoft.PowerShell.Core\Registry::
HKEY_LOCAL_MACHINE\software\microsoft\powershell\1\shellids

PSChildName     : Microsoft.PowerShell
PSDrive         : HKLM
PSProvider      : Microsoft.PowerShell.Core\Registry
ExecutionPolicy : RemoteSigned
Path            : C:\Windows\system32\WindowsPowerShell\v1.0\powershell.exe

PS> Pop-Location     <-- 4 Return
```

`Get-ChildItem` is used to view the available entries ❷. Note that the key values are returned as properties. In addition to the `*-Item` cmdlets we've discussed, there's a set of `*-ItemProperty` cmdlets we can use to work with properties:

- `Clear-ItemProperty`
- `Copy-ItemProperty`
- `Get-ItemProperty`
- `Move-ItemProperty`
- `New-ItemProperty`
- `Remove-ItemProperty`
- `Rename-ItemProperty`
- `Set-ItemProperty`

DISCUSSION

PowerShell's consistent verb naming conventions means that we need the `Get-ItemProperty` cmdlet to read the key values ❸. Be aware that some other information will be displayed as well as the values in which we're interested. The data is shown in the listing. We could modify the execution policy by accessing through the registry in this manner. It's more efficient to use the `*-ExecutionPolicy` cmdlets. Finally, we use `Pop-Location` ❹ to return to our known starting point.

It's possible to short-circuit this procedure by a single call to `Get-ItemProperty`:

```
Get-ItemProperty `
-Path HKLM:\software\microsoft\powershell\1\ `
shellids\Microsoft.PowerShell
```

We can examine portions of the registry, in the same way we can `dir` through a folder hierarchy, for example:

```
Get-ChildItem -Path HKLM:\software\microsoft\powershell -Recurse
```

The next item on the agenda is learning how to create entries in the registry. This is fun.

TECHNIQUE 72 Creating registry entries

This section deals with performing modifications to the registry. This activity should always be treated with care. Now that we have the obligatory warning out of the way, we can learn how to use our PowerShell knowledge to create registry entries.

PROBLEM

The registry is used to store configuration data as we have seen. There's a periodic need to store configuration data in the registry, perhaps to enable a particular application to work or to prevent an attack on our systems. We need to be able to create and

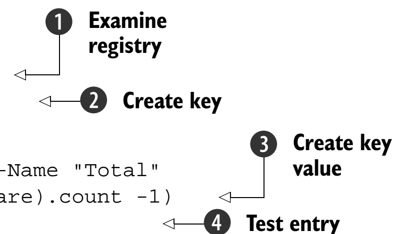
write to registry entries. Please note that this should be a safe example to try, as it doesn't modify live data.

SOLUTION

The `*-Item` and `*-ItemProperty` cmdlets solve this problem for us. We'll create a counter for the number of entries in the software key of the local machine hive for the purposes of this exercise. A key and a key value will have to be created. In this example, we don't move into the registry. All of these tasks can be performed by using the registry drives that are created by the provider, as shown in listing 8.13.

The first task is to examine the Registry key HKLM:\software ❶. This shows a number of keys usually related to software vendors; for example, on my machine there are more than 180 entries in the Microsoft key.

Listing 8.13 Creating registry entries




```
Get-ChildItem -Path HKLM:\software
New-Item -Path HKLM:\software\count
Get-ChildItem -Path HKLM:\software

New-ItemProperty -Path HKLM:\software\count -Name "Total"
   -Value $((Get-ChildItem -Path HKLM:\software).count -1)
Get-ItemProperty -Path HKLM:\software\count
```

`New-Item` is used to create the key ❷. We have to provide the path to the item we're creating. As we can only create keys in the registry, we don't have to provide a type for the item. The new key can be seen when we use `Get-ChildItem` as shown.

> **ALIASES** `Get-ChildItem` is aliased to `dir` and `ls`. In PowerShell v2, use `Get-Alias -Definition Get-ChildItem` to see the aliases. I tend to use `ls` at the PowerShell prompt, as it's less typing! In scripts I use the full cmdlet name.

DISCUSSION

After creating the key, we need to create a value. Key values are manipulated using the `*-ItemProperty` cmdlets. In this case, `New-ItemProperty` is used for creation. The command has to be supplied with the path to the key, a name for the item, and a value ❸. We're using the count of items in the software key minus one to account for the count key itself as the value.

`Get-ItemProperty` is used to examine the value ❹ as we've previously seen. The last activity we need to consider is how to change registry values and how to delete registry keys.

TECHNIQUE 73 Managing registry data

Managing data is often described by using the acronym *CRUD* (not a reflection on the quality of the data) which stands for *Create, Read, Update, and Delete.* It would be possible to do this manually using Regedit.exe, but scripting the solution using PowerShell is a better alternative.

PROBLEM

All data follows a lifecycle and registry keys are no exception. Once the keys been created, we need to be able to manage them in a safe manner.

SOLUTION

We've seen how create and read the data; now we need to discover how perform the update and deletion. We will be using the `*-ItemProperty` and `*-Item` cmdlets, as shown in listing 8.14. The verb *Set* is used to perform changes, and *Remove* when we need to delete.

Listing 8.14 Managing registry data

```
Set-ItemProperty `
-Path HKLM:\software\count -Name "Total" -Value 67                 ❶
Get-ItemProperty -Path HKLM:\software\count                        ❷

Remove-ItemProperty -Path HKLM:\software\count -Name "Total"       ❸ Delete value
Remove-Item -Path HKLM:\software\count                             ❹ Delete key
Get-ChildItem -Path HKLM:\software                                 ❺
```

Changing a registry key value is achieved by using `Set-ItemProperty` ❶. In the same way as when we created the key value, we supply the path (key) the name and the actual value. The change can be verified using `Get-ItemProperty` ❷.

The functionality to wrap registry changes inside transactions was introduced in PowerShell v2. This will provide another level of protection to Registry data.

DISCUSSION

Deleting data from the registry follows a reverse path to creation. The key value is deleted first using `Remove-ItemProperty` ❸. This cmdlet needs to know the path to the key and the name of the value to remove.

> **EVENT LOGS** One thing to think about is creating an event log to record when scripts are run and what changes are performed by the script. This creates an audit trail for your administration, which allows you to prove what you've done, and possibly prove your script didn't do something wrong. Creating and writing to event logs is covered in the next section.

The key itself is removed using `Remove-Item` ❹. A final check can be performed to verify that the key has been removed ❺. These commands can be executed at the PowerShell prompt or within a script.

Working with the filesystem, services, processes, and the registry are all necessary skills for the administrator. At some stage, we need to start investigating what's happening on our systems. The event logs are used to record this information. The next, and last, section in this chapter shows how PowerShell can be used to interrogate and administer the event logs.

8.5 Event logs

Event logs are used by Windows and applications for recording events that occur on the system. The information that's recorded may be of several forms:

- `Error`
- `Warning`
- `Information`
- `SuccessAudit`
- `FailureAudit`

There are a number of standard event logs, with more being created as applications are installed or additional functionality is installed. Event logs are the first port of call for an administrator when troubleshooting problems. I've lost track of the number of times looking at the event logs has enabled me to solve a problem. This includes issues from Active Directory replication not working to cluster nodes refusing to failover. If I were to give a new administrator one tip, it would be learn how to discover information in the event logs: "Learn PowerShell" wouldn't be a tip…it would be an order!

We can use the event viewer to read the logs. It's possible to filter the view. Using PowerShell, we can interrogate and search the logs based on any of the information in the logs. PowerShell v1 only allowed us to read the event logs. If we wanted to do anything else, we had to write scripts as in listings 8.16 through 8.19. PowerShell v2 introduces a number of new cmdlets to work with event logs:

```
PS> Get-Command *eventlog

CommandType     Name
-----------     ----
Cmdlet          Clear-EventLog
Cmdlet          Get-EventLog
Cmdlet          Limit-EventLog
Cmdlet          New-EventLog
Cmdlet          Remove-EventLog
Cmdlet          Show-EventLog
Cmdlet          Write-EventLog
```

We'll see examples of using these as alternatives to the scripts. We can export the information in the logs, possibly for importing into a database. A further possibility is to create our own logs, for instance to record the use of our production scripts. The user running the script could be recorded as well as the actual script used.

PowerShell can be used to configure the event logs. WMI or the `*-EventLog` cmdlets in PowerShell v2 can be used to manage the event logs of remote servers. There may be a performance issue if attempting to manage a large number of servers in a single script.

Before we do any of this, we have to learn to read the logs.

TECHNIQUE 74 Reading event logs

Before we can read the logs, we need to discover what logs are available on the system. PowerShell supplies a cmdlet `Get-Eventlog` to read the logs. This can also be used to discover the event logs on the system:

```
Get-Eventlog -List
```

This will return a lot of useful information, as shown in figure 8.2.

```
PS> Get-EventLog -List

  Max(K) Retain OverflowAction        Entries Name
  ------ ------ --------------        ------- ----
  25,600      0 OverwriteAsNeeded       3,999 Application
  15,168      0 OverwriteAsNeeded           0 DFS Replication
  20,480      0 OverwriteAsNeeded           0 HardwareEvents
     512      7 OverwriteOlder              0 Internet Explorer
  20,480      0 OverwriteAsNeeded           0 Key Management Service
  16,384      0 OverwriteAsNeeded           0 ODiag
  16,384      0 OverwriteAsNeeded         512 OSession
     512      7 OverwriteOlder              0 Scripts
  20,480      0 OverwriteAsNeeded       6,041 Security
  20,480      0 OverwriteAsNeeded      21,097 System
  15,360      0 OverwriteAsNeeded       1,510 Windows PowerShell
```

Figure 8.2 Discovering the event logs. The maximum size in kilobytes and the number of entries supply information regarding log usage. The retention days and overflow action determine what happens if the log becomes full.

The default information includes the number of entries in the log, the maximum size of the logs, and what the logs are configured to do when they become full. This data was taken from the laptop I normally use. These events had built up in less than a month of normal usage. Note how many entries there are in the system log.

> **NOTE** `Get-EventLog` can't read the new style event logs introduced with Windows Vista and Windows Server 2008. We'll see how to work with these logs at the end of this section.

Having found the logs, we can now discover how to read them.

PROBLEM

There's information in the event logs that we need to retrieve. How can we do achieve this?

SOLUTION

We've already seen that we can use `Get-EventLog` to discover the logs on the system. We can also use it to read the data in the event logs, as in listing 8.15. We can retrieve the contents of a particular log ❶ by using its name with the `-LogName` parameter. It has to be the name of the log rather than the display name. Wildcards aren't permitted. This will display all of the entries in the particular log. In the case of the system log, this could be many thousands of entries. This will take a long time to scroll up the screen. We can restrict the number of entries returned by using the `-newest` parameter, or the `-after` and `-before` parameters in version 2 to restrict the returned entries by date. An integer value is supplied to the parameter and the cmdlet will only return that number of entries, starting with the most recent. Using this parameter can be useful if you want to check that an event has just happened.

> **REQUIRED PARAMETER** The `logname` parameter is required. This means that if you don't supply it, PowerShell will prompt you for the value. Many cmdlets have required parameters. The help file for a cmdlet will indicate which parameters are required.

If we're only interested in a particular event, we can filter on `EventId` ❷. This immediately restricts the amount of data that's returned to more manageable proportions.

One problem with using the `EventId` as a filter parameter is that we have to know what the event IDs mean. Many event IDs are documented on the Microsoft and other web sites.

Listing 8.15 Reading event logs

```
Get-EventLog -LogName System                                     <-- 1 Read all

Get-EventLog -LogName System | Where {$_.EventId -eq 7036}       <-- 2 Filtering data

Get-EventLog -LogName System |
Where {$_.Timewritten -gt ((Get-Date).Adddays(-2))}              <-- 3 Filtering by time

Get-Eventlog security |
where {$_.TimeWritten -gt (get-date).AddDays(-7)
-and $_.TimeWritten -lt (get-date) ).AddDays(-2)  }
| Sort EventId | Group EventId                                   <-- 4 Grouping events
```

The date and time when an event occurs (`TimeGenerated`) are written to the log. The time the entry was written is also recorded. On a busy system, there may be more than a slight difference between the two. We can filter on time ❶. Looking at the system log again, we compare the `TimeWritten` property against a date. In this case, we take the current date and subtract two days. Note that we have to use `AddDays()` but supply a negative number. It would've been simpler if we had methods to perform a subtraction. Don't try to define a date that you'll then subtract from the current date. You'll end up with a `TimeSpan` object, which won't work in this context.

It may sometimes be useful to group the events so we can see how many events of a particular type are happening. In particular, we may want to do this against the security log. For instance, if we see a large number of failed logins or failed attempts to access a particular file, it may be a warning that an attack is being mounted against the system. In this case, we read the security log ❹. We perform a filter to restrict the data to a particular time period. The example shows us reading data between seven and two days old. The entries are sorted on event ID and then the data is grouped (`group` is an alias for `Group-Object`).

DISCUSSION

PowerShell must be started with elevated privileges to access the security logs. In Windows Vista/2008 and above when UAC is turned on, this means using Run as Administrator, and logging on using an account with administrator privileges on XP/2003.

Having discovered that we can read the logs, is there a way that we can copy the data to another file for further analysis?

TECHNIQUE 75 Exporting logs

There's often a requirement to preserve the information in event logs. Files can be created to store the data. These files can either be the final home of the data, or they could be used as an intermediary stage to loading the data into a database.

PROBLEM

The data in the event logs has to be exported to CSV files. The file names have to include the date so that we can easily ascertain the period covered by the events in the file.

SOLUTION

We've seen that the Get-EventLog cmdlet can be used to read the contents of the event log. This can be combined with the Export-Csv cmdlet to produce the required file, as in listing 8.16.

Listing 8.16 Copy event logs

```
$date = get-date                                            <-- 1 Current date

if ($date.Month -le 9) {
    $fname = $date.Year.ToString() + "0" +
$date.Month.ToString() + $date.Day.ToString() +
"_security.txt"
     }
                                                            <-- 2 Create file name
else {
    $fname = $date.Year.ToString() + $date.Month.ToString() +
$date.Day.ToString() + "_security.txt"
     }                                                          3 Export security log

get-eventlog security | export-csv $fname  -noTypeInformation  <--

$fname = $fname -replace "security", "application"           Export application log 4
get-eventlog application |
export-csv $fname   -NoTypeInformation    . . . . . . . . . . . . .   <--

$fname = $fname -replace "application", "system"
get-eventlog system | export-csv $fname  -noTypeInformation  <-- 5 Export system log
```

DISCUSSION

This script can be used to create a copy of the data in the three main event logs—application, security, and system. Remember that PowerShell will have to be running with elevated privileges to access the security log.

The script starts by retrieving the current date ❶. Any time you want to manipulate the date information, it's easier to create a variable that can hold the date. We can use the date information to create the file name ❷. The year, month, and day are concatenated, in that order, with the name of the log and a file extension. There are two variations on the way the file name is produced, depending on the number of digits in the month. A leading zero is added for months with only a single digit—January to September. The file names can be sorted on the date part when created in this style.

We could use the -f operator to format the filename for us:

```
$fname = "{0}{1:00}{2:00}_security.txt" -f `
$date.Year, $date.Month, $date.Day
```

The first part, {0}{1:00}{2:00}_security.txt, defines three fields that will be filled by the data on the right side of the -f operator, plus the static part of the file name. The first field takes the year, which will be four digits, whereas the month and day are substituted into the second and third fields respectively. These two fields are defined as being two digits wide, so a leading zero is automatically appended for a day or month value of 9 or less. These formatted strings look a bit scary, but it's worth getting to know how to use them, as they can save a lot of effort.

The only thing left to do is to get the data from the event log ❸ and write it out to the file. We use the `-noTypeInformation` parameter on `Export-Csv` to avoid writing the .NET type information into the first line of the file. This technique can be applied to other logs, including the application ❹ and system logs ❺ as shown. The `-replace` operator can be used to modify the file name to reflect the correct log. The file name is a string, so we could use `$file.Replace()` instead.

Having explored how we can read from the event log, it's now time to look at writing to an event log. Before we can write to a log, we need to be able to create a log.

TECHNIQUE 76 Creating an event log

Windows supplies a number of event logs by default and will create others depending on the applications and functionality installed on the system. The generic system and application logs would seem ideal to use for recording events from our scripts. If a specialized log is created just for scripting events that we create, we can have much greater control of the data. The data will be easier to search, as it's restricted to scripting events.

PROBLEM

We've decided to create an event log specifically for events from our scripts.

SOLUTION

This problem requires using some, but not a lot, of .NET code. Just one little line. This problem can be solved using a static method of the `System.Diagnostics.EventLog` class. Static methods were explained in chapter 3; to recap they're methods that can be used without creating an object. The `CreateEventSource` method is given two parameters, as shown in listing 8.17.

Listing 8.17 Create an event log

```
[System.Diagnostics.EventLog]::CreateEventSource("PSscripts","Scripts")
```

The first parameter is the event source. Sources are registered against a particular event log. They're in effect a handle for the particular log. Figure 8.3 shows the central part of the event viewer console. Our Scripts log is the selected log and it contains a single event. The log name and source can be seen in the lower part of the panel.

The second parameter is the name of the log. `CreateEventSource` can be used to create a source for an existing event log, or as in this case, it'll create the event log if it doesn't already exist. .NET code will rarely get any simpler.

DISCUSSION

The PowerShell v2 version of this is just as simple:

```
New-EventLog -LogName Scripts -Source PSscripts
```

It has parameters for the log name and source as seen previously. Having created an event log, we need to learn how to populate it. This means we have to use a bit more .NET code, but it's straightforward, as we shall see.

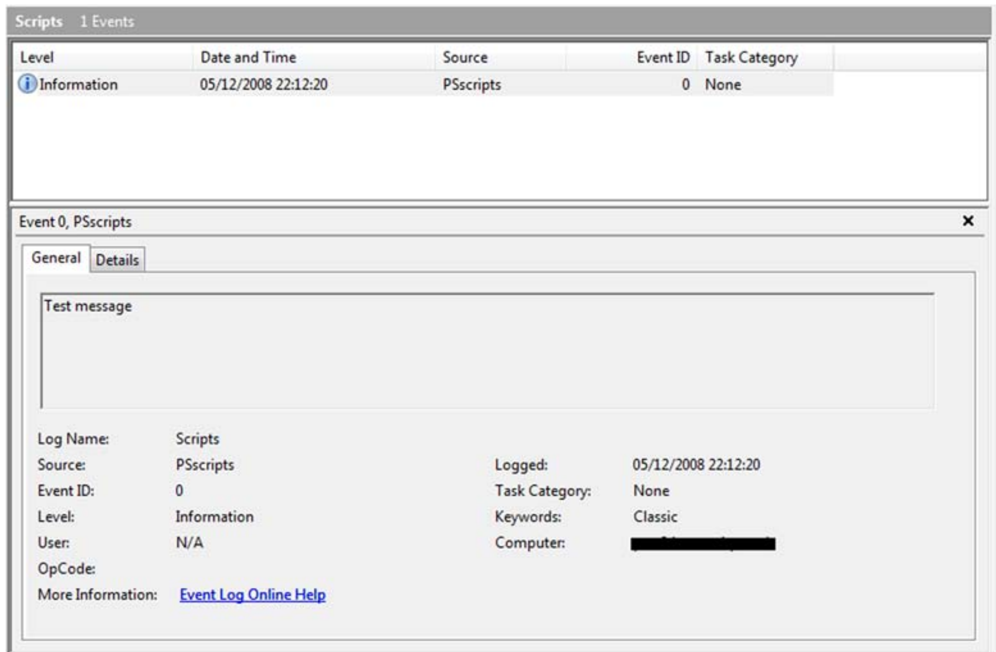


Figure 8.3 Event log example with source. The computer name is blacked out to obscure the internal domain for security reasons.

TECHNIQUE 77 Creating events

Having seen how to create an event log, we now need to think about writing events into the log. An empty log isn't going to do us much good. Ideally, this should be done in a way that allows the functionality to be used from any of our scripts without having to copy the code into every script.

PROBLEM

In order to keep track of which scripts are run when, we want to be able to write an entry into the event log as the script is executing. So that we get the most out of the time we spend developing the functionality, we also want to be able to call at any time during the execution of the script. There are much more interesting things to do than write variations on the same code. Write once, run many times.

SOLUTION

We can solve this by creating a function using the `System.Diagnostics.EventLog` class we saw in the previous example. Our problem statement said that we needed to be able to use this from multiple scripts without putting the code into every script. The best way to achieve this is to create a function that we load into memory when we start PowerShell. How do we do this? We either put the function into our profile or use a dot-sourced call to a script containing the function into the profile, as in listing 8.18. In both cases we have it loaded and ready to use. Now all we've got to do is write it!

Listing 8.18 Write to an event log

```
function Write-EventLog
{
param([string]$msg = "Default Message", [string]$type="Information")
$log = New-Object System.Diagnostics.EventLog
$log.set_log("Scripts")
$log.set_source("PSscripts")

$log.WriteEntry($msg,$type)
}
```

The function takes two parameters—a message and the event type. Event types can one of several types, including information, warning, and error. If we want to see the available types, we can use:

```
[enum]::GetNames([System.Diagnostics.EventLogEntryType])
```

This is explained in chapter 3. As a quick recap, `[enum]` is a shorthand for the `System.Enum` class. The `Enum` class is the base class for enumerations, which are closed lists. Enums are often used for parameters where there's a discrete set of possibilities. The contents of a particular enumeration can be found in the .NET documentation.

The bulk of the function is taken up by creating an object using the `System.DiagnosticsEventLog` class. We then set the log and the source we want to use. The final action is to write the event message and the event type to the log.

We can use our function from the PowerShell prompt or from within a script by a simple call to the function:

```
Write-EventLog "Testing Function use" "Information"
```

The first parameter is the message and the second is the event type.

DISCUSSION

We can test that the event has been written to the log by using:

```
Get-EventLog -LogName Scripts
```

This will show the log entries including the messages. The PowerShell v2 version is:

```
Write-EventLog -LogName Scripts -Source PSscripts `
-EntryType "Information" `
-Message "Testing the event log" -EventId  1
```

We have to provide an `EventId` when using this cmdlet. It'd be worthwhile documenting the IDs to be used with the event log when it's created. After learning how to create and write to event logs, it's now time to learn how to manage them.

TECHNIQUE 78 Managing event logs

Administering computer systems involves a number of activities, including configuring the system components and creating backups. In this section, we'll look at both of these activities in relation to the event logs on our systems.

> **NOTE** If you want to work with the Security event log, you'll need to be working with elevated privileges.

The scripts in this section are WMI-based and written to work against the local machine. Using the `-computername` parameter, we can manage event logs on remote systems.

PROBLEM

We need to be able to configure the event log and be able to back it up.

SOLUTION

The WMI class `Win32_NTEventLogFile` is available to us for performing these actions. There are two main parameters we think about configuring for an event log file:

- The maximum file size (`MaxFileSize`)
- Action to take when the log is full (`OverWritePolicy`)

Event log configuration can be achieved with the `Win32_NTEventLogFile` WMI class, as in listing 8.19. We start by creating a WMI object for the log ❶. In this case, we're configuring the Application log. The file size is set using the `MaxFileSize` parameter. We have to use the `psbase` construction to get to the underlying object so we can call the `Put()` method. This saves the configuration change. `Win32_NTEventLogFile` can only be used against the "old-style" event logs in Windows 2003 and earlier. It won't work against the new-style logs in Vista/Windows 2008.

Listing 8.19 Managing event logs

```
$applog = Get-WmiObject -Class Win32_NTEventLogFile        <-- ❶ Set max size
-Filter "LogFileName = 'Application'"
$applog.MaxFileSize = 26214400
$applog.psbase.Put()

$log = Get-WmiObject -Class Win32_NTEventLogFile        <-- ❷ Backup event log
-Filter  "LogFileName = 'Application'"
$ret = $log.BackupEventLog("c:\test\applog.evt")
if ($ret.returnvalue -eq 0){$log.ClearEventLog()}
else {Write-Host "could not back up log file"}

Get-WmiObject -Class Win32_NTEventLogFile |        <-- ❸ Size triggered backup
Where {$_.FileSize -gt 10MB} | Foreach {
    $file = "c:\test\" + $_.LogFileName + ".evt"
    $_.BackupEventLog($file)
    $_.ClearEventLog()
}

$date = Get-Date        <-- ❹ Records triggered backup
Get-WmiObject -Class Win32_NTEventLogFile |
Where {$_.NumberofRecords -gt 5} | Foreach {
    $file = "c:\test\" + $_.LogFileName + "_{0}_{1}_{2}.evt" -f
           $date.Year, $date.Month, $date.Day
    $_.BackupEventLog($file)
    $_.ClearEventLog()
}
```

`Win32_NTEventLogFile` can also be used to back up the event logs. If we want to back up a single log, we can apply a filter to restrict ourselves to the particular log ❷. We can use the `BackupEventLog()` method to perform the backup. The backup file is the only parameter. If the return code is zero, we can then clear the event log.

It may be desirable to back up and clear the event log depending on a trigger. This could be when the log reaches a particular size ❸ or when it has more than a preset number of records ❹. All of the logs are retrieved via WMI. The test is applied, and for each log that passes the test, a backup file is created and then the logs are cleared.

DISCUSSION

PowerShell v2 cmdlets allow us to configure the event logs and clear the event logs:

```
Limit-EventLog -LogName scripts -MaximumSize 25MB
Clear-EventLog -LogName scripts
```

We don't get a cmdlet to perform the backup.

> **Windows 2008 and Vista logs**
>
> A new type of event log was introduced in Windows Server 2008 and Windows Vista. At the moment, all we can do is retrieve records from these logs using `Get-WinEvent` in PowerShell v2. This is used in a similar manner to `Get-EventLog`, which we have already seen in detail.
>
> This completes our look at event logs. We've covered most if not all of the tasks that administrators need to perform on them. You should be well placed to administer your event logs using PowerShell after reading this section.

8.6 *Summary*

We've looked at a number of facets of server administration in this chapter. The services and applications need to be managed. This can involve discovering their status as well as creating or terminating processes. These techniques will be useful when considering specific applications in later chapters. Think about the situation where you can write a script that tests the status of the required services running on a remote machine and then starts them if they aren't running.

The filesystem involves us in a number of activities. Some activities are simple such as creating files and folders. Other activities are much more involved, in that we have to read or write the contents of the files. At some stage, we'll need to be able to search the contents of files for particular pieces of text or possibly search through the filesystem for a particular file. How many times have you had a user come along and say "I can't find my file. It's called xyz.txt. Can you find it for me?" Now you can.

The registry holds configuration data that we need to be able to read and possibly change. Accessing and modifying the registry can be regarded as a dangerous occupation but the tools in PowerShell enable us to perform these tasks in a safe and controlled manner.

The activities we've been performing when administering the servers are recorded in the event logs. We looked at reading the contents of the event logs so we can diagnose issues. Event logs are also objects that we need to administer. We can change the settings of an existing log and create new logs. Backing up the event logs gives us a way to keep the records for future use.

A book this size can't exhaustively cover every variation involved in administering the server. The examples in this chapter form a firm foundation for undertaking the core administrative tasks on a Windows server. Now that we know how to do that, we can start looking at the applications running on the server, starting with DNS.

WINDOWS ADMINISTRATION

PowerShell IN PRACTICE

Richard Siddaway

PowerShell is a powerful scripting language that lets you automate Windows processes you now manage by hand. It will make you a better administrator.

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.

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.

Richard Siddaway is an IT Architect with 20 years of experience as a server administrator, support engineer, DBA, and architect. He is a PowerShell MVP and founder of the UK PowerShell User Group

"A definitive source."
—Wolfgang Blass
Microsoft Germany

"A must read!"
—Peter Johnson
Unisys Corp.

"A new perspective on PowerShell!"
—Andrew Tearle
Thoughtware N.Z.

"Real-world examples... in a language you can understand."
—Marco Shaw
Microsoft MVP

For online access to the author, and a free ebook for owners of this book, go to www.manning.com/PowerShellinPractice

ISBN 13: 978-1-935182-00-9
ISBN 10: 1-935182-00-5
54999
9 781935 182009

MANNING $49.99 / Can $62.99 [INCLUDING eBOOK]