EXPERT INSIGHT

Windows Server Automation with PowerShell Cookbook

Powerful ways to automate and manage Windows administrative tasks

Foreword by:
Jeffrey Snover
Microsoft Technical Fellow

Fourth Edition

Thomas Lee

Packt>

Windows Server Automation with PowerShell Cookbook

Fourth Edition

Powerful ways to automate and manage Windows administrative tasks

Thomas Lee

Packt

BIRMINGHAM—MUMBAI

Windows Server Automation with PowerShell Cookbook

Fourth Edition

Producer: Caitlin Meadows

Acquisition Editor – Peer Reviews: Divya Mudaliar

Project Editor: Parvathy Nair

Content Development Editor: Lucy Wan

Copy Editor: Safis Editor

Technical Editor: Aditya Sawant

Proofreader: Safis Editor

Indexer: Rekha Nair

Presentation Designer: Ganesh Bhadwalkar

First published: March 2013

Second edition: September 2017

Third edition: February 2019

Fourth edition: July 2021

Production reference: 1290721

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-80056-845-7

www.packt.com

Foreword

I was excited when Thomas Lee told me he was writing a new book.

I was even more excited when he told me that it was about Windows Server 2022.

I was even more excited when he told me that it was using PowerShell 7.

Thomas Lee has been a fixture of the PowerShell community starting from the moment he disrupted my first presentation about it, shouting wildly: "*Take my money! I'll buy it now!*". He has watched and participated in every step of the PowerShell journey and has been an effective and articulate advocate for the user community. Thomas effectively communicates the needs of the community to Windows Server and PowerShell teams. Thomas also effectively communicates the new advances of Windows Server and PowerShell to the community. He lets them know both what is possible and how to achieve it.

As part of his community advocacy, Thomas was forthright about the challenges that Windows administrators were having with PowerShell Core. PowerShell Core was a departure from Windows PowerShell. It moved from .NET Framework to .NET Core. The core benefit of .NET Core was that it was cross-platform – it ran on Linux and it was open source. The core issue with .NET Core was that it did not have all the features of .NET Framework. As such, PowerShell Core did not have all the features of Windows PowerShell. Because it did not have all the features, several partner teams did not support it.

You know how some teenagers experience a growth spurt and have an awkward period? Core PowerShell was our awkward period. The ability to run anywhere was awesome, there were a bunch of new features, it was much faster, but the lack of critical mass support meant that it wasn't for everyone and every scenario.

At the end of the day, no one cares about how you make the sausage – they just care about how it tastes. At some point, someone will tell the Herculean story of what was involved in migrating to .NET Core. It was massive and required a complete overhaul of our engineering and test systems. But that overhaul delivered us the gift of agility. And with that agility came increased customer focus and responsiveness. And because PowerShell was open source, the community was able to adapt the product to their needs and go even faster. I remember a video of Jeff Woolsey demonstrating a new capability and Thomas Lee tweeting out: "*This is awesome. It kind of feels like PowerShell V1 all over again. SUPER excited*." I responded, saying: "*Yes, but with much better velocity*."

The virtuous cycle of agility, customer focus, and velocity allowed us to make rapid progress and close the gap in features with Windows PowerShell so that by the time we shipped V7, we decided to change the name to just "PowerShell". No qualifiers. No caveats. Just PowerShell.

Now to be completely forthright, there are still some things that Windows PowerShell does that PowerShell V7 does not. Based upon low usage and community feedback, we dropped a set of features like Workflow. I was confident that these would not get in the way of managing Windows Server, but I always knew that Thomas Lee would be a reality check on that.

Given that Thomas uses PowerShell V7 as the basis for this book on managing Windows Server, I can rest easy.

Before I go, let me reiterate PowerShell's focus on making customers successful. We recently got feedback that we had become a bottleneck and were slowing down community contributions to the code base. We modified our governance, restructured the code with new projects to allow more parallel development, and added more people from the community to the process. We anticipate that that will further accelerate the agility, customer focus, and velocity cycle.

PowerShell is all about making you successful.

Have fun scripting!

Jeffrey Snover

Microsoft Technical Fellow

June 2021

Contributors

About the author

Thomas Lee is a consultant/trainer/writer from England and has been in the IT business since the late 1960s. After graduating from Carnegie Mellon University, Thomas joined ComShare, where he was a systems programmer building the Commander II time-sharing operating system, a forerunner of today's cloud computing paradigm. In the mid-1970s, he moved to ICL to work on the VME/K operating system. After a sabbatical in 1980/81, he joined what is today known as Accenture, leaving in 1988 to run his own consulting and training business, which is still active today.

Thomas holds numerous Microsoft certifications, including MCSE (one of the first in the world) and later versions, MCT (25 years), and was awarded Microsoft's MVP award 17 times. He lives today in a cottage in the English countryside with his family, a nice wine cellar, and a huge collection of live recordings by The Grateful Dead and The Jerry Garcia band.

I'd first like to thank Jeffrey Snover of Microsoft for the invention of PowerShell. I was lucky enough to be in the room the very first time he presented what was then called Monad. His enthusiasm was infectious, and nearly 20 years later I am still excited. And, of course, no book on PowerShell would be complete without acknowledging the great work done by the PowerShell team, including Joey Aiello, Steve Lee, Jim Truher, and many more. The team has consistently listened to the community, found ways to make PowerShell better, and has delivered release after release of solid, well written code.

When you write a book, there is a large publishing team behind you without whom this book would just be a dream. This is even more relevant as this book was written during the Covid19 pandemic. Coping with the results of that has been a real challenge for all of us. A huge thank you has to go to the Packt team: Caitlin Meadows (a truly outstanding editor) and all the folks she brought to the party. Thanks too to Parvathy Nair and Lucy Wan, two dedicated editors who turned my badly written gibberish into decent technical English, and to Aditya Sawant, who helped with proofs.

Thanks to our most excellent tech reviewer, Joshua King. His reviews were always helpful and useful. He uncovered issues I had initially missed, and his many suggestions contributed to a better book. I look forward to working with him (and this whole team) again.

As each recipe evolved, I would sometimes hit problems. I got a lot of help from the Spiceworks community. Their PowerShell forum is a great source of information and encouragement. If you have problems with PowerShell, this is a great place to get a solution.

And finally, I have to thank my wonderful wife, Susan, and our amazing daughter, Rebecca. My wife has been patient as things progressed, she put up with my bad moods when progress was not as smooth as desirable, and kept me sane when all around me was craziness. And my daughter's smile could not help but brighten even the darkest days.

About the reviewer

Josh King is a Microsoft MVP and Infrastructure Operations Engineer at Chocolatey Software. He has a long history working within Windows and VMware environments and has a passion for all things PowerShell and automation.

Josh was an author of *The PowerShell Conference Book Volume 2* and *Volume 3*.

You can find Josh on Twitter, `@WindosNZ`, or his blog at `https://toastit.dev/`.

Table of Contents

Preface

PowerShell was first introduced to the world at the Professional Developers Conference in Los Angeles in 2003 by Jeffrey Snover. Code-named Monad, it represented a complete revolution in management. A white paper written around that time, *The Monad Manifesto* (refer to `http://www.jsnover.com/blog/2011/10/01/monad-manifesto/`), remains an amazing analysis of the problem at the time, that of managing large numbers of Windows systems. A key takeaway is that the GUI does not scale, whereas PowerShell does.

PowerShell has transformed the management of complex, network-based Windows infrastructure, and, increasingly, non-Windows infrastructure. Knowledge of PowerShell and how to get the most from PowerShell is now obligatory for any IT professional. The popular adage continues to be true: learn PowerShell or learn golf.

Windows PowerShell was developed on Windows for Windows administrators. PowerShell 7, the open-source successor, is also available for Mac and most of the more popular Linux distributions. This book, however, concentrates on PowerShell within a Windows environment.

This book takes you through the use of PowerShell in a variety of scenarios, using many of the rich set of features included in Windows Server 2022 and 2019. This preface provides you with an introduction to what is in the book, along with some tips on how to get the most out of it.

Who this book is for

This book is aimed at IT professionals, including system administrators, system engineers, architects, and consultants who need to understand PowerShell 7 to simplify and automate their daily tasks. The recipes in this book have been tested on the latest versions of Windows Server.

What this book covers

Chapter 1, *Installing and Configuring PowerShell 7*, shows you how you can install and configure both PowerShell 7 and VS Code, which replaces the Windows PowerShell **Integrated Scripting Environment** (**ISE**), as well as how to install a new font, Cascadia Code. This chapter also examines the PowerShell 7 environment, including the `PSReadLine` module.

Chapter 2, *Introducing PowerShell 7*, looks at what's new in PowerShell 7. This chapter examines the new features you can use with PowerShell 7, including a number of new operators and improvements in parallel processing. The chapter also looks at how PowerShell 7 formats and manages error messages.

Chapter 3, *Exploring Compatibility with Windows PowerShell*, explores PowerShell 7's compatibility with Windows PowerShell. PowerShell 7 is based on the open-source .NET, which is largely, but not fully, compatible with the older .NET Framework. This means some features of Windows PowerShell do not work natively within PowerShell 7, including many of the Windows PowerShell modules that come with Windows Server. The chapter examines the compatibility mechanism adopted by the PowerShell developers to enable older Windows PowerShell modules to function within PowerShell 7 and to close the gap between what you can do with Windows PowerShell and what you can do with PowerShell 7.

Chapter 4, *Using PowerShell 7 in the Enterprise*, looks at how you can use various PowerShell 7 features that might be more common within larger enterprises. These include the **Remote Server Administration Tools** (**RSAT**), package management and the PowerShell Gallery, and creating a local module repository. The chapter also looks at PowerShell script signing, using shortcuts, and working with archive (`.zip`) files.

Chapter 5, *Exploring .NET*, examines .NET, which provides the foundation for PowerShell 7. The chapter looks at .NET assemblies, classes, and methods, and concludes by demonstrating how you can create simple C#-based PowerShell extensions and full cmdlets.

Chapter 6, *Managing Active Directory*, examines how to install, manage, and leverage **Active Directory**, including installing domains and child domains, managing AD objects, and leveraging Group Policy. The chapter also examines how you can use PowerShell to report on your AD environment.

Chapter 7, *Managing Networking in the Enterprise*, shows you how to manage Windows networking with PowerShell. Networks are today central to almost every organization and this chapter looks at a variety of network-related tasks, including looking at new ways to do old things with PowerShell, setting up DNS, DHCP, and DHCP failover and load balancing.

Chapter 8, *Implementing Enterprise Security*, looks at security aspects within the context of an enterprise environment. The chapter looks at **Just Enough Administration** (**JEA**), which limits the actions an administrator can perform remotely. It also looks at the event log, PowerShell 7's script block logging, setting PowerShell 7-related group policies, and configuring a fine-grained AD password policy. The chapter concludes by looking at the Windows Defender Antivirus product built into Windows Server.

Chapter 9, *Managing Storage*, looks at managing storage in Windows Server, including locally attached devices and Windows Storage Spaces. The chapter also looks at managing Storage Replica, a feature of Windows Server 2022.

Chapter 10, *Managing Shared Data*, examines different ways to share data and manage your shared data with Windows Server and PowerShell. This includes managing NTFS permissions, creating and securing SMB shares, and setting up and using iSCSI. The chapter concludes by looking at **File Server Resource Manager** (**FSRM**), a feature of Windows Server, and managing FSRM quotas, file screening, and reporting.

Chapter 11, *Managing Printing*, shows you how to manage printers, printer queues, and printer drivers as well as how to set up a printer pool. You also examine how to print a test page.

Chapter 12, *Managing Hyper-V*, demonstrates the use of Hyper-V. This chapter shows you how to build and deploy VMs with Hyper-V. This includes nested Hyper-V, and running a Hyper-V VM inside another Hyper-V VM, which is useful for a number of scenarios.

Chapter 13, *Managing Azure*, looks at managing IaaS and PaaS resources in Azure using PowerShell. To test the recipes in this chapter, you need access to Azure. This chapter describes Azure Storage and how to set up a Virtual Machine, an Azure website, and an SMB3 file share.

Chapter 14, *Troubleshooting with PowerShell*, looks at a number of aspects of both reactive and proactive troubleshooting. This includes using the PowerShell script debugger, getting events from the event log, and using the Best Practices Analyzer contained in Windows Server.

Chapter 15, *Managing with Windows Management Instrumentation*, examines **Windows Management Instrumentation** (**WMI**) and enables you to investigate WMI namespaces, classes, and class occurrences. You retrieve information from WMI classes, update WMI using WMI methods, and manage WMI events including WMI permanent eventing.

To get the most out of this book

I designed and wrote this book based on some assumptions and with some constraints. Please read this section to understand how I intended the book to be used and what I have assumed about you. This should help you to get the most from this book.

The first assumption I made in writing this book is that you know the very basics of PowerShell. For that reason, this book is not a PowerShell tutorial. The recipes in this book make use of a wide range of PowerShell features, including WMI, Remoting, AD, and so on, but you need to know the basics of PowerShell. The book was developed using Windows 10 and both Windows Server 2019 and the emerging Windows Server 2022.

The second, related, assumption is that you have a reasonable background in Windows infrastructure, including AD, networking, and storage. The recipes in each chapter provide an overview of the various technologies, and I've tried to provide good links for more information on the topics in this book. The recipes are designed to show you the basics of how to manage aspects of Windows Server and how you might adapt them for your environment.

You start your exploration by installing and configuring PowerShell 7 and VS Code and creating Hyper-V VMs to test out each chapter's recipes. I built and tested the recipes in this book step by step (i.e. not running the entire recipe as a single script file). If you run a recipe as a single step, some of the output may not be what you see here, due to how PowerShell formats objects.

Once you have any recipe working, try to re-factor the recipe's code into your own reusable functions. In some cases, we build simple functions as a guide to richer scripts you could build. Once you have working and useful functions, incorporate them into organizational or personal modules and reuse the code.

As any author knows, writing PowerShell scripts for publication in a book is a layout and production nightmare. To reduce the issues specifically with line width and line wrapping, I have made extensive use of methods that ensure the command line width fits in the chapters in this book without wrapping. Many recipes use hash tables, property spatting, and other devices to ensure that every line of every recipe is 73 characters or less and that there are no unintended line breaks. I hope there are not too many issues with layout!

Many of the cmdlets, commands, and object methods used in this book produce output that may not be all that helpful or useful, particularly in production. Some cmdlets generate output that would fill many pages of this book but with little added value. For this reason, many recipes pipe cmdlet output to `Out-Null`. Feel free to remove this where you want to see more details. I have also adjusted the output in many cases to avoid wasted white space.

Thus, if you test a recipe, you may see that the output is laid out a bit differently, but it should contain the same information. Finally, remember that the specific output you see may be different based on your environment and the specific values you use in each step.

To write this book, I have used a large VM farm consisting of over 12 Windows Server 2022 hosts and Windows 10 clients. My main development host was a well-configured Windows 10 system (96 GB RAM, 2 x 6 core Xeon processors, and fast SSDs). All the hosts used in this book are a combination of some physical hardware (running almost entirely on Windows 10 and a large set of VMs) as described in the recipe.

To assist in writing this book, I created a set of scripts that built the Hyper-V VMs that I used to develop this book. These scripts are published at `https://github.com/doctordns/ReskitBuildScripts`. I have also published some details of the network of VMs created by using these scripts, complete with host names and IP addresses, at the same URL. The full set of VMs, at the end of this writing, took up around 600 GB of storage. Fortunately, storage is cheap! The GitHub repository has more details on the scripts and how to run them. If you have any issues with the scripts, please file an issue on GitHub and I can assist.

PowerShell 7 provides great feature coverage with respect to being able to manage the functions and features of Windows Server 2022 using PowerShell. As with Windows PowerShell, you have considerable flexibility as to what commands you use in your scripts. While PowerShell cmdlets are generally your first choice, in some cases, you need to dip down into .NET, or into WMI, to get to objects, properties, and methods that no existing PowerShell command provides. And if that is not enough, you can develop your own .NET classes and full PowerShell 7 cmdlets.

An important aspect of the recipes in this book is the use of third-party modules obtained from the PowerShell Gallery. There is a rich and vibrant PowerShell community that has created a substantial amount of functionality for you to use. The PowerShell Gallery, a repository provided by Microsoft, enables you to download and use these modules. The `NTFSSecurity` module, for example, makes it simple to manage the **Access Control List** (**ACL**) on NTFS files and folders.

All the code provided in this book has been tested. It worked when I tested it, and it did what it says (at least during the writing stage). I have taken some liberties with respect to the layout and formatting to cater for the book's production and printing process, but you should get the same results. That said, the book production process is very complex and it is possible that errors can creep in during the production stages. So if you find a step in any recipe that fails for you, file an issue on my GitHub repository for this book (see below). For generic issues, please post issues to the Spiceworks PowerShell forum.

In writing the recipes, I have used full cmdlet names with all parameter names spelled out in full. This makes the text a bit longer, but hopefully easier to read and understand.

In writing this book, I set out to create content around a number of features of Windows Server 2022. In order to publish the book, it was necessary to avoid going too deep into every Windows feature. I have had to decide which features (and commands) to show and which to not cover, since every chapter could easily have become a small book. To paraphrase Jeffrey Snover, *to ship is to choose*. I hope I chose well.

Some recipes in this book rely on you having run other recipes in prior chapters. These related recipes worked well when we wrote and tested them and hopefully work for you as well. If you have problems with any of the recipes, then raise issues on my GitHub repository.

Finally, there is a fine line between PowerShell and a Windows feature. To use PowerShell to manage a Windows feature, you need to understand the Windows feature itself. The chapters provide short overviews of the Windows Server features and I have provided links to help you get more information. And as ever, Bing and Google are your best friends.

Download the example code files and VM build scripts

I have published every recipe (and a bit more) to a public GitHub repository: `https://github.com/PacktPublishing/Windows-Server-Automation-with-PowerShell-7.1-Cookbook-Fourth-Edition`. There is a `README.md` file at the top of the repository introducing what is in the repo. Within the `scripts` folder, you can find all the recipes within this book.

Should you find any issues with the recipes in this repository, please file an issue at `https://github.com/PacktPublishing/Windows-Server-Automation-with-PowerShell-7.1-Cookbook-Fourth-Edition/issues` and I can assist.

This book makes use of a farm of Hyper-V VMs that you can use to replicate the recipes. I have created a set of VM build scripts you can download from my GitHub repository at `https://github.com/doctordns/ReskitBuildScripts`. To use these scripts, you need to obtain an ISO image of Windows Server 2022 to serve as a base image. You can get that from `https://www.microsoft.com/evalcenter/evaluate-windows-server-2022-preview/` or via your Visual Studio or other subscription. You use the ISO image to create a "reference disk" – then for each VM, the scripts create a unique VM based on the reference disk. This VM is a brand-new install based on unattended XML used to pre-configure each VM. In theory, you could create these VMs in Azure, but I have not tested that. You can read more about how to use the build scripts from the `README.md` document in the repository.

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://static.packt-cdn.com/downloads/9781800568457_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example; "To establish a link shortcut, you can use the `Wscript.Shell` COM object."

A block of code is set as follows:

```
"On Host [$(hostname)]"
"Total features available      [{0}]"  -f $Features.count
"Total features installed      [{0}]"  -f $FeaturesI.count
"Total RSAT features available [{0}]"  -f $RSATF.count
"Total RSAT features installed [{0}]"  -f $RSATFI.count
```

Bold: Indicates a new term, an important word, or words that you see on the screen, for example, in menus or dialog boxes, also appear in the text like this. For example: "Signing a script is simple once you have a digital certificate issued by a **Certificate Authority** (**CA**)."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com`, and mention the book's title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `http://www.packtpub.com/submit-errata`, select your book, click on the Errata Submission Form link, and enter the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `http://authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packtpub.com`.

Share your thoughts

Once you've read *Windows Server Automation with PowerShell Cookbook, Fourth Edition*, we'd love to hear your thoughts! Please `click here to go straight to the Amazon review page` for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

1

Installing and Configuring PowerShell 7

This chapter covers the following recipes:

- Installing PowerShell 7
- Using the PowerShell 7 console
- Exploring PowerShell 7 installation artifacts
- Building PowerShell 7 profile files
- Installing VS Code
- Installing the Cascadia Code font
- Exploring PSReadLine

Introduction

PowerShell 7 represents the latest step in the development of PowerShell. PowerShell, first introduced to the public in 2003, was released formally as Windows PowerShell v1 in 2006. Over the next decade, Microsoft released multiple versions, ending with PowerShell 5.1. During the development of Windows PowerShell, the product moved from being an add-in to Windows to an integrated feature. Microsoft plans to support Windows PowerShell 5.1 for a long time, but no new features are likely.

The PowerShell development team began working on an open-source version of PowerShell based on the open-source version of .NET Core. The first three versions, PowerShell Core 6.0, 6.1, and 6.2, represented a proof of concept – you could run the core functions and features of PowerShell across the Windows, Mac, and Linux platforms. But they were quite limited in terms of supporting the rich needs of the IT pro community.

With the release of PowerShell 7.0 came improved parity with Windows PowerShell. There were a few modules that did not work with PowerShell 7, and a few more that work via a compatibility mechanism described in *Chapter 3*, *Exploring Compatibility with Windows PowerShell*. PowerShell 7.0 shipped in 2019 and has been followed by version 7.1. This book uses the term "PowerShell 7" to include both PowerShell 7.0 and 7.1.

Once you have installed PowerShell 7, you can run it and use it just as you used the Windows PowerShell console. The command you run to start PowerShell 7 is now `pwsh.exe` (versus `powershell.exe` for Windows PowerShell). PowerShell 7 also uses different profile file locations from Windows PowerShell. You can customize your PowerShell 7 profiles to make use of the new PowerShell 7 features. You can also use different profile files for Windows PowerShell and PowerShell 7.

The Windows PowerShell **Integrated Scripting Environment** (**ISE**) is a tool you use with PowerShell. The ISE, however, is not supported with PowerShell 7. To replace it, use **Visual Studio Code** (**VS Code**), an open-source editing project that provides all the features of the ISE and a great deal more.

Microsoft also developed a new font, Cascadia Code, to coincide with the launch of VS Code. This font is a nice improvement over Courier or other mono-width fonts. All screenshots of working code in this book use this new font.

`PSReadLine` is a PowerShell module designed to provide color-coding of PowerShell scripts in the PowerShell 7 console. The module, included with PowerShell 7 by default, makes editing at the command line easier and more on par with the features available in Linux shells.

Installing PowerShell 7

PowerShell 7 is not installed in Windows by default, at least not at the time of writing. The PowerShell team has made PowerShell 7.1 available from the Microsoft Store, which is useful to install PowerShell 7.1 or later on Windows 10 systems. As the Microsoft Store is not overly relevant to Windows Server installations, you can install PowerShell by using a script, `Install-PowerShell.ps1`, which you download from the internet, as shown in this recipe. You can also use this recipe on Windows 10 hosts.

Getting ready

This recipe uses SRV1, a Windows Server workgroup host. There are no features of applications loaded on this server (yet).

How to do it...

1. Setting an execution policy for Windows PowerShell (in a new Windows PowerShell console)

```
Set-ExecutionPolicy -ExecutionPolicy Unrestricted -Force
```

2. Installing the latest versions of NuGet and PowerShellGet

```
Install-PackageProvider Nuget -MinimumVersion 2.8.5.201 -Force |
  Out-Null
Install-Module -Name PowerShellGet -Force -AllowClobber
```

3. Ensuring the C:\Foo folder exists

```
$LFHT = @{
  ItemType    = 'Directory'
  ErrorAction = 'SilentlyContinue' # should it already exist
}
New-Item -Path C:\Foo @LFHT | Out-Null
```

4. Downloading the PowerShell 7 installation script

```
Set-Location C:\Foo
$URI = 'https://aka.ms/install-powershell.ps1'
Invoke-RestMethod -Uri $URI |
  Out-File -FilePath C:\Foo\Install-PowerShell.ps1
```

5. Viewing the installation script help

```
Get-Help -Name C:\Foo\Install-PowerShell.ps1
```

6. Installing PowerShell 7

```
$EXTHT = @{
  UseMSI                = $true
  Quiet                 = $true
  AddExplorerContextMenu = $true
  EnablePSRemoting      = $true
}
C:\Foo\Install-PowerShell.ps1 @EXTHT | Out-Null
```

7. For the Adventurous – installing the preview and daily builds as well

```
C:\Foo\Install-PowerShell.ps1 -Preview -Destination C:\PWSHPreview |
  Out-Null
C:\Foo\Install-PowerShell.ps1 -Daily   -Destination C:\PWSHDailBuild |
  Out-Null
```

8. Creating Windows PowerShell default profiles

```
$URI = 'https://raw.githubusercontent.com/doctordns/PACKT-PS7/master/' +
       '/scripts/goodies/Microsoft.PowerShell_Profile.ps1'
$ProfileFile    = $Profile.CurrentUserCurrentHost
New-Item $ProfileFile -Force -WarningAction SilentlyContinue |
   Out-Null
(Invoke-WebRequest -Uri $URI -UseBasicParsing).Content |
  Out-File -FilePath  $ProfileFile
$ProfilePath    = Split-Path -Path $ProfileFile
$ChildPath      = 'Microsoft.PowerShell_profile.ps1'
$ConsoleProfile = Join-Path -Path $ProfilePath -ChildPath $ChildPath
(Invoke-WebRequest -Uri $URI -UseBasicParsing).Content |
  Out-File -FilePath  $ConsoleProfile
```

9. Checking the versions of PowerShell 7 loaded

```
Get-ChildItem -Path C:\pwsh.exe -Recurse -ErrorAction SilentlyContinue
```

How it works...

In *step 1*, you open a new Windows PowerShell console and set the PowerShell execution policy to unrestricted, which simplifies using scripts to configure hosts. In production, you may wish to set PowerShell's execution policy to be more restrictive. But note that an execution policy is not truly a security mechanism – it just slows down an inexperienced administrator.

For a good explanation of PowerShell's Security Guiding Principles, see `https://devblogs.microsoft.com/powershell/powershells-security-guiding-principles/`.

The PowerShell Gallery is a repository of PowerShell modules and scripts and is an essential resource for the IT pro. This book makes use of several modules from the PowerShell Gallery. In *step 2*, you update both the NuGet package provider (to version 2.8.5.201 or later) and an updated version of the PowerShellGet module.

Throughout this book, you'll use the `C:\Foo` folder to hold various files that you use in conjunction with the recipes. In *step 3*, you ensure the folder exists.

PowerShell 7 is not installed by default, at present, in Windows (or macOS or Linux), although this could change. To enable you to install PowerShell 7 in Windows, you retrieve an installation script from GitHub and store that in the `C:\Foo` folder. In *step 4*, you use a shortcut URL that points to GitHub and then use `Invoke-RestMethod` to download the file. Note that *steps 1* through *4* produce no output.

In *step 5*, you view the help information contained in the help file, which produces the following output:

```
PS C:\Foo]> # 5. Viewing the installation script help
PS C:\Foo]> Get-Help -Name C:\Foo\Install-PowerShell.ps1
Install-PowerShell.ps1 [-Destination <string>] [-Daily] [-DoNotOverwrite] [-AddToPath] [-Preview] [<CommonParameters>]
Install-PowerShell.ps1 [-UseMSI] [-Quiet] [-AddExplorerContextMenu] [-EnablePSRemoting] [-Preview] [<CommonParameters>]
```

Figure 1.1: Viewing installation script help

Note that after installing PowerShell 7, the first time you run `Get-Help` you are prompted to download help text (which is not shown in this figure).

In *step 6*, you use the installation script and install PowerShell 7. The commands use an MSI, which you then install silently. The MSI updates the system execution path to add the PowerShell 7 installation folder. The code retrieves the latest supported version of PowerShell 7, and you can view the actual filename in the following output:

```
PS C:\Foo> # 6. Installing PowerShell 7
PS C:\Foo> $EXTHT = @{
             UseMSI                = $true
             Quiet                 = $true
             AddExplorerContextMenu = $true
             EnablePSRemoting      = $true
           }
PS C:\Foo> C:\Foo\Install-PowerShell.ps1 @EXTHT | Out-Null
VERBOSE: About to download package from
'https://github.com/PowerShell/PowerShell/releases/download/v7.1.0/PowerShell-7.1.0-win-x64.msi'
```

Figure 1.2: Installing PowerShell 7

PowerShell 7 is a work in progress. Every day, the PowerShell team builds updated versions of PowerShell and releases previews of the next major release. The preview builds are mostly stable and allow you to try out new features that are coming in the next major release. The daily build allows you to view progress on a specific bug or feature. You may find it useful to install both of these (and ensure you keep them up to date as time goes by).

In *step 7*, you install the daily build and the latest preview build, which looks like this:

```
PS C:\Foo> # 7. For the Adventurous - installing the preview and daily builds as well
PS C:\Foo> C:\Foo\Install-PowerShell.ps1 -Preview -Destination c:\PWSHPreview |
             Out-Null
VERBOSE: Destination: C:\PWSHPreview
VERBOSE: About to download package from 'https://github.com/PowerShell/PowerShell/releases/
download/v7.1.0-preview.6/PowerShell-7.1.0-preview.6-win-x64.zip'
PowerShell has been installed at C:\PWSHPreview

PS C:\Foo> C:\Foo\Install-PowerShell.ps1 -Daily   -Destination c:\PWSHDailBuild |
             Out-Null
VERBOSE: Destination: C:\PWSHDailBuild
VERBOSE: About to download package from 'https://pscoretestdata.blob.core.windows.net/v7-1-
0-daily-20200909/PowerShell-7.1.0-daily.20200909-win-x64.zip'
PowerShell has been installed at C:\PWSHDailBuild
```

Figure 1.3: Installing the preview and daily builds

PowerShell uses **profile files** to enable you to configure PowerShell each time you run it (whether in the PowerShell console or as part of VS Code or the ISE). In *step 8*, you download a sample PowerShell profile script and save it locally. Note that the profile file you create in *step 8* is for Windows PowerShell only.

The executable name for PowerShell 7 is `pwsh.exe`. In *step 9*, you view the versions of this file as follows:

```
PS C:\Foo> # 9. Checking versions of PowerShell 7 loaded
PS C:\Foo> Get-ChildItem -Path C:\pwsh.exe -Recurse -ErrorAction SilentlyContinue


    Directory: C:\Program Files\PowerShell\7

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        06/11/2020     02:33         280456 pwsh.exe


    Directory: C:\PSDailyBuild

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        13/11/2020     01:08         269824 pwsh.exe


    Directory: C:\PSPreview

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        13/11/2020     20:03         274304 pwsh.exe
```

Figure 1.4: Checking PowerShell 7 versions loaded

As you can see, there are three versions of PowerShell 7 installed on SRV1: the latest full release, the latest preview, and the build of the day.

There's more...

In *step 1*, you open a new Windows PowerShell console. Make sure you run the console as the local administrator.

In *step 4*, you use a shortened URL to download the `Install-PowerShell.ps1` script. When you use `Invoke-RestMethod`, PowerShell discovers the underlying target URL for the script. The short URL allows Microsoft and the PowerShell team to publish a well-known URL and then have the flexibility to move the target location should that be necessary. The target URL, at the time of writing, is `https://raw.githubusercontent.com/PowerShell/PowerShell/master/tools/install-powershell.ps1`.

In *step 7*, you install both the latest daily build and the latest preview versions. The specific file versions you see are going to be different from the output shown here, at least for the preview versions!

Using the PowerShell 7 console

With PowerShell 7, the name of the PowerShell executable is now `pwsh.exe`, as you saw in the previous recipe. After installing PowerShell 7 in Windows, you can start the PowerShell 7 console by clicking **Start** and typing `pwsh.exe`, then hitting *Return*. The PowerShell MSI installer does not create a Start panel or taskbar shortcut.

Getting ready

You run this recipe on SRV1 after you have installed PowerShell 7.

How to do it...

1. Running the PowerShell 7 console

   From the Windows desktop in SRV1, click on the Windows key, then type `pwsh`, followed by the *Enter* key.

2. Viewing the PowerShell version

   ```
   $PSVersionTable
   ```

3. Viewing the `$Host` variable

```
$Host
```

4. Looking at the PowerShell process

```
Get-Process -Id $Pid |
  Format-Custom MainModule -Depth 1
```

5. Looking at resource usage statistics

```
Get-Process -Id $Pid |
  Format-List CPU,*Memory*
```

6. Updating the PowerShell help

```
$Before = Get-Help -Name about_*
Update-Help -Force | Out-Null
$After = Get-Help -Name about_*
$Delta = $After.Count - $Before.Count
"{0} Conceptual Help Files Added" -f $Delta
```

7. How many commands are available?

```
Get-Command |
  Group-Object -Property CommandType
```

How it works...

In *step 1*, you start the PowerShell 7 console on `SRV1`. The console should look like this:

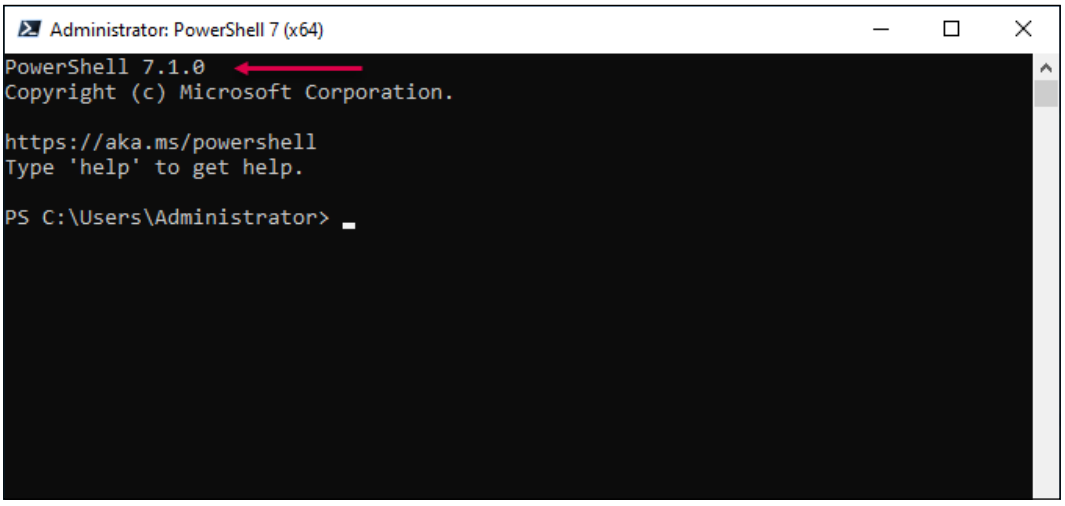


Figure 1.5: The PowerShell 7 console

In *step 2*, you view the specific version of PowerShell by looking at the built-in variable `$PSVersionTable`, which looks like this:

```
PS C:\Foo> # 2. Viewing the PowerShell version
PS C:\Foo> $PSVersionTable

Name                           Value
----                           -----
PSVersion                      7.1.0
PSEdition                      Core
GitCommitId                    7.1.0
OS                             Microsoft Windows 10.0.20270
Platform                       Win32NT
PSCompatibleVersions           {1.0, 2.0, 3.0, 4.0…}
PSRemotingProtocolVersion      2.3
SerializationVersion           1.1.0.1
WSManStackVersion              3.0
```

Figure 1.6: Viewing the PowerShell version

In *step 3*, you examine the `$Host` variable to determine details about the PowerShell 7 host (the PowerShell console), which looks like this:

```
PS C:\Foo> # 3. Viewing the $Host variable
PS C:\Foo> $Host

Name             : ConsoleHost
Version          : 7.1.0
InstanceId       : ea087cdd-ddbe-43ef-b2eb-4c6cb8d8c541
UI               : System.Management.Automation.Internal.Host.InternalHostUserInterface
CurrentCulture   : en-GB
CurrentUICulture : en-US
PrivateData      : Microsoft.PowerShell.ConsoleHost+ConsoleColorProxy
DebuggerEnabled  : True
IsRunspacePushed : False
Runspace         : System.Management.Automation.Runspaces.LocalRunspace
```

Figure 1.7: Viewing the $Host variable

As you can see, in this case, the current culture is `EN-GB`. You may see a different value depending on which specific version of Windows Server you are using.

In *step 4*, you use `Get-Process` to look at the details of the PowerShell process, which looks like this:

```
PS C:\Foo> # 4. Looking at the PowerShell process
PS C:\Foo> Get-Process -Id $PID |
             Format-Custom -Property MainModule -Depth 1

class Process
{
  MainModule =
    class ProcessModule
    {
      ModuleName = pwsh.exe
      FileName = C:\Program Files\PowerShell\7\pwsh.exe
      BaseAddress = 140695338418176
      ModuleMemorySize = 290816
      EntryPointAddress = 140695338483552
      FileVersionInfo = File:             C:\Program Files\PowerShell\7\pwsh.exe
      InternalName:     pwsh.dll
      OriginalFilename: pwsh.dll
      FileVersion:      7.1.0.0
      FileDescription:  pwsh
      Product:          PowerShell
      ProductVersion:   7.1.0 SHA: d2953dcaf8323b95371380639ced00dac4ed209f
      Debug:            False
      Patched:          False
      PreRelease:       False
      PrivateBuild:     False
      SpecialBuild:     False
      Language:         Language Neutral

      Site =
      Container =
      Size = 284
      Company = Microsoft Corporation
      FileVersion = 7.1.0.0
      ProductVersion = 7.1.0 SHA: d2953dcaf8323b95371380639ced00dac4ed209f
      Description = pwsh
      Product = PowerShell
    }
}
```

Figure 1.8: Looking at the PowerShell process details

In this figure, you can see the path to the PowerShell 7 executable. This value changes depending on whether you are running the release version or the daily/preview releases.

You can see, in *step 5*, details of resource usage of the `pwsh.exe` process running on the `SRV1` host:

```
PS C:\Foo> # 5. Looking at resource usage statistics
PS C:\Foo> Get-Process -Id $PID |
             Format-List CPU,*Memory*

CPU                        : 4.765625
NonpagedSystemMemorySize64 : 74552
NonpagedSystemMemorySize   : 74552
PagedMemorySize64          : 63938560
PagedMemorySize            : 63938560
PagedSystemMemorySize64    : 443552
PagedSystemMemorySize      : 443552
PeakPagedMemorySize64      : 72388608
PeakPagedMemorySize        : 72388608
PeakVirtualMemorySize64    : 2204211732480
PeakVirtualMemorySize      : 893509632
PrivateMemorySize64        : 63938560
PrivateMemorySize          : 63938560
VirtualMemorySize64        : 2204191739904
VirtualMemorySize          : 873517056
```

Figure 1.9: Looking at the resource usage statistics of pwsh.exe

The values of each of the performance counters are likely to vary, and you may see different values.

By default, PowerShell 7, like Windows PowerShell, ships with minimum help files. You can, as you can see in *step 6*, use the `Update-Help` command to download updated PowerShell 7 help content, like this:

```
PS C:\Users\Administrator> # 6. Updating the PowerShell help
PS C:\Users\Administrator> $Before = Get-Help -Name about_*
PS C:\Users\Administrator> Update-Help -Force | Out-Null
Update-Help: Failed to update Help for the module(s) 'PSReadline' with UI culture(s) {en-US} :
One or more errors occurred. (Response status code does not indicate success: 404
(The specified blob does not exist.).).
English-US help content is available and can be installed using: Update-Help -UICulture en-US.
PS C:\Users\Administrator> $After = Get-Help -Name about_*
PS C:\Users\Administrator> $Delta = $After.Count - $Before.Count
PS C:\Users\Administrator> $Delta = $After.Count - $Before.Count
PS C:\Users\Administrator> "{0} Conceptual Help Files Added" -f $Delta
128 Conceptual Help Files Added
```

Figure 1.10: Updating PowerShell help

As you can see from the output, not all help files were updated. In this case, the `ConfigDefender` module does not, at present, have updated help information. Also note that although the UK English versions of help details may be missing, there are US English versions that you can install that may be useful. There is no material difference between the UK and US texts.

Commands in PowerShell include functions, cmdlets, and aliases. In *step 7*, you examine how many of each type of command is available by default, like this:

```
PS C:\Foo> # 7. How many commands are available?
PS C:\Foo> Get-Command |
             Group-Object -Property CommandType

Count Name      Group
----- ----      -----
   58 Alias     {Add-AppPackage, Add-AppPackageVolume, Add-AppProvisionedPackage, Add-ProvisionedAppPa…
 1144 Function  {A:, Add-BCDataCacheExtension, Add-DnsClientDohServerAddress, Add-DnsClientNrptRule, A…
  587 Cmdlet    {Add-AppxPackage, Add-AppxProvisionedPackage, Add-AppxVolume, Add-BitsFile, Add-Certif…
```

Figure 1.11: Examining the number of each type of command available

There's more...

In *step 1*, you open the PowerShell console for the version of PowerShell you installed in the *Installing PowerShell 7* recipe. With the release of PowerShell 7.1, the version number you would see is 7.1.0. By the time you read this book, that version number may have advanced. To ensure you have the latest released version of PowerShell 7, re-run the `Install-PowerShell.ps1` script you downloaded in the *Installing PowerShell 7* recipe.

Also, in *step 1*, you can see the output generated by the `Write-Host` statements in the profile file you set up in *Installing PowerShell 7*. You can remove these statements to reduce the amount of output you see each time you start up PowerShell.

In *step 4*, you use the variable `$PID`, which contains the Windows process identifier of the PowerShell 7 console process. The actual value of `$PID` changes each time you run PowerShell, but the value always contains the process ID of the current console process.

In *step 6*, you can see an error in the `PSReadLine` module's help information (*Figure 1.10*). The developers of this module have, in later versions, renamed this module to `PSReadLine` (capitalizing the L in Line). However, help URLs are, for some reason, case-sensitive. You can fix this by renaming the module on disk and capitalizing the folder name.

In *step 7*, you saw that you had 1786 commands available. This number changes as you add more features (and their accompanying modules) or download and install modules from repositories such as the PowerShell Gallery.

Exploring PowerShell 7 installation artifacts

In PowerShell 7, certain objects added by the PowerShell 7 installer (and PowerShell 7) differ from those used by Windows PowerShell.

Getting ready

This recipe uses SRV1 after you have installed PowerShell 7. In this recipe, you use the PowerShell 7 console to run the steps.

How to do it...

1. Checking the version table for the PowerShell 7 console

```
$PSVersionTable
```

2. Examining the PowerShell 7 installation folder

```
Get-Childitem -Path $env:ProgramFiles\PowerShell\7 -Recurse |
  Measure-Object -Property Length -Sum
```

3. Viewing the PowerShell 7 configuration JSON file

```
Get-ChildItem -Path $env:ProgramFiles\PowerShell\7\powershell*.json |
  Get-Content
```

4. Checking the initial Execution Policy for PowerShell 7

```
Get-ExecutionPolicy
```

5. Viewing the module folders

```
$I = 0
$ModPath = $env:PSModulePath -split ';'
$ModPath |
  Foreach-Object {
    "[{0:N0}]   {1}" -f $I++, $_
  }
```

6. Checking the modules

```
$TotalCommands = 0
Foreach ($Path in $ModPath){
  Try { $Modules = Get-ChildItem -Path $Path -Directory -ErrorAction Stop
        "Checking Module Path:  [$Path]"
  }
  Catch [System.Management.Automation.ItemNotFoundException] {
    "Module path [$path] DOES NOT EXIST ON $(hostname)"
  }
  $CmdsInPath = 0
  Foreach ($Module in $Modules) {
    $Cmds = Get-Command -Module ($Module.name)
    $TotalCommands += $Cmds.Count
  }
}
```

7. Viewing the total number of commands and modules

```
$Mods = (Get-Module * -ListAvailable | Measure-Object).count
"{0} modules providing {1} commands" -f $Mods,$TotalCommands
```

How it works...

In *step 1*, you examine the `$PSVersionTable` variable to view the version information for PowerShell 7, which looks like this:

```
PS C:\Users\Administrator> # 1. Checking the version table for PowerShell 7 console
PS C:\Users\Administrator> $PSVersionTable

Name                           Value
----                           -----
PSVersion                      7.1.0
PSEdition                      Core
GitCommitId                    7.1.0
OS                             Microsoft Windows 10.0.20270
Platform                       Win32NT
PSCompatibleVersions           {1.0, 2.0, 3.0, 4.0…}
PSRemotingProtocolVersion      2.3
SerializationVersion           1.1.0.1
WSManStackVersion              3.0
```

Figure 1.12: Checking the version information for PowerShell 7

The PowerShell 7 installation program installs PowerShell 7 into a different folder (by default) from that used by Windows PowerShell. In *step 2*, you see a summary of the files installed into the PowerShell 7 installation folder as follows:

```
PS C:\Users\Administrator> # 2. Examining the PowerShell 7 installation folder
PS C:\Users\Administrator> Get-Childitem -Path $env:ProgramFiles\PowerShell\7 -Recurse |
                             Measure-Object -Property Length -Sum

Count             : 982
Average           :
Sum               : 252056323
Maximum           :
Minimum           :
StandardDeviation :
Property          : Length
```

Figure 1.13: Examining the installation folder

PowerShell 7 stores configuration values in a JSON file in the PowerShell 7 installation folder. In *step 3*, you view the contents of this file:

```
PS C:\Users\Administrator> # 3. Viewing PowerShell configuration JSON file
PS C:\Users\Administrator> Get-ChildItem -Path $env:ProgramFiles\PowerShell\7\powershell*.json |
                             Get-Content
{
    "WindowsPowerShellCompatibilityModuleDenyList":  [
                                                         "PSScheduledJob",
                                                         "BestPractices",
                                                         "UpdateServices"
                                                     ],
    "Microsoft.PowerShell:ExecutionPolicy":  "RemoteSigned"
}
```

Figure 1.14: Viewing the JSON configuration file

In *step 4*, you view the execution policy for PowerShell 7, as follows:

```
PS C:\Users\Administrator> # 4. Checking initial Execution Policy for PowerShell 7
PS C:\Users\Administrator> Get-ExecutionPolicy
RemoteSigned
```

Figure 1.15: Checking the execution policy for PowerShell 7

As with Windows PowerShell, PowerShell 7 loads commands from modules. PowerShell uses the `$PSModulePath` variable to determine which file store folders PowerShell 7 should use to find these modules. Viewing the contents of this variable, and discovering the folders, in *step 5*, looks like this:

```
PS C:\Users\Administrator> # 5. Viewing module folders
PS C:\Users\Administrator> $I = 0
PS C:\Users\Administrator> $ModPath = $env:PSModulePath -split ';'
PS C:\Users\Administrator> $ModPath |
                             Foreach-Object {
                              "[{0:N0}]   {1}" -f $I++, $_
                           }
[0]   C:\Users\Administrator\Documents\PowerShell\Modules
[1]   C:\Program Files\PowerShell\Modules
[2]   c:\program files\powershell\7\Modules
[3]   C:\Program Files\WindowsPowerShell\Modules
[4]   C:\Windows\system32\WindowsPowerShell\v1.0\Modules
```

Figure 1.16: Viewing the module folders

With those module folders defined (by default), you can check how many commands exist in each folder, in *step 6*, the output of which looks like this:

```
PS C:\Users\Administrator> # 6. Checking the modules
PS C:\Users\Administrator> $TotalCommands = 0
PS C:\Users\Administrator> Foreach ($Path in $ModPath){
                             Try { $Modules = Get-ChildItem -Path $Path -Directory -ErrorAction Stop
                                   "Checking Module Path:  [$Path]"
                             }
                             Catch [System.Management.Automation.ItemNotFoundException] {
                             "Module path [$path] DOES NOT EXIST ON $(hostname)"
                             }
                             $CmdsInPath = 0
                             Foreach ($Module in $Modules) {
                               $Cmds = Get-Command -Module ($Module.name)
                               $TotalCommands += $Cmds.Count
                             }
                           }
Module path [C:\Users\Administrator\Documents\PowerShell\Modules] DOES NOT EXIST ON SRV1
Module path [C:\Program Files\PowerShell\Modules] DOES NOT EXIST ON SRV1
Checking Module Path:  [c:\program files\powershell\7\Modules]
Checking Module Path:  [C:\Program Files\WindowsPowerShell\Modules]
Checking Module Path:  [C:\Windows\system32\WindowsPowerShell\v1.0\Modules]]
```

Figure 1.17: Checking how many commands exist in each module folder

In *step 7*, you can view the results to see how many commands exist in each of the modules in each module path. The output looks like this:

```
PS C:\Users\Administrator> # 7. Viewing totals of commands and modules
PS C:\Users\Administrator> $Mods = (Get-Module * -ListAvailable | Measure-Object).count
PS C:\Users\Administrator> "{0} modules providing {1} commands" -f $Mods,$TotalCommands
72 modules providing 4930 commands
```

Figure 1.18: Viewing the total number of modules and commands

There's more...

In *step 1*, you viewed the PowerShell version table. Depending on when you read this book, the version numbers you see may be later than shown here. You would see this when the PowerShell team releases an updated version of PowerShell.

In *step 4*, you viewed PowerShell 7's execution policy. Each time PowerShell 7 starts up, it reads the JSON file to obtain the value of the execution policy. You can use the `Set-Execution` policy to reset the policy immediately, or change the value in the JSON file and restart the PowerShell 7 console.

In *step 5*, you viewed the default folders that PowerShell 7 uses to search for a module (by default). The first folder is your personal modules, followed by PowerShell 7, and then the Windows PowerShell modules. We cover the Windows PowerShell modules and Windows PowerShell compatibility in more detail in *Chapter 2*, *Introducing PowerShell 7*.

Building PowerShell 7 profile files

In Windows PowerShell and PowerShell 7, profile files are PowerShell scripts that PowerShell runs each time you start a new instance of PowerShell (whether the console, ISE, or VS Code). These files enable you to pre-configure PowerShell 7. You can add variables, PowerShell PSDrives, functions, and more using profiles. As part of this book, you download and install initial profile files based on samples that you can download from GitHub.

This recipe downloads and installs the profile files for the PowerShell 7 console.

Getting ready

You run this recipe on SRV1 after you have installed PowerShell 7.

How to do it...

1. Discovering the profile filenames

```
$ProfileFiles = $PROFILE |  Get-Member -MemberType NoteProperty
$ProfileFiles | Format-Table -Property Name, Definition
```

2. Checking for the existence of each PowerShell profile file

```
Foreach ($ProfileFile in $ProfileFiles){
  "Testing $($ProfileFile.Name)"
  $ProfilePath = $ProfileFile.Definition.split('=')[1]
  If (Test-Path $ProfilePath){
    "$($ProfileFile.Name) DOES EXIST"
    "At $ProfilePath"
  }
  Else {
    "$($ProfileFile.Name) DOES NOT EXIST"
  }
  ""
}
```

3. Displaying the Current User/Current Host profile

```
$CUCHProfile = $PROFILE.CurrentUserCurrentHost
"Current User/Current Host profile path: [$CUCHPROFILE]"
```

4. Creating a Current User/Current Host profile for the PowerShell 7 console

```
$URI = 'https://raw.githubusercontent.com/doctordns/PACKT-PS7/master/' +
       'scripts/goodies/Microsoft.PowerShell_Profile.ps1'
New-Item $CUCHProfile -Force -WarningAction SilentlyContinue |
   Out-Null
(Invoke-WebRequest -Uri $URI).Content |
  Out-File -FilePath  $CUCHProfile
```

5. Exiting the PowerShell 7 console

```
Exit
```

6. Restarting the PowerShell 7 console and viewing the profile output at startup

```
Get-ChildItem -Path $Profile
```

How it works...

In *step 1*, you discover the names of each of the four profile files (for the PowerShell 7 console), then view their name and location (that is, the definition), which looks like this:

```
PS C:\Users\Administrator> # 1. Discovering the profile file names
PS C:\Users\Administrator> $ProfileFiles = $profile |  Get-Member -MemberType NoteProperty
PS C:\Users\Administrator> $ProfileFiles | Format-Table -Property Name, Definition

Name                  Definition
----                  ----------
AllUsersAllHosts      string AllUsersAllHosts=C:\Program Files\PowerShell\7\profile.ps1
AllUsersCurrentHost   string AllUsersCurrentHost=C:\Program Files\PowerShell\7\Microsoft.PowerShell_profile.ps1
CurrentUserAllHosts   string CurrentUserAllHosts=C:\Users\Administrator\Documents\PowerShell\profile.ps1
CurrentUserCurrentHost string CurrentUserCurrentHost=C:\Users\Administrator\Documents\PowerShell\Microsoft.PowerShell_…
```

Figure 1.19: Discovering the profile filenames

In *step 2*, you check to see which, if any, of the profile files exist, which looks like this:

```
PS C:\Users\Administrator> # 2. Checking for existence of each PowerShell profile file
PS C:\Users\Administrator> Foreach ($ProfileFile in $ProfileFiles){
                             "Testing $($ProfileFile.Name)"
                             $ProfilePath = $ProfileFile.Definition.split('=')[1]
                             If (Test-Path $ProfilePath){
                               "$($ProfileFile.Name) DOES EXIST"
                               "At $ProfilePath"
                             }
                             Else {
                               "$($ProfileFile.Name) DOES NOT EXIST"
                             }
                             ""
                           }
Testing AllUsersAllHosts
AllUsersAllHosts DOES NOT EXIST

Testing AllUsersCurrentHost
AllUsersCurrentHost DOES NOT EXIST

Testing CurrentUserAllHosts
CurrentUserAllHosts DOES NOT EXIST

Testing CurrentUserCurrentHost
CurrentUserCurrentHost DOES NOT EXIST
```

Figure 1.20: Checking for the existence of PowerShell profile files

In *step 3*, you obtain and display the filename of the Current User/Current Host profile file, which looks like this:

```
PS C:\Users\Administrator> # 3. Discovering Current User/Current Host profile
PS C:\Users\Administrator> $CUCHProfile = $PROFILE.CurrentUserCurrentHost
PS C:\Users\Administrator> "Current User/Current Host profile path: [$CUCHPROFILE]"
Current User/Current Host profile path: [C:\Users\Administrator\Documents\PowerShell\Microsoft.PowerShell_profile.ps1]
```

Figure 1.21: Discovering the Current User/Current Host profile file

In *step 4*, you create an initial Current User/Current Host profile file. This file is part of the GitHub repository that supports this book. In *step 5*, you exit the current PowerShell 7 console host. These two steps create no output.

In *step 6*, you start a new PowerShell profile. This time, as you can see here, the profile file exists, runs, and customizes the console:

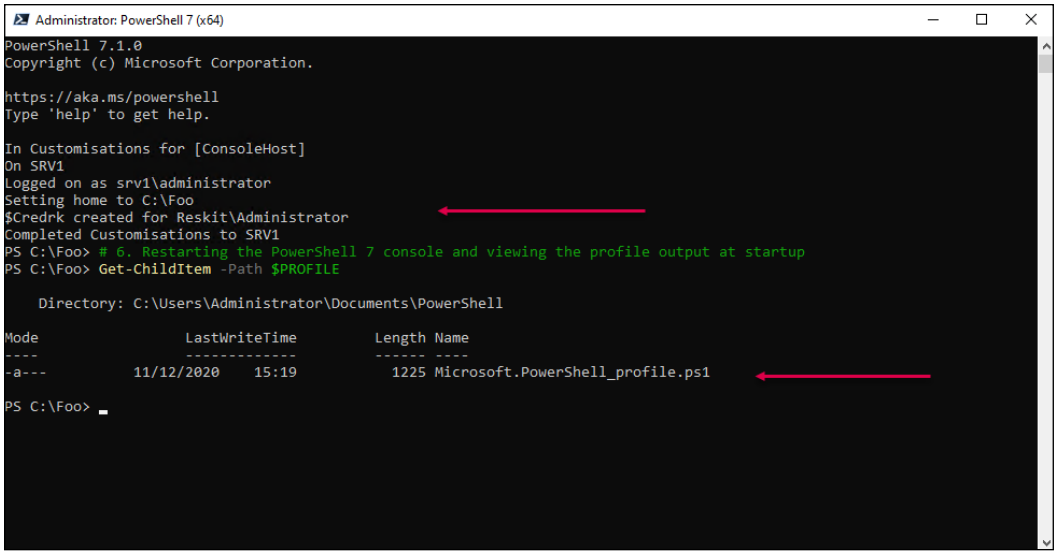


Figure 1.22: The profile file in action

There's more...

In *step 4*, you download a sample profile file from GitHub. This sample profile file contains customizations for the PowerShell console configuration, including changing the default starting folder (to `C:\Foo`) and creating some aliases, PowerShell drives, and a credential object. These represent sample content you might consider including in your console profile file. Note that VS Code, which you install in the next recipe, uses a separate Current User/Current Host profile file, which enables you to customize PowerShell at the console and in VS Code differently.

Installing VS Code

The Windows PowerShell ISE was a great tool that Microsoft first introduced with Windows PowerShell v2 (and vastly improved with v3). This tool has reached feature completeness, and Microsoft has no plans for further development.

In its place, however, is Visual Studio Code, or VS Code. This open-source tool provides an extensive range of features for IT pros and others. For IT professionals, this should be your editor of choice. While there is a learning curve (as for any new product), VS Code contains all the features you find in the ISE and more.

VS Code, and the available extensions, are works in progress. Each new release brings additional features which can be highly valuable. A recent addition from Draw.io, for example, is the ability to create diagrams directly in VS Code. Take a look at this post for more details on this diagram tool: `https://tfl09.blogspot.com/2020/05/over-weekend-i-saw-tweet-announcing-new.html`.

There are a large number of other extensions you might be able to use, depending on your workload. For more details on VS Code, see `https://code.visualstudio.com/`.

For details of the many VS Code extensions, you can visit `https://code.visualstudio.com/docs/editor/extension-marketplace`.

Getting ready

You run this recipe on `SRV1` after you have installed PowerShell 7 and have created a console profile file.

How to do it...

1. Downloading the VS Code installation script from the PowerShell Gallery

```
$VSCPATH = 'C:\Foo'
Save-Script -Name Install-VSCode -Path $VSCPATH
Set-Location -Path $VSCPATH
```

2. Running the installation script and adding in some popular extensions

```
$Extensions =  'Streetsidesoftware.code-spell-checker',
               'yzhang.markdown-all-in-one',
               'hediet.vscode-drawio'
$InstallHT = @{
  BuildEdition         = 'Stable-System'
  AdditionalExtensions = $Extensions
  LaunchWhenDone       = $true
}
.\Install-VSCode.ps1 @InstallHT
```

3. Exiting VS Code

   Close the VS Code window

4. Restarting VS Code as an administrator

   Click on the Windows key and type **code**, which brings up the VS Code tile in the Windows Start panel. Then, right click the VS Code tile and select **Run as Administrator** to start VS Code as an administrator.

5. Opening a new VS Code terminal window

   Inside VS Code, type *Ctrl* + *Shift* + `` ` ``.

6. Creating a Current User/Current Host profile for VS Code

```
$SAMPLE =
  'https://raw.githubusercontent.com/doctordns/PACKT-PS7/master/' +
  'scripts/goodies/Microsoft.VSCode_profile.ps1'
(Invoke-WebRequest -Uri $Sample).Content |
  Out-File $Profile
```

7. Updating the user settings for VS Code

```
$JSON = @'
{
  "workbench.colorTheme": "PowerShell ISE",
  "powershell.codeFormatting.useCorrectCasing": true,
  "files.autoSave": "onWindowChange",
  "files.defaultLanguage": "powershell",
  "editor.fontFamily": "'Cascadia Code',Consolas,'Courier New'",
  "workbench.editor.highlightModifiedTabs": true,
  "window.zoomLevel": 1
}
'@
$JHT = ConvertFrom-Json -InputObject $JSON -AsHashtable
$PWSH = "C:\\Program Files\\PowerShell\\7\\pwsh.exe"
$JHT += @{
  "terminal.integrated.shell.windows" = "$PWSH"
}
$Path = $Env:APPDATA
$CP   = '\Code\User\Settings.json'
$Settings = Join-Path  $Path -ChildPath $CP
$JHT |
  ConvertTo-Json  |
    Out-File -FilePath $Settings
```

8. Creating a shortcut to VS Code

```
$SourceFileLocation  = "$env:ProgramFiles\Microsoft VS Code\Code.exe"
$ShortcutLocation    = "C:\foo\vscode.lnk"
# Create a  new wscript.shell object
$WScriptShell        = New-Object -ComObject WScript.Shell
$Shortcut            = $WScriptShell.CreateShortcut($ShortcutLocation)
$Shortcut.TargetPath = $SourceFileLocation
#Save the Shortcut to the TargetPath
$Shortcut.Save()
```

9. Creating a shortcut to PowerShell 7

```
$SourceFileLocation  = "$env:ProgramFiles\PowerShell\7\pwsh.exe"
$ShortcutLocation    = 'C:\Foo\pwsh.lnk'
# Create a new wscript.shell object
$WScriptShell        = New-Object -ComObject WScript.Shell
$Shortcut            = $WScriptShell.CreateShortcut($ShortcutLocation)
$Shortcut.TargetPath = $SourceFileLocation
# Save the Shortcut to the TargetPath
$Shortcut.Save()
```

10. Building an updated layout XML

```
$XML = @'
<?xml version="1.0" encoding="utf-8"?>
<LayoutModificationTemplate
  xmlns="http://schemas.microsoft.com/Start/2014/LayoutModification"
  xmlns:defaultlayout=
    "http://schemas.microsoft.com/Start/2014/FullDefaultLayout"
  xmlns:start="http://schemas.microsoft.com/Start/2014/StartLayout"
  xmlns:taskbar="http://schemas.microsoft.com/Start/2014/TaskbarLayout"
  Version="1">
<CustomTaskbarLayoutCollection>
<defaultlayout:TaskbarLayout>
<taskbar:TaskbarPinList>
 <taskbar:DesktopApp DesktopApplicationLinkPath="C:\Foo\vscode.lnk" />
 <taskbar:DesktopApp DesktopApplicationLinkPath="C:\Foo\pwsh.lnk" />
</taskbar:TaskbarPinList>
</defaultlayout:TaskbarLayout>
</CustomTaskbarLayoutCollection>
</LayoutModificationTemplate>
'@
$XML | Out-File -FilePath C:\Foo\Layout.Xml
```

11. Importing the start layout XML file

    ```
    Import-StartLayout -LayoutPath C:\Foo\Layout.Xml -MountPath C:\
    ```

12. Logging of

    ```
    logoff.exe
    ```

13. Log back in to Windows and observe the taskbar
14. Run the PowerShell console from the shortcut
15. Run VS Code from the new taskbar shortcut and observe the profile file running

How it works...

In *step 1*, you download the VS Code installation script from the PowerShell Gallery. This step produces no output.

Then, in *step 2*, you run the installation script and add in three specific extensions. Running this step in the PowerShell 7 console looks like this:

```
PS C:\Foo> # 2. Running the installation script and adding in some popular extensions
PS C:\Foo> $Extensions =  'Streetsidesoftware.code-spell-checker',
                          'yzhang.markdown-all-in-one',
                          'hediet.vscode-drawio'
PS C:\Foo> $InstallHT = @{
             BuildEdition         = 'Stable-System'
             AdditionalExtensions = $Extensions
             LaunchWhenDone       = $true
           }
PS C:\Foo> .\Install-VSCode.ps1 @InstallHT | Out-Null

Installing extension ms-vscode.PowerShell...
Installing extensions...
Installing extension 'ms-vscode.powershell' v2020.6.0...

Installing extension Streetsidesoftware.code-spell-checker...

Installing extension yzhang.markdown-all-in-one...

Installing extension hediet.vscode-drawio...

Installation complete, starting Visual Studio Code (64-bit)...
```

Figure 1.23: Running the installation script and adding some extensions

Once VS Studio has started, you will see the initial opening window, which looks like this:

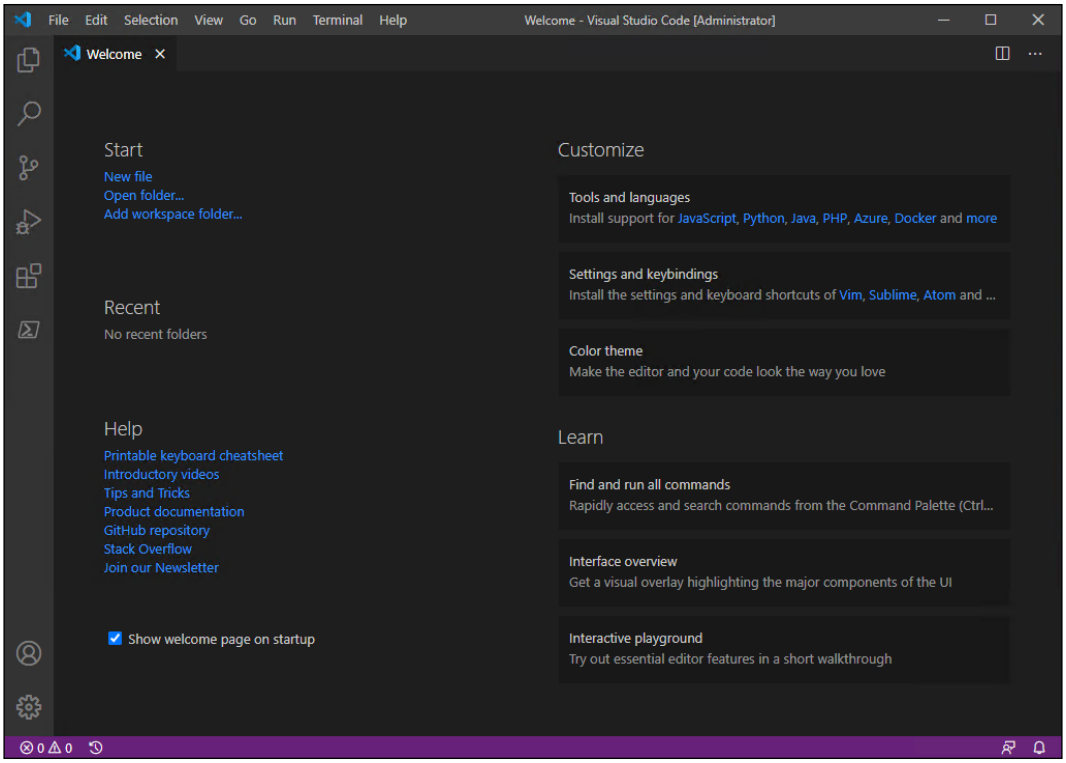


Figure 1.24: The VS Code Welcome window

In *step 3*, you close this window. Then, in *step 4*, you run VS Code as an administrator. In *step 5*, you open a new VS Code Terminal and run PowerShell 7, which now looks like this:

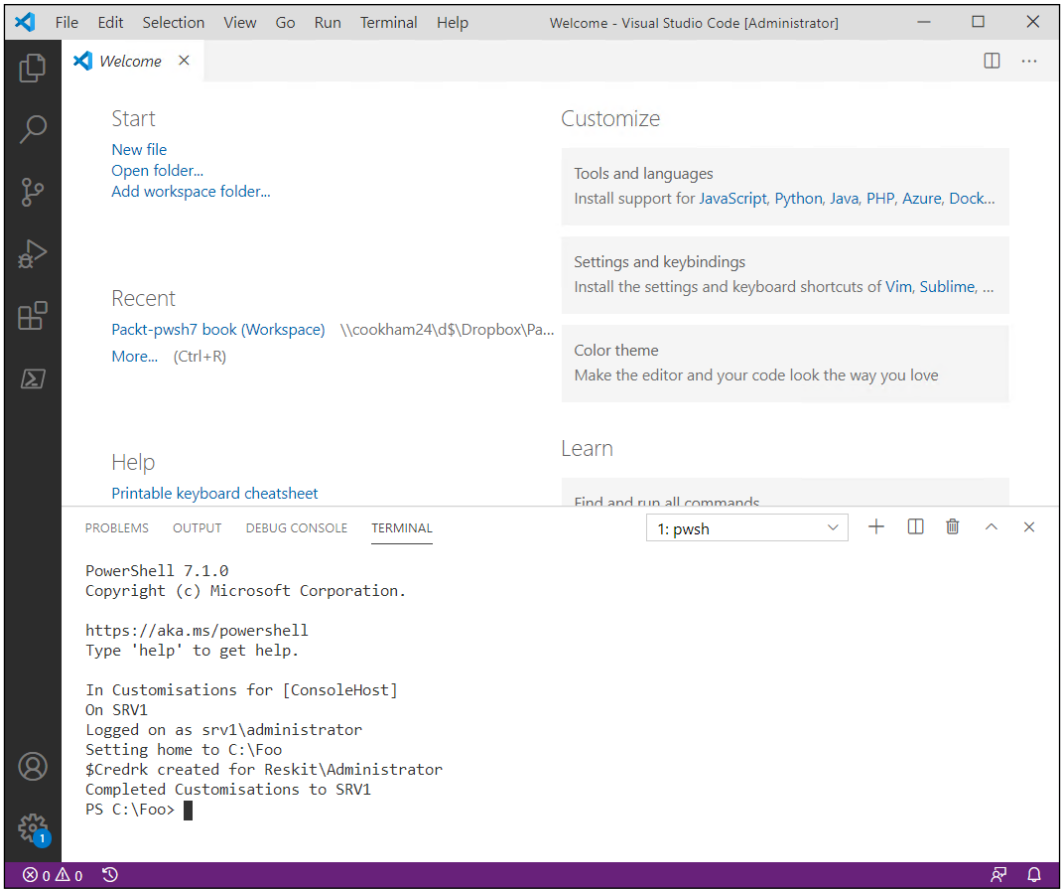


Figure 1.25: Running PowerShell 7 in VS Code

In *step 6*, you create a VS Code sample profile file. The VS Code PowerShell extension uses this profile file when you open a `.PS1` file. This step generates no output.

In *step 7*, you update several VS Code runtime options. In *step 8*, you create a shortcut to VS Code, and in *step 9*, you create a shortcut to the PowerShell 7 console. You use these later in this recipe to create a shortcut on your Windows taskbar.

In *step 10*, you update the XML that describes the Windows taskbar to add the shortcuts to VS Code and the PowerShell console you created previously. In *step 11*, you import the updated task pane description back into Windows. These steps produce no output as such.

Next, in *step 12*, you log off from Windows. In *step 13*, you re-login and note the updated taskbar, as you can see here:

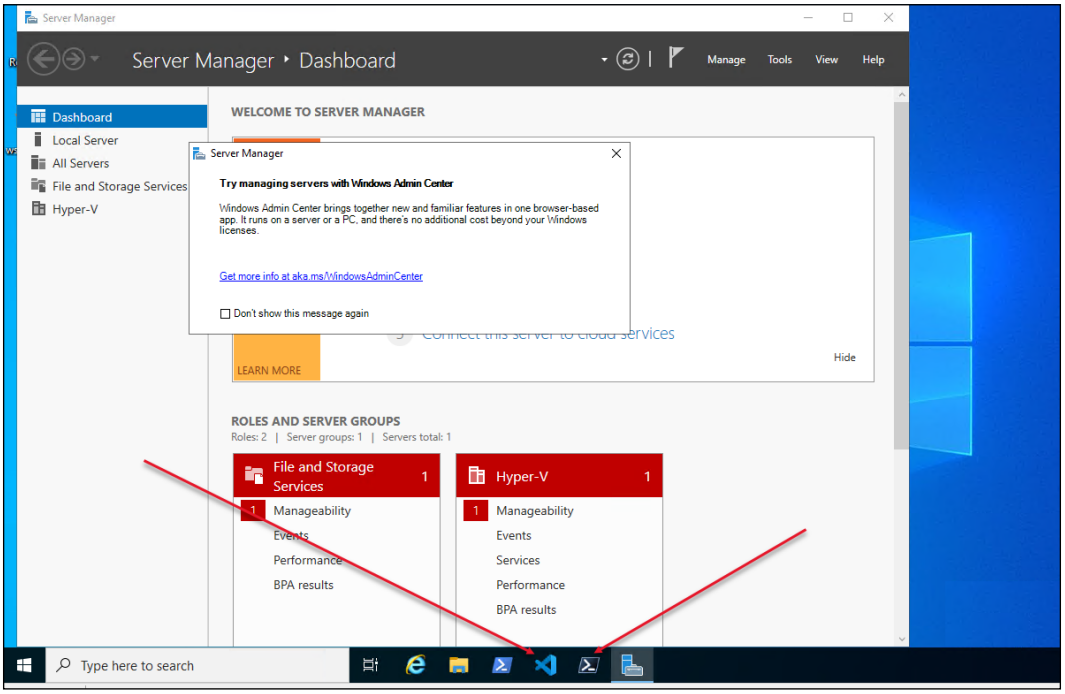


Figure 1.26: Updated taskbar with shortcuts

In *step 14*, you open a PowerShell 7 console, which looks like this:

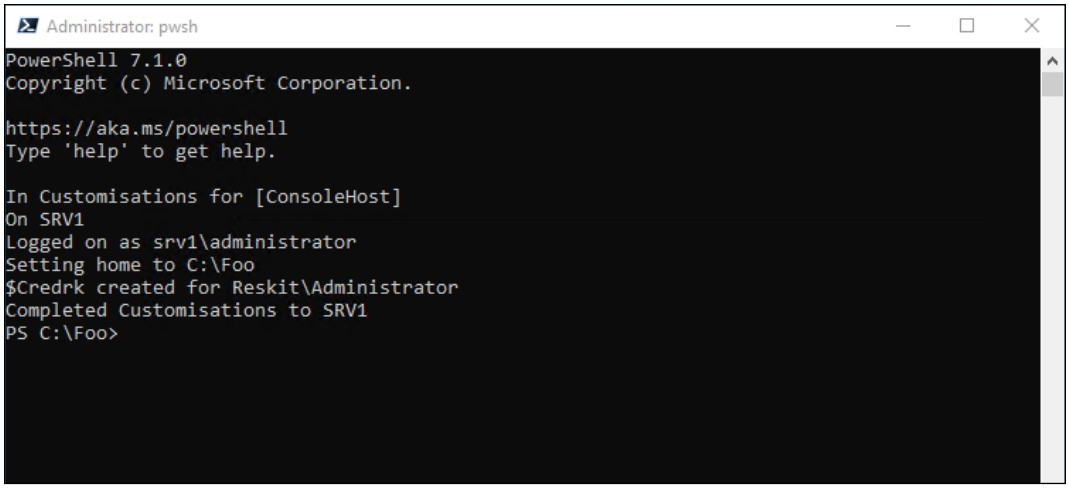


Figure 1.27: PowerShell 7 console (from shortcut)

In *step 15*, you open VS Code, which looks like this:

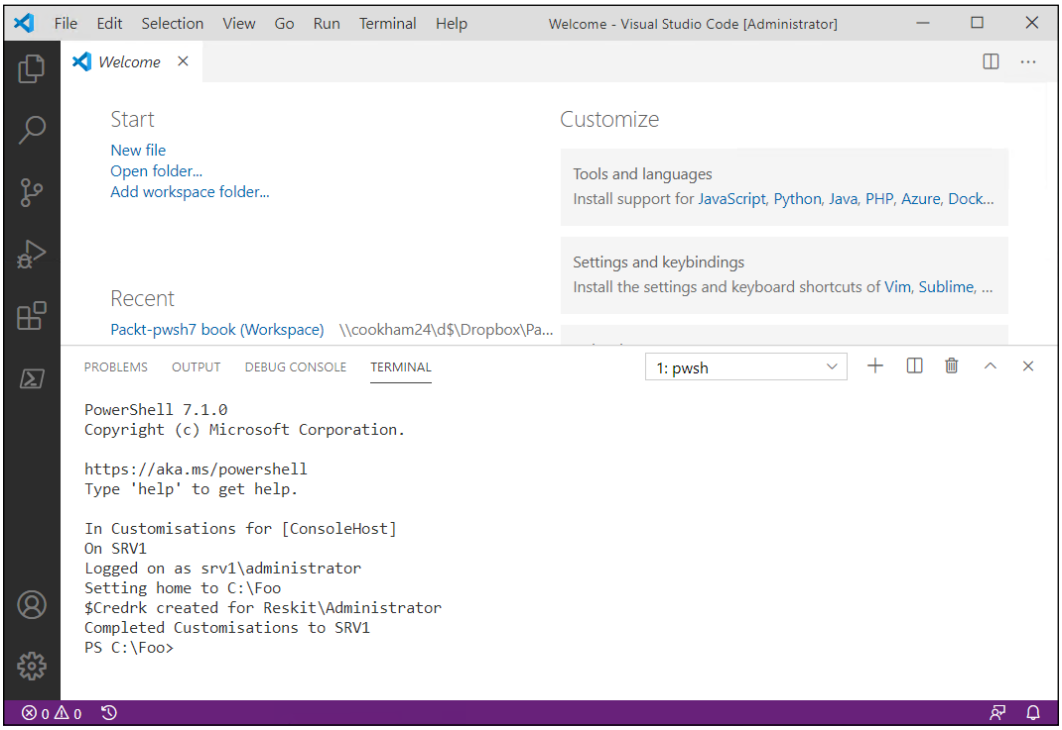


Figure 1.28: VS Code (from shortcut)

There's more...

In *step 2*, you install VS Code and three additional VS Code extensions. The `Streetsidesoftware.code-spell-checker` extension provides spell-checking for your scripts and other files. The `yzhang.markdown-all-in-one` extension supports the use of Markdown. This extension is useful if, for example, you are writing documentation in GitHub or updating the existing public PowerShell 7 help information. The `hediet.vscode-drawio` extension enables you to create rich diagrams directly in VS Code. Visit the VS Code Marketplace for details on these and other extensions.

In *step 4*, you ensure you are running VS Code as an administrator. Some of the code requires this and fails if you are not running PowerShell (inside VS Code) as an admin.

In *step 5*, you open a terminal inside VS Code. In VS Code, the "terminal" is initially a Windows PowerShell console. You can see in the output the results of running the profile file for Windows PowerShell. This terminal is the one you see inside VS Code by default.

In *step 7*, you update and save some updates to the VS Code settings. Note that in this step, you tell VS Code where to find the version of PowerShell you wish to run. You can, should you choose, change this to run a preview version of PowerShell or even the daily build.

Note that VS Code has, in effect, two PowerShell profile files at play. When you open VS Code on its own, you get the terminal you see in *step 5* – this is the default terminal. The PowerShell VS Code runs a separate terminal whenever you open a `.PS1` file. In that case, VS Code runs the VS Code-specific profile you set in *step 6*.

In *step 8* and *step 9*, you create shortcuts to VS Code and the PowerShell 7 console. In *step 10*, you update the layout of the Windows taskbar to include the two shortcuts. Unfortunately, you have to log off (as you do in *step 12*) before logging back in to Windows where you can observe and use the two shortcuts.

Installing the Cascadia Code font

As part of the launch of VS Code, Microsoft also created a new and free font that you can download and use both at the PowerShell 7 console and inside VS Code. This recipe shows how you can download the font, install it, and set this font to be the default in VS Code.

Getting ready

You run this recipe on SRV1 after you have installed both PowerShell 7 and VS Code.

How to do it...

1. Getting the download locations for the Cascadia Code font

```
$CascadiaFont    = 'Cascadia.ttf'    # font file name
$CascadiaRelURL  = 'https://github.com/microsoft/cascadia-code/releases'
$CascadiaRelease = Invoke-WebRequest -Uri $CascadiaRelURL # Get all
$CascadiaPath    = "https://github.com" + ($CascadiaRelease.Links.href |
                     Where-Object { $_ -match "($CascadiaFont)" } |
                       Select-Object -First 1)
$CascadiaFile    = "C:\Foo\$CascadiaFont"
```

2. Downloading the Cascadia Code font file

```
Invoke-WebRequest -Uri $CascadiaPath -OutFile $CascadiaFile
```

3. Installing the Cascadia Code font

```
$FontShellApp = New-Object -Com Shell.Application
$FontShellNamespace = $FontShellApp.Namespace(0x14)
$FontShellNamespace.CopyHere($CascadiaFile, 0x10)
```

4. Restarting VS Code

   Click on the shortcut in the taskbar

How it works...

In *step 1*, you discover the latest version of the Cascadia Code font, and in *step 2*, you download this font.

In *step 3*, you install the font into Windows. Since Windows does not provide any cmdlets to perform the font installation, you rely on the older Windows `Shell.Application` COM object.

In *step 4*, you restart VS Code and note that the new font now looks like this:

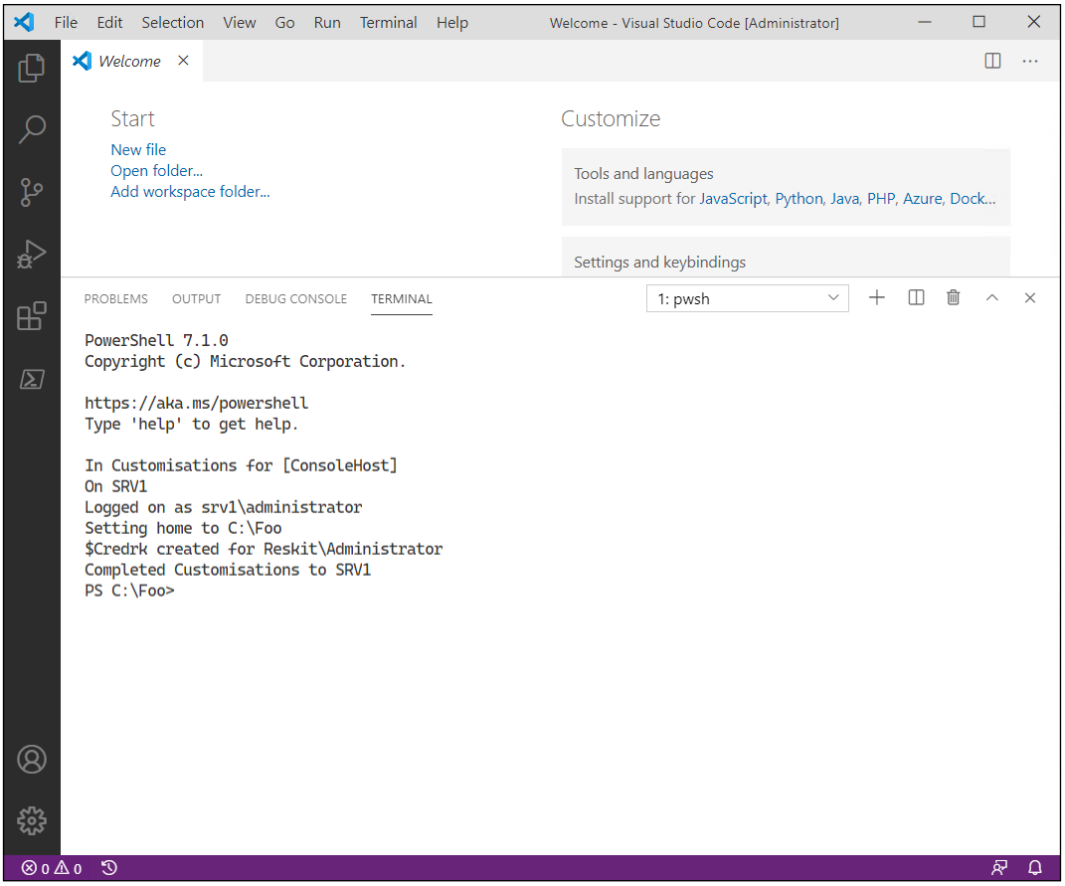


Figure 1.29: Looking at the new font

There's more...

Like so many things in customizing PowerShell 7, you have a wide choice of fonts to use with VS Code. The Cascadia Code font is a good, clear font and many like it. You can always change it if you do not like it.

Exploring PSReadLine

`PSReadLine` is a PowerShell module that provides additional console editing features within both PowerShell 7 and Windows PowerShell. The module provides a command-line editing experience that is on par with the best of the Linux command shells (such as Bash).

When you type into a PowerShell console, `PSReadLine` intercepts your keystrokes to provide syntax coloring, simple syntax error notification, and a great deal more. `PSReadLine` enables you to customize your environment to suit your personal preferences. Some key features of the module include:

- Syntax coloring of the command-line entries
- Multi-line editing
- History management
- Customizable key bindings
- Highly customizable

For an overview of `PSReadLine`, see `https://docs.microsoft.com/powershell/module/psreadline/about/about_psreadline`. And for more details, you can view the `PSReadLine` GitHub README file: `https://github.com/PowerShell/PSReadLine/blob/master/README.md`.

An important issue surrounds the naming of this module. The original name of the module was `PSReadline`. At some point, the module's developers changed the name of the module to `PSReadLine` (capitalizing the L character in the module name).

The `PSReadLine` module ships natively with PowerShell 7. At startup, in both the console and VS Code, PowerShell imports `PSReadLine` so it is ready for you to use. You can also use `PSReadLine` in Windows PowerShell. To simplify the updating of this module's help, consider renaming the module to capitalize the L in `PSReadLine`.

The `PSReadLine` module is a work in progress. Microsoft incorporated an early version of this module within Windows PowerShell v3. Windows PowerShell 5.1 (and the ISE) inside Windows Server ships with `PSReadLine` v2. If you are still using earlier versions of Windows PowerShell, you can download and utilize the latest version of `PSReadLine`.

The module has improved and, with the module's v2 release, some changes were made that are not backward-compatible. Many blog articles, for example, use the older syntax for `Set-PSReadLineOption`, which fails with version 2 (and later) of the module. You may still see the old syntax if you use your search engine to discover examples. Likewise, some of the examples in this recipe fail should you run them utilizing `PSReadline` v1 (complete with the old module name's spelling).

Getting ready

This recipe uses SRV1 after you have installed PowerShell 7, VS Code, and the Cascadia Code font (and customized your environment).

How to do it...

1. Getting the commands in the PSReadLine module

```
Get-Command -Module PSReadLine
```

2. Getting the first 10 PSReadLine key handlers

```
Get-PSReadLineKeyHandler |
  Select-Object -First 10 |
    Sort-Object -Property Key |
      Format-Table -Property Key, Function, Description
```

3. Counting the unbound key handlers

```
$Unbound = (Get-PSReadLineKeyHandler -Unbound).count
"$Unbound unbound key handlers"
```

4. Getting the PSReadLine options

```
Get-PSReadLineOption
```

5. Determining the VS Code theme name

```
$Path       = $Env:APPDATA
$CP         = '\Code\User\Settings.json'
$JsonConfig = Join-Path  $Path -ChildPath $CP
$ConfigJSON = Get-Content $JsonConfig
$Theme = $ConfigJson |
           ConvertFrom-Json |
             Select-Object -ExpandProperty 'workbench.colorTheme'
```

6. Changing the color scheme if the theme name is Visual Studio Light

```
If ($Theme -eq 'Visual Studio Light') {
  Set-PSReadLineOption -Colors @{
    Member    = "`e[33m"
    Number    = "`e[34m"
    Parameter = "`e[35m"
    Command   = "`e[34m"
  }
}
```

How it works...

In *step 1*, you view the commands in the `PSReadLine` module, which looks like this:

```
PS C:\Foo> # 1. Getting commands in the PSReadLine module
PS C:\Foo> Get-Command -Module PSReadLine

CommandType  Name                           Version  Source
-----------  ----                           -------  ------
Function     PSConsoleHostReadLine          2.1.0    PSReadLine
Cmdlet       Get-PSReadLineKeyHandler       2.1.0    PSReadLine
Cmdlet       Get-PSReadLineOption           2.1.0    PSReadLine
Cmdlet       Remove-PSReadLineKeyHandler    2.1.0    PSReadLine
Cmdlet       Set-PSReadLineKeyHandler       2.1.0    PSReadLine
Cmdlet       Set-PSReadLineOption           2.1.0    PSReadLine
```

Figure 1.30: Viewing commands in the PSReadLine module

In *step 2*, you display the first 10 `PSReadLine` key handlers currently in use, which looks like this:

```
PS C:\Foo> # 2. Getting the first 10 PSReadLine key handlers
PS C:\Foo> Get-PSReadLineKeyHandler |
            Select-Object -First 10
              Sort-Object -Property Key |
                Format-Table -Property Key, Function, Description


Basic editing functions
=======================

Key            Function          Description
---            --------          -----------
Enter          AcceptLine        Accept the input or move to the next line if input is missing a closing
                                 token.
Shift+Enter    AddLine           Move the cursor to the next line without attempting to execute the input
Backspace      BackwardDeleteChar Delete the character before the cursor
Ctrl+h         BackwardDeleteChar Delete the character before the cursor
Ctrl+Home      BackwardDeleteLine Delete text from the cursor to the start of the line
Ctrl+Backspace BackwardKillWord  Move the text from the start of the current or previous word to the
                                 cursor to the kill ring
Ctrl+w         BackwardKillWord  Move the text from the start of the current or previous word to the
                                 cursor to the kill ring
Ctrl+C         Copy              Copy selected region to the system clipboard.  If no region is selected,
                                 copy the whole line
Ctrl+c         CopyOrCancelLine  Either copy selected text to the clipboard, or if no text is selected,
                                 cancel editing the line with CancelLine.
Ctrl+x         Cut               Delete selected region placing deleted text in the system clipboard
```

Figure 1.31: Displaying the first 10 PSReadLine key handlers

`PSReadLine` provides over 160 internal functions to which you can bind a key combination. In *step 3*, you discover how many functions are currently unbound, which looks like this:

```
PS C:\Foo> # 3. Discovering a count of unbound key handlers
PS C:\Foo> $Unbound = (Get-PSReadLineKeyHandler -Unbound).count
PS C:\Foo> "$Unbound unbound key handlers"
116 unbound key handlers
```

Figure 1.32: Discovering the count of unbound key handlers

In *step 4,* you view the `PSReadLine` options as you can see here:

```
PS C:\Foo> # 4. Getting the PSReadLine options
PS C:\Foo> Get-PSReadLineOption

EditMode                            : Windows
AddToHistoryHandler                 : System.Func`2[System.String,System.Object]
HistoryNoDuplicates                 : True
HistorySavePath                     : C:\Users\Administrator\AppData\Roaming\Microsoft\Windows\
                                      PowerShell\PSReadLine\Visual Studio Code Host_history.txt
HistorySaveStyle                    : SaveIncrementally
HistorySearchCaseSensitive          : False
HistorySearchCursorMovesToEnd       : False
MaximumHistoryCount                 : 4096
ContinuationPrompt                  : >>
ExtraPromptLineCount                : 0
PromptText                          : {> }
BellStyle                           : Audible
DingDuration                        : 50
DingTone                            : 1221
CommandsToValidateScriptBlockArguments : {ForEach-Object, %, Invoke-Command, icm, Measure-Command, New-Module,
                                      nmo, Register-EngineEvent, Register-ObjectEvent, Register-WMIEvent,
                                     Set-PSBreakpoint, sbp, Start-Job, sajb, Trace-Command, trcm,
                                     Use-Transaction,Where-Object, ?, where}
CommandValidationHandler            :
CompletionQueryItems                : 100
MaximumKillRingCount                : 10
ShowToolTips                        : True
ViModeIndicator                     : None
WordDelimiters                      : ;:,.[]{}()/\|^&*-=+'"–—
AnsiEscapeTimeout                   : 100
CommandColor                        : "`e[34m"
CommentColor                        : "`e[32m"
ContinuationPromptColor             : "`e[37m"
DefaultTokenColor                   : "`e[37m"
EmphasisColor                       : "`e[96m"
ErrorColor                          : "`e[91m"
KeywordColor                        : "`e[92m"
MemberColor                         : "`e[33m"
NumberColor                         : "`e[34m"
OperatorColor                       : "`e[90m"
ParameterColor                      : "`e[35m"
SelectionColor                      : "`e[30;47m"
StringColor                         : "`e[36m"
TypeColor                           : "`e[37m"
VariableColor                       : "`e[92m
```

Figure 1.33: Getting the PSReadLine options

In *step 5* and *step 6*, you determine the current VS Code theme name and update the colors used by `PSReadLine` for better clarity. These two steps produce no output, although you should see the colors of specific PowerShell syntax tokens change (run `Get-PSReadLineOption` to see the changes).

There's more...

In *step 1*, you view the commands inside the `PSReadLine` module. The `PSConsoleHostReadLine` function is the entry point to this module's functionality. When the PowerShell console starts, it loads this module and invokes the `PSConsoleHostReadLine` function.

One small downside to `PSReadLine` is that it does not work well with all Windows screen reader programs. For accessibility reasons, if the PowerShell console detects you are using any screen reader program, the console startup process does not load `PSReadLine`. You can always load `PSReadLine` manually either in the console or by updating your startup profile(s).

In *step 2*, you view the key handlers currently used by `PSReadLine`. Each key handler maps a keystroke combination (for example, *Ctrl + L*) to one of over 160 internal `PSReadLine` functions. By default, *Ctrl + L* maps to the `PSReadLine` `ClearScreen` function – when you type that key sequence, `PSReadLine` clears the screen in the console.

In *step 5* and *step 6*, you detect the VS Code theme name and adjust the color scheme to match your preferences. To some degree, the `PSReadLine` and VS Code color themes can result in hard-to-read color combinations. If that is the case, you can persist the specific settings into your VS Code profile file and change the color scheme to meet your tastes.

2

Introducing PowerShell 7

This chapter covers the following recipes:

- Exploring new operators
- Exploring parallel processing with `ForEach-Object`
- Exploring Improvements in `Foreach` and `ForEach-Object`
- Improvements in `Test-Connection`
- Using `Select-String`
- Exploring the error view and `Get-Error`
- Exploring experimental features

Introduction

In *Chapter 1*, *Installing and Configuring PowerShell 7.1*, you installed and configured PowerShell 7, along with VS Code and a new font. In this chapter, we look at PowerShell 7 and how it differs from Windows PowerShell. The recipes in this chapter illustrate some of the important new features that come with PowerShell 7.

Now that PowerShell is cross-platform, it has a new, expanded audience, one with a background in Linux shells such as Bash. With PowerShell 7, the PowerShell team added several new operators that improved parity with other shells and made life that little bit easier for IT pros.

With the move to open source, the PowerShell code was open to inspection by the community. Many talented developers were able to make improvements to performance and functionality. One example is how PowerShell performs iteration using `ForEach` and `ForEach-Object`. In Windows PowerShell, the `ForEach` syntax item and the `Foreach-Object` command allowed you to process collections of objects. With Windows PowerShell, each iteration through a collection was serial, which could result in long script runtimes. PowerShell 7 introduces an improvement in the `ForEach-Object` command that enables you to run iterations in parallel. This review has led to a reduction in the overhead of using these popular language features, thereby speeding up production scripts.

Another improvement is the revised `Test-Connection`, a command you use to test a network connection with a remote system. `Test-Connection`, in PowerShell 7, not only does more, but is faster than with Windows PowerShell.

Error reporting in Windows PowerShell was excellent: clear and generally actionable error messages with details of exactly where the error occurred. In PowerShell 7, you now get, by default, a concise view of an error without all the extra text that was often of little value. As always, you can revert to less concise messages if you choose. In the *Exploring the error view and Get-Error* recipe, you see how error reporting (in Windows PowerShell) becomes better with PowerShell 7.

In the final recipe of the chapter, we take a look at some of the experimental features that can be enabled in PowerShell 7.

Exploring new operators

Operators are symbols or combinations of keystrokes that PowerShell recognizes and assigns some meaning to. PowerShell uses the + operator to mean addition, either arithmetic addition or string addition/concatenation. Most of the PowerShell operators were defined with Windows PowerShell V1.

PowerShell 7 now implements some new operators, including the following:

- Pipeline chain operators: `||` and `&&`
- Null-coalescing operator: `??`
- Null-coalescing assignment operator: `??=`
- Experimental null conditional member access operators: `?.` and `?[]`
- Background processing operator: `&`
- Ternary operator: `? <if-true> : <if-false>`

You see examples of these operators in this recipe.

Getting ready

This recipe uses SRV1, a Windows Server 2020 host. You have installed and configured PowerShell 7 and VS Code. You run this, and all remaining recipes in this book, in either a PowerShell 7 console or VS Code.

How to do it...

1. Using PowerShell 7 to check results traditionally

```
Write-Output 'Something that succeeds'
if ($?) {Write-Output 'It worked'}
```

2. Checking results with the pipeline operator &&

```
Write-Output 'Something that succeeds' && Write-Output 'It worked'
```

3. Using the pipeline chain operator ||

```
Write-Output 'Something that succeeds' ||
  Write-Output 'You do not see this message'
```

4. Defining a simple function

```
function Install-CascadiaPLFont{
  Write-Host 'Installing Cascadia PL font...'
}
```

5. Using the || operator

```
$OldErrorAction        = $ErrorActionPreference
$ErrorActionPreference = 'SilentlyContinue'
Get-ChildItem -Path C:\FOO\CASCADIAPL.TTF ||
   Install-CascadiaPLFont
$ErrorActionPreference = $OldErrorAction
```

6. Creating a function to test null handling

```
Function Test-NCO {
  if ($args -eq '42') {
    Return 'Test-NCO returned a result'
  }
}
```

7. Testing null results traditionally

```
$Result1 = Test-NCO    # no parameter
if ($null -eq $Result1) {
    'Function returned no value'
} else {
    $Result1
}
$Result2 = Test-NCO 42  # using a parameter
if ($null -eq $Result2) {
    'Function returned no value'
} else {
    $Result2
}
```

8. Testing using the null-coalescing operator `??`

```
$Result3 =  Test-NCO
$Result3 ?? 'Function returned no value'
$Result4 =  Test-NCO 42
$Result4 ?? 'This is not output, but result is'
```

9. Demonstrating the null conditional assignment operator

```
$Result5 = Test-NCO
$Result5 ?? 'Result is null'
$Result5 ??= Test-NCO 42
$Result5
```

10. Running a method on a null object traditionally

```
$BitService.Stop()
```

11. Using the null conditional operator for a method

```
${BitService}?.Stop()
```

12. Testing null property name access

```
$x = $null
${x}?.Propname
$x = @{Propname=42}
${x}?.Propname
```

13. Testing array member access of a null object

```
$y = $null
${y}?[0]
$y = 1,2,3
${y}?[0]
```

14. Using the background processing operator &

```
Get-CimClass -ClassName Win32_Bios &
```

15. Waiting for the job to complete

```
$JobId = (Get-Job | Select-Object -Last 1).Id
Wait-Job -id $JobId
```

16. Viewing the output

```
$Results = Receive-Job -Id $JobId
$Results | Format-Table
```

17. Creating an object without using the ternary operator

```
$A = 42; $B = (42,4242) | Get-Random
$RandomTest = ($true, $false) | Get-Random
if ($A -eq $B) {
  $Property1 = $true
} else {
  $Property1 = $false
}
if ($RandomTest) {
  $Property2 = "Hello"
} else {
  $Property2 = "Goodbye"
}
[PSCustomObject]@{
  "Property1" = $Property1
  "Property2" = $Property2
}
```

18. Creating an object using the ternary operator

```
[PSCustomObject]@{
    "Property1" = (($A -eq $B) ? $true : $false)
    "Property2" = (($RandomTest) ? "Hello" : "Goodbye")
}
```

How it works...

In *step 1*, you write output, which succeeds. Then you test the value of `$?` to determine whether that previous step did, in fact, succeed. The output is as follows:

```
PS C:\Foo> # 1. Checking results traditionally
PS C:\Foo> Write-Output 'Something that succeeds'
Something that succeeds
PS C:\Foo> if ($?) {Write-Output 'It worked'}
It worked
```

Figure 2.1: Checking results traditionally

In *step 2*, you use the `&&` operator to check that a preceding command finished without an error. The output looks like this:

```
PS C:\Foo> # 2. Checking results with pipeline operator &&
PS C:\Foo> Write-Output 'Something that succeeds' && Write-Output 'It worked'
Something that succeeds
It worked
```

Figure 2.2: Checking results with the pipeline operator

The pipeline chain operator, `||`, tells PowerShell to run the commands after the operator if the preceding command fails (in effect, the opposite to `&&`). In *step 3*, you see the operator in use, with output like this:

```
PS C:\Foo> # 3. Using pipeline chain operator  ||
PS C:\Foo> Write-Output 'Something that succeeds' ||
             Write-Output 'You do not see this message'
Something that succeeds
```

Figure 2.3: Using the pipeline chain operator

In *step 4*, you define a function. Defining the function produces no output. This function writes output to simulate the installation of the Cascadia Code PL font.

In *step 5*, you check to see whether the TTF file exists, and if not, you call the `Install-CascadiaPLFont` function to simulate installing the font. By piping the output from `Get-ChildItem` to `Out-Null`, you avoid the actual output from `Get-ChildItem`, and if the file does not exist, you call the `Install-CascadiaPLFont` function. The output of this snippet looks like this:

```
PS C:\Foo> # 5. Using the || operator
PS C:\Foo> $OldErrorAction        = $ErrorActionPreference
PS C:\Foo> $ErrorActionPreference = 'SilentlyContinue'
PS C:\Foo> Get-ChildItem -Path C:\FOO\CASCADIAPL.TTF  ||
             Install-CascadiaPLFont
Installing Cascadia PL font..
PS C:\Foo> $ErrorActionPreference = $OldErrorAction
```

Figure 2.4: Using the || operator and installing the Cascadia font

To illustrate the handling of null results from a function, in *step 6*, you create a function that either returns nothing (if you call the function with no parameters) or a string value (if you call it specifying a parameter). This function illustrates how you can handle a function that returns null. This step produces no output.

In *step 7*, you illustrate the traditional handling of a function that returns null. You call the function, first without a parameter, that returns no result and then with a value that does return a value. You then test to see whether the function returned an actual value in each case, which looks like this:

```
PS C:\Foo> # 7. Test null results traditionally
PS C:\Foo> $Result1 = Test-NCO    # no parameter
PS C:\Foo> if ($null -eq $Result1) {
             'Function returned no value'
           } else {
             $Result1
           }
Function returned no value

PS C:\Foo> $Result2 = Test-NCO 42  # using a parameter
            if ($null -eq $Result2) {
              'Function returned no value'
            } else {
               $Result2
            }
Test-NCO returned a result
```

Figure 2.5: Testing null results traditionally

When you use the null-coalescing operator (`??`) between two operands, the operator returns the value of its left-hand operand if it isn't `null`; otherwise, it evaluates the right-hand operand and returns the results. In *step 8*, you call the `Test-NCO` function and check whether the function returns a value, which looks like this:

```
PS C:\Foo> # 8. Testing using null coalescing operator ??
PS C:\Foo> $Result3 =  Test-NCO
PS C:\Foo> $Result3 ?? 'Function returned no value'
Function returned no value
PS C:\Foo> $Result4 =  Test-NCO 42
PS C:\Foo> $Result4 ?? 'This is not output, but result is'
Test-NCO returned a result
```

Figure 2.6: Testing using the null-coalescing operator

You use the null conditional assignment operator, `??=`, to assign a value to a variable if that variable is currently null, as you can see in *step 9*, the output from which looks like this:

```
PS C:\Foo> # 9. Demonstrating null conditional assignment operator
PS C:\Foo> $Result5 = Test-NCO
PS C:\Foo> $Result5 ?? 'Result is is null'
Result is is null
PS C:\Foo> $Result5 ??= Test-NCO 42
PS C:\Foo> $Result5
Test-NCO returned a result
```

Figure 2.7: Using the null conditional assignment operator

One common issue often seen in the various PowerShell support forums arises when you attempt to invoke a method on an object that is null. You might have used an expression or a command to attempt to return a value (for example, all AD users in the Marin County office) and that returns a null values. In *step 10*, you attempt to invoke the `Stop()` method on the `$BitService` object. Since you have not assigned a value to `$BitService`, you see the result (an error, `You cannot call a method on a null-valued expression`). The traditional method of displaying errors looks like this:

```
PS C:\Foo> # 10. Running a method on a null object traditionally
PS C:\Foo> $BitService.Stop()
InvalidOperation:
Line |
   2 |  $BitService.Stop()
     |  ~~~~~~~~~~~~~~~~~~
     | You cannot call a method on a null-valued expression.
```

Figure 2.8: Running a method on a null object traditionally

By using the null conditional operator, you can run the `Stop()` method if the `$BitService` variable is non-null, but skip calling the method if the variable is null. In effect, what you are doing in *step 11* is calling the `Stop()` method if the variable is non-null, and doing nothing otherwise. Because the variable does not have a value, this step does nothing (and produces no output).

When a variable is null, whether due to an error in your scripts or because a command returns a null instead of an actual value, accessing property names can also cause errors. The output of *step 12* looks like this:

```
PS C:\Foo> # 12. Testing null property name access
PS C:\Foo> $x = $null
PS C:\Foo> ${x}?.Propname
PS C:\Foo> $x = @{Propname=42}
PS C:\Foo> ${x}?.Propname
42
```

Figure 2.9: Testing null property name access

You can also encounter issues with null objects when you attempt to access an array member of an object that may or may not exist. In *step 13*, you attempt to access an array member of an array that does not exist, followed by one that does exist. The output from this step looks like this:

```
PS C:\Foo> # 13. Testing array member access of a null object
PS C:\Foo> $Y = $null
PS C:\Foo> ${Y}?[0]
PS C:\Foo> $Y = 1,2,3
PS C:\Foo> ${Y}?[0]
1
```

Figure 2.10: Testing array member access of a null object

In *step 14*, you investigate the use of the background processing operator, `&`. The idea is that you append this character to the end of a command or script, and PowerShell runs that code in the background. The output from this step looks like this:

```
PS C:\Foo> # 14. Using background processing operator &
PS C:\Foo> Get-CimClass -ClassName Win32_Bios &

Id  Name  PSJobTypeName  State    HasMoreData  Location   Command
--  ----  -------------  -----    -----------  --------   -------
1   Job1  BackgroundJob  Running  True         localhost  Microsoft.PowerShell.Man…
```

Figure 2.11: Using the background processing operator

In *step 15*, you wait for the job you created in *step 14* to complete, which looks like this:

```
PS C:\Foo> # 15. Waiting for the job to complete
PS C:\Foo> $JobId = (Get-Job | Select-Object -Last 1).Id
PS C:\Foo> Wait-Job -Id $JobId

Id  Name  PSJobTypeName  State     HasMoreData  Location   Command
--  ----  -------------  -----     -----------  --------   -------
1   Job1  BackgroundJob  Completed True         localhost  Microsoft.PowerShell.Man…
```

Figure 2.12: Waiting for the job to complete

After the job has completed, in *step 16*, you receive and display the job's output, which looks like this:

```
PS C:\Foo> # 16. Viewing the output
PS C:\Foo> $Results = Receive-Job -Id $JobId
PS C:\Foo> $Results

   NameSpace:
CimClassName   CimClassMethods  CimClassProperties
------------   ---------------  ------------------
Win32_BIOS     {}               {Caption, Description, InstallDate, Name…
```

Figure 2.13: Displaying the job's output

In *step 17*, you create an object using a more traditional approach. This step creates a random value for two properties. Then, you create an object using the values of these two properties. The output from this step looks like this:

```
PS C:\Foo> # 17. Creating an object without using the ternary operator
PS C:\Foo> $A = 42; $B = (42,4242) | Get-Random
PS C:\Foo> $RandomTest = ($true, $false) | Get-Random
PS C:\Foo> if ($A -eq $B) {
             $Property1 = $true
           } else {
             $Property1 = $false
           }
PS C:\Foo> if ($RandomTest) {
             $Property2 = 'Hello'
             } else {
             $Property2 = 'Goodbye'
           }
PS C:\Foo> [PSCustomObject]@{
             "Property1" = $Property1
             "Property2" = $Property2
           }

Property1 Property2
--------- ---------
    False Hello
```

Figure 2.14: Creating an object without using the ternary operator

In *step 18*, you use the new PowerShell 7 ternary operator. This operator tests a condition and runs different code depending on whether the result is true or false. This is very similar to what you saw in *step 17*, but in a lot fewer lines of code. The output of this step looks like this:

```
PS C:\Foo> # 18. Creating an object using the ternary operator
PS C:\Foo> [PSCustomObject]@{
             "Property1" = (($A -eq $B) ? $true : $false)
             "Property2" = (($RandomTest) ? 'Hello' : 'Goodbye')
           }

Property1 Property2
--------- ---------
    False Hello
```

Figure 2.15: Creating an object using the ternary operator

There's more...

In *step 4*, the function you create simulates the installation of a font if the font does not exist. The font file, `CASCADIAPL.TTF`, is a TrueType font file for the Cascadia Code Powerline font. This font is the Cascadia Code font you installed in *Chapter 1*, *Installing and Configuring PowerShell 7.1*, with the addition of symbols for Powerline. For more details on this font, see `https://www.hanselman.com/blog/PatchingTheNewCascadiaCodeToIncludePowerlineGlyphsAndOtherNerdFontsForTheWindowsTerminal.aspx`.

In *step 5*, you simulate the installation of the font if it does not already exist. When you check to test whether the TTF file currently exists, the default setting for `$ErrorActionPreference` (`Continue`) means you see an error message if the file does not exist. By default, when `Get-ChildItem` checks to see whether the file exists, it generates an error message if the file does not exist. One approach to avoiding this error message is to set the value of `$ErrorActionPreference` to `SilentlyContinue` and, after ensuring that the font file now exists, set it back to the default value. The syntax is a bit convoluted unless you are familiar with it; this may be another case where not using these operators, and using the Windows PowerShell approach instead, might make the script easier to read and understand.

In *steps 11* and *12*, you attempt to access a property from an object. The assumption here is that you only want to invoke the method or access a property value if the object exists and you do not care otherwise. Thus, if `$BitService` has a value, you call the `Stop()` method, otherwise the code carries on without the script generating errors. This approach is great from the command line, but in production scripts, the approach could mask other underlying issues. As with all PowerShell features, you have to use null handling with due care and attention.

With *step 14*, you tell PowerShell to run a command as a background job by appending the `&` character to the command. Using this operator is a more straightforward way to invoke the command as a job than by calling `Invoke-Command` and specifying the command using the `-ScriptBlock` or `-Script` parameters.

In *steps 15* and *16*, you use `Get-Job`, `Wait-Job`, and `Receive-Job` to wait for the last job run and get the output. One downside to not using `Start-Job` to create a background job is that you cannot specify a job name. That means using the technique shown in *step 15* to obtain the job and job results for the job created in *step 14*. Using that technique is thus more useful from the command line than in a production script.

In *step 17*, you create an object using older Windows PowerShell syntax, whereas, in *step 18*, you use the ternary operator. As with other operators, use this with care, and if you use these operators in production code, make sure you document what you are doing.

In this recipe, you have seen the new operators added to PowerShell 7. Most of them provide a shortcut way to perform some operation or other, particularly at the command line.

Exploring parallel processing with ForEach-Object

Situations often arise where you want to run many commands in parallel. For example, you might have a list of computer names, and for each of those computers, you want to run a script on that computer. You might wish to verify the status and resource usage of various services on each computer. In this scenario, you might use `Get-Content` to get an array of computer names, and then use either `ForEach` or `ForEach-Object` to run the script on the computer. If there are 10 computers and the script takes 10 minutes, the total runtime is over 100 minutes.

With Windows PowerShell, the only built-in methods of running scripts in parallel were using background jobs or using workflows. With background jobs, you could create a set of jobs, each of which starts a script on a single computer. In that case, PowerShell runs each job in a separate process, which provides isolation between each job but is resource-intensive. The Windows PowerShell team added workflows with Windows PowerShell V4, which also allow you to run script blocks in parallel. However, workflows are not carried forward into PowerShell 7. Like other features no longer available in PowerShell 7, you can continue to use Windows PowerShell to run workflows and gradually convert them as and when appropriate.

An alternative to background jobs is to use the `ThreadJob` module you can download from the PowerShell gallery. For more details on this module, see its repository page at `https://github.com/PaulHigin/PSThreadJob`.

With PowerShell 7, the PowerShell team added an option to the `ForEach-Object` command to allow you to run script blocks in parallel. This option simplifies running script blocks or scripts, especially long-running ones, in parallel and avoids the need for third-party modules or having to deal with the complexity of workflows.

This recipe demonstrates running operations in parallel traditionally, using background jobs, and using `ForEach-Object -Parallel`.

Getting ready

You run this recipe on `SRV1` after you have installed PowerShell 7 and, optionally, VS Code.

How to do it...

1. Simulating a long-running script block

```
$SB1 = {
  1..3 | ForEach-Object {
    "In iteration $_"
    Start-Sleep -Seconds 5
  }
}
Invoke-Command -ScriptBlock $SB1
```

2. Timing the expression

```
Measure-Command -Expression $SB1
```

3. Refactoring into code that uses jobs

```
$SB2 = {
1..3 | ForEach-Object {
  Start-Job -ScriptBlock {param($X) "Iteration $X " ;
                          Start-Sleep -Seconds 5} -ArgumentList $_
}
Get-Job | Wait-Job | Receive-Job -Keep
}
```

4. Invoking the script block

```
Invoke-Command -ScriptBlock $SB2
```

5. Removing any old jobs and timing the script block

```
Get-Job | Remove-Job
Measure-Command -Expression $SB2
```

6. Defining a script block using `ForEach-Object -Parallel`

```
$SB3 = {
1..3 | ForEach-Object -Parallel {
               "In iteration $_"
               Start-Sleep -Seconds 5
        }
}
```

7. Executing the script block

```
Invoke-Command -ScriptBlock $SB3
```

8. Measuring the script block execution time

```
Measure-Command -Expression $SB3
```

9. Creating and running two short script blocks

```
$SB4 = {
    1..3 | ForEach-Object {
                   "In iteration $_"
            }
}
Invoke-Command -ScriptBlock $SB4

$SB5 = {
        1..3 | ForEach-Object -Parallel {
                       "In iteration $_"
            }
}
Invoke-Command -ScriptBlock $SB5
```

10. Measuring the execution time for both script blocks

```
Measure-Command -Expression $SB4
Measure-Command -Expression $SB5
```

How it works...

In *step 1*, you create and then invoke a script block. The script block simulates how you can run several long script blocks traditionally using the `ForEach-Object` cmdlet, with output like this:

```
PS C:\Foo> # 1. Simulating a long running script block
PS C:\Foo> $SB1 = {
             1..3 | ForEach-Object {
               "In iteration $_"
               Start-Sleep -Seconds 5
             }
           }
PS C:\Foo> Invoke-Command -ScriptBlock $SB1
In iteration 1
In iteration 2
In iteration 3
```

Figure 2.16: Simulating a long-running script block

In *step 2*, you determine how long it takes PowerShell to run this script block, with output like this:

```
PS C:\Foo> # 2. Timing the expression
PS C:\Foo> Measure-Command -Expression $SB1

Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 15
Milliseconds      : 29
Ticks             : 150299813
TotalDays         : 0.000173958116898148
TotalHours        : 0.00417499480555556
TotalMinutes      : 0.250499688333333
TotalSeconds      : 15.0299813
TotalMilliseconds : 15029.9813
```

Figure 2.17: Timing the expression

In *step 3*, you refactor the `$SB1` script block to use PowerShell background jobs. The script block runs the simulated long-running task using jobs and then waits for and displays the output from each job. The concept is that instead of doing each iteration serially, all the jobs run in parallel. Defining the function creates no output.

In *step 4*, you invoke the script block to view the results, which looks like this:

```
PS C:\Foo> # 4. Invoking the script block
PS C:\Foo> Invoke-Command -ScriptBlock $SB2

Id  Name   PSJobTypeName  State    HasMoreData  Location   Command
--  ----   -------------  -----    -----------  --------   -------
63  Job63  BackgroundJob  Running  True         localhost  param($X) "Iteration $X …
65  Job65  BackgroundJob  Running  True         localhost  param($X) "Iteration $X …
67  Job67  BackgroundJob  Running  True         localhost  param($X) "Iteration $X …
Iteration 1
Iteration 2
Iteration 3
```

Figure 2.18: Invoking the script block

In *step 5*, you remove any existing jobs and then re-run the updated script block. This step enables you to determine the runtime for the entire expression. The output of this step looks like this:

```
PS C:\Foo> # 5. Removing any old jobs and timing the script block
PS C:\Foo> Get-Job | Remove-Job
PS C:\Foo> Measure-Command -Expression $SB2

Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 6
Milliseconds      : 836
Ticks             : 68369244
TotalDays         : 7.91310694444444E-05
TotalHours        : 0.00189914566666667
TotalMinutes      : 0.11394874
TotalSeconds      : 6.8369244
TotalMilliseconds : 6836.9244
```

Figure 2.19: Removing any existing jobs and timing the script block

In *step 6*, you create another script block that uses the PowerShell 7 `ForEach-Object -Parallel` construct. When you define this script block, PowerShell creates no output.

In *step 7*, you run the script block, which looks like this:

```
PS C:\Foo> # 7. Executing the script block
PS C:\Foo> Invoke-Command -ScriptBlock $SB3
In iteration 1
In iteration 2
In iteration 3
```

Figure 2.20: Executing the script block created in step 6

In *step 8*, you time the execution of the script block, making use of the `ForEach-Object -Parallel` feature, which looks like this:

```
PS C:\Foo> 8. Measuring the script block execution time
PS C:\Foo> Measure-Command -Expression $SB3

Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 5
Milliseconds      : 149
Ticks             : 51490168
TotalDays         : 5.95951018518519E-05
TotalHours        : 0.00143028244444444
TotalMinutes      : 0.0858169466666667
TotalSeconds      : 5.1490168
TotalMilliseconds : 5149.0168
```

Figure 2.21: Timing the script block execution

In *step 9*, you define and then invoke two script blocks, which looks like this:

```
PS C:\Foo> # 9. Creating and running two short script blocks
PS C:\Foo> $SB4 = {
               1..3 | ForEach-Object {
                                "In iteration $_"
                           }
           }
PS C:\Foo> Invoke-Command -ScriptBlock $SB4
In iteration 1
In iteration 2
In iteration 3
PS C:\Foo> $SB5 = {
                  1..3 | ForEach-Object -Parallel {
                                           "In iteration $_"
                           }
           }
PS C:\Foo> Invoke-Command -ScriptBlock $SB5
In iteration 1
In iteration 2
In iteration 3
```

Figure 2.22: Creating and running two short script blocks

In the final step in this recipe, *step 10*, you measure the execution time of these two script blocks, which looks like this:

```
PS C:\Foo> # 10. Measuring execution time for both script blocks
PS C:\Foo> Measure-Command -Expression $SB4

Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 0
Milliseconds      : 2
Ticks             : 29910
TotalDays         : 3.46180555555556E-08
TotalHours        : 8.30833333333333E-07
TotalMinutes      : 4.985E-05
TotalSeconds      : 0.002991
TotalMilliseconds : 2.991


PS C:\Foo> Measure-Command -Expression $SB5

Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 0
Milliseconds      : 83
Ticks             : 837963
TotalDays         : 9.69864583333333E-07
TotalHours        : 2.327675E-05
TotalMinutes      : 0.001396605
TotalSeconds      : 0.0837963
TotalMilliseconds : 83.7963
```

Figure 2.23: Measuring execution time of the script blocks created in step 9

There's more...

In *steps 1* and *2*, you invoke a long-running task multiple times. As you can see from *Figure 2.17*, running these script blocks, one at a time, takes just over 15 seconds. In *step 5*, you see that by refactoring the long-running task into PowerShell background jobs, you reduce the runtime to 6.83 seconds. Finally, in *step 8*, you measure the elapsed runtime when you use `ForEach-Object -Parallel`, which is now a little over 5 seconds.

As this recipe shows, if you have independent script blocks, you can run them in parallel to reduce the overall runtime, in this case, from just over 15 seconds to just over 5. And the gains would have been even higher had you run the loop more than three times. Running the loop serially 10 times would have taken over 50 seconds, compared to just over 5 for `ForEach-Object -Parallel`.

However, there is a default limit of five script blocks that PowerShell can run simultaneously. You can use the `-ThrottleLimit` parameter to allow more or less than that default. One thing to note: if you attempt to run more parallel script blocks than you have processor cores, PowerShell just uses a processor core queue. This all takes time and would end up raising the overall runtime. The good news is that PowerShell handles all this, so if you run, say, 1,000 parallel script blocks on a system with 12 processor cores, PowerShell works as fast as your host computer allows.

It is also worth remembering that there is some overhead involved in `ForEach-Object -Parallel`. Under the hood, the command has to set up and then manage separate threads of execution. If the script block is very short, you can find that the overhead involved results in slower runtimes. In this case, the runtime went from 2.9 ms to 83.7 ms. The critical point here is that this construct is useful for non-trivial script blocks (or scripts) that you run in parallel. You benefit up to the number of cores you have available.

Another thing to note is that when you use the `ForEach-Object {script}` syntax, you are using a positional parameter (`-Process`). On the other hand, when you use the `-Parallel` parameter, the parameter value is the script block you wish PowerShell to run in parallel.

Improvements in ForEach and ForEach-Object

Windows PowerShell users are well versed in the use of both the `ForEach` statement and the `ForEach-Object` cmdlet. You can use both of these methods in your scripts to process collections, such as all the users in a specific Active Directory group, or the audio files in a file share. In PowerShell 7, both of these iteration methods are considerably faster.

Using either `ForEach` mechanism is a quick and easy way of processing a collection. One downside some IT pros may have noticed is that the overhead of `ForEach` processing in Windows PowerShell grows with the size of the collection. With small collection sizes, you are not likely to notice any difference. As the collection size grows, so does the overhead.

Getting ready

You run this recipe on SRV1 after you have installed PowerShell 7 and have created a console profile file. Run this recipe in an elevated console.

How to do it...

1. Creating a remoting connection to the localhost using Windows PowerShell

```
New-PSSession -UseWindowsPowerShell -Name 'WPS'
```

2. Getting a remoting session

```
$Session = Get-PSSession -Name 'WPS'
```

3. Checking the version of PowerShell in the remoting session

```
Invoke-Command -Session $Session  -ScriptBlock {$PSVersionTable}
```

4. Defining a long-running script block using `ForEach-Object`

```
$SB1 = {
  $Array  = (1..10000000)
  (Measure-Command {
    $Array | ForEach-Object {$_}}).TotalSeconds
}
```

5. Running the script block locally:

```
[gc]::Collect()
$TimeInP7 = Invoke-Command -ScriptBlock $SB1
"Foreach-Object in PowerShell 7.1: [{0:n4}] seconds" -f $TimeInP7
```

6. Running the script block in PowerShell 5.1

```
[gc]::Collect()
$TimeInWP  = Invoke-Command -ScriptBlock $SB1 -Session $Session
"ForEach-Object in Windows PowerShell 5.1: [{0:n4}] seconds" -f $TimeInWP
```

7. Defining another long-running script block using `ForEach`

```
$SB2 = {
    $Array  = (1..10000000)
    (Measure-Command {
      ForEach ($Member in $Array) {$Member}}).TotalSeconds
}
```

8. Running it locally in PowerShell 7

```
[gc]::Collect()
$TimeInP72 = Invoke-Command -ScriptBlock $SB2
"Foreach in PowerShell 7.1: [{0:n4}] seconds" -f $TimeInP72
```

9. Running it in Windows PowerShell 5.1

```
[gc]::Collect()
$TimeInWP2  = Invoke-Command -ScriptBlock $SB2 -Session $Session
"Foreach in Windows PowerShell 5.1: [{0:n4}] seconds" -f $TimeInWP2
```

How it works...

In *step 1*, you use `New-PSSession` to create a remoting session using a Windows PowerShell endpoint. This step produces output like this:

```
PS C:\Foo> # 1. Creating a remoting connection to the local host
PS C:\Foo> New-PSSession -UseWindowsPowerShell -Name 'WPS'

 Id Name   Transport ComputerName  ComputerType   State   ConfigurationName  Availability
 -- ----   --------- ------------  ------------   -----   -----------------  ------------
  2 WPS    Process   localhost     RemoteMachine  Opened                        Available
```

Figure 2.24: Creating a remoting connection to the localhost

In *step 2*, you get the session object representing the session you created in the previous step. This creates no output.

In *step 3*, you obtain the version of PowerShell that the remoting session is using to process commands, namely, Windows PowerShell 5.1. The output of this step looks like this:

```
PS C:\Foo> # 3. Checking the version of PowerShell in the remoting session
PS C:\Foo> Invoke-Command -Session $session  -ScriptBlock {$PSVersionTable}

Name                           Value
----                           -----
PSCompatibleVersions           {1.0, 2.0, 3.0, 4.0, 5.0, 5.1.20221.1000}
BuildVersion                   10.0.20221.1000
PSEdition                      Desktop
PSRemotingProtocolVersion      2.3
SerializationVersion           1.1.0.1
CLRVersion                     4.0.30319.42000
WSManStackVersion              3.0
PSVersion                      5.1.20221.1000
```

Figure 2.25: Checking the PowerShell version in the remoting session

In *step 4*, you create a script block, `$SB1`, which uses the `ForEach-Object` cmdlet to iterate over a large collection. This step creates no output.

You invoke the `$SB1` script block in the local session, in *step 5*. This step runs the script block in PowerShell 7. The output from this step looks like this:

```
PS C:\Foo> # 5. Running the script block locally
PS C:\Foo> [gc]::Collect()
PS C:\Foo> $TimeInP7 = Invoke-Command -ScriptBlock $SB1
PS C:\Foo> "Time in PowerShell 7.1        : {0:n4} seconds" -f $TimeInP7
Foreach-Object in PowerShell 7.1: [75.3361] seconds
```

Figure 2.26: Running the script block locally

With *step 6*, you run the `$SB1` script block in Windows PowerShell 5.1, which produces output like this:

```
PS C:\Foo> # 6. Running it in PowerShell 5.1
PS C:\Foo> [gc]::Collect()
PS C:\Foo> $TimeInWP  = Invoke-Command -ScriptBlock $SB1 -Session $Session
PS C:\Foo> "Time in Windows PowerShell 5.1 : {0:n3} seconds" -f $TimeInWP
ForEach-Object in Windows PowerShell 5.1: [103.9153] seconds
```

Figure 2.27: Running the script block in PowerShell 5.1

You next create a script block that makes use of the `ForEach` syntax item, in *step 7*, producing no output. You then run this second script block in PowerShell 7, in *step 8*, which produces output like this:

```
PS C:\Foo> # 8. Running it locally in PowerShell 7
PS C:\Foo> [gc]::Collect()
PS C:\Foo> $TimeInP72 = Invoke-Command -ScriptBlock $SB2
PS C:\Foo> "Foreach in PowerShell 7.1: [{0:n4}] seconds" -f $TimeInP72
Foreach in PowerShell 7.1: [8.1041] seconds
```

Figure 2.28: Running the script block locally in PowerShell 7

In the final step, *step 9*, you run `$SB1` in the remoting session created earlier (in other words, in Windows PowerShell 5.1), which produces output like this:

```
PS C:\Foo> # 9. Running it in Windows PowerShell 5.1
PS C:\Foo> [gc]::Collect()
PS C:\Foo> $TimeInWP2  = Invoke-Command -ScriptBlock $SB2 -Session $Session
PS C:\Foo> "Foreach in Windows PowerShell 5.1: [{0:n4}] seconds" -f $TimeInWP2
Foreach in Windows PowerShell 5.1: [13.6315] seconds
```

Figure 2.29: Running the script block in PowerShell 5.1

There's more...

In *step 1*, you create, implicitly, a remoting session to the localhost using a process transport that is much faster than the traditional remoting session using WinRM.

In *steps 5*, *6*, *8*, and *9*, you force .NET to perform a garbage collection. These are steps you can use to minimize the performance hits of running a script block in a remote session (in Windows PowerShell) and to reduce any impact of garbage collections while you are performing the tests in this recipe.

As you can see from the outputs, running `ForEach-Object` is much faster in PowerShell 7, as is running `ForEach` in PowerShell 7. Processing large collections of objects is a lot faster in PowerShell 7.

The improvements to loop processing that you can see in the recipe, combined with the use of `ForEach-Object -Parallel` you saw in *Exploring parallel processing with ForEach-Object*, provide an excellent reason to switch to PowerShell 7 for most operations.

The performance of iterating through large collections is complex. You can read an excellent article on this subject at `https://powershell.one/tricks/performance/pipeline`. This article also addresses the performance of using the pipeline versus using `ForEach` to iterate across collections in a lot more detail.

Improvements in Test-Connection

In Windows PowerShell, you could use the `Test-Connection` cmdlet as a replacement for the Win32 console command, `ping.exe`. One advantage of using the cmdlet was that the cmdlet returns objects that you can use more easily in scripting. You can use string hacking and regular expressions to extract the same data from the output of `ping.exe`, but that is a lot more work and results in scripts that are harder to read.

With Windows PowerShell 5.1, the `Test-Connection` command makes use of WMI. The command returns objects of the type `System.Management.ManagementObject#root\cimv2\Win32_PingStatus`. With PowerShell 7.1, the command no longer depends on WMI and returns objects of the type `Microsoft.PowerShell.Commands.TestConnectionCommand+PingStatus`. As a result of this change of object type, property names returned in PowerShell 7.1 differ from the properties returned in Windows PowerShell. Scripts that made use of some properties may not work properly without adjustment, in PowerShell 7, but that should not be a common issue.

Getting ready

You run this recipe on SRV1 after you have installed PowerShell 7 and created a console profile file.

How to do it...

1. Using `Test-Connection` with the `-Target` parameter

```
Test-Connection -TargetName www.packt.com -Count 1
```

2. Using `Test-Connection` with an IPv4 address

```
Test-Connection -TargetName www.packt.com -Count 1 -IPv4
```

3. Using `Resolve-DnsName` to resolve the destination address

```
$IPs = (Resolve-DnsName -Name Dns.Google -Type A).IPAddress
$IPs |
  Test-Connection -Count 1 -ResolveDestination
```

4. Resolving the destination and performing a traceroute

```
Test-Connection -TargetName 8.8.8.8 -ResolveDestination -Traceroute |
  Where-Object Ping -eq 1
```

5. Using infinite ping and stopping with *Ctrl-C*

```
Test-Connection -TargetName www.reskit.net -Repeat
```

6. Checking the speed of `Test-Connection` in PowerShell 7

```
Measure-Command -Expression {test-connection 8.8.8.8 -count 1}
```

7. Checking the speed of `Test-Connection` in Windows PowerShell

```
$Session = New-PSSession -UseWindowsPowerShell
Invoke-Command -Session $Session -Scriptblock {
    Measure-Command -Expression {
      Test-Connection -ComputerName 8.8.8.8 -Count 1}
}
```

How it works...

In *step 1*, you test the connection between SRV1 and our publisher's online website. The output of this command looks like this:

```
PS C:\Foo> # 1. Using Test-Connection with the -TargetName parameter
PS C:\Foo> Test-Connection -TargetName www.packt.com -Count 1

   Destination: www.packt.com

Ping Source   Address                 Latency BufferSize Status
                                         (ms)       (B)
---- ------   -------                 ------- ---------- ------
   1 SRV1     2606:4700:10::6816:43b4       6         32 Success
```

Figure 2.30: Using Test-Connection with the -TargetName parameter

If you have a computer with a working IPv6 address, `Test-Connection` prefers using IPv6, by default, as shown in the output from *step 1*. Should you want to test the IPv4 connection specifically, you can specify the `-IPv4` switch explicitly, as shown in *step 2*:

```
PS C:\Foo> # 2. Using Test-Connection with an IPv4 address
PS C:\Foo> Test-Connection -TargetName www.packt.com -Count 1 -IPv4

   Destination: www.packt.com

Ping Source  Address        Latency  BufferSize  Status
                               (ms)        (B)
---- ------  -------        -------  ----------  ------
   1 SRV1    104.22.67.180        6          32  Success
`
```

Figure 2.31: Using Test-Connection with an IPv4 address

In *step 3*, you use `Resolve-DnsName` cmdlet to determine the IPv4 address(es) for `Dns.Google`. This site is Google's free DNS service, which Google offers via two well-known IPv4 addresses (8.8.8.8 and 8.8.4.4), which you can see in the output from this step:

```
PS C:\Foo> # 3. Using Resolve-DnsName to resolve destination address
PS C:\Foo> $IPs = (Resolve-DnsName -Name Dns.Google -Type A).IPAddress
PS C:\Foo> $IPs |
             Test-Connection -Count 1 -ResolveDestination

   Destination: dns.google

Ping Source   Address  Latency BufferSize  Status
                           (ms)       (B)
---- ------   -------  ------- ----------  ------
   1 SRV1     8.8.4.4        7         32  Success
   1 SRV1     8.8.8.8        6         32  Success
```

Figure 2.32: Using Resolve-DnsName to resolve the destination address

The `tracert.exe` Win32 console application allows you to trace the route between your host system and some external host. In *step 4*, you use `Test-Connection` to trace the route between `SRV1` and the computer at `8.8.8.8`. The output of this step looks like this:

```
PS C:\Foo> # 4. Resolving destination and trace route
PS C:\Foo> Test-Connection -TargetName 8.8.8.8 -ResolveDestination -Traceroute |
             Where-Object Ping -eq 1


   Target: dns.Google

Hop Hostname                 Ping Latency Status          Source       TargetAddress
                                  (ms)
--- --------                 ---- ------- ------          ------       -------------
  1 dg.reskit.org               1       0 Success         SRV1         8.8.8.8
  2 vt7.cor1.lond2.ptn.zen.n…   1       6 Success         SRV1         8.8.8.8
  3 ab-5.p1.ixn-lon.zen.net.…   1       * TimedOut        SRV1         8.8.8.8
  4 ae-5.p1.thn-lon.zen.net.…   1       * TimedOut        SRV1         8.8.8.8
  5 51-148-42-240.dsl.zen.co…   1       7 Success         SRV1         8.8.8.8
  6 72.14.223.28                1       7 Success         SRV1         8.8.8.8
  7 209.85.248.229              1       6 Success         SRV1         8.8.8.8
  8 216.239.57.119              1       6 Success         SRV1         8.8.8.8
  9 dns.Google                  1       6 Success         SRV1         8.8.8.8
```

Figure 2.33: Resolving the destination and trace route

With the `ping.exe` Win32 console application, you can specify the `-t` parameter to ping the target host continuously, which you can then stop by entering *Ctrl-C*. With `Test-Connection` in PowerShell 7, you can now use the `-Repeat` parameter to achieve the same outcome (and stop the test using *Ctrl-C*). You can see this in the output of *step 5*:

```
PS C:\Foo> # 5. Using infinite ping and stopping with Ctrl-C
PS C:\Foo> Test-Connection -TargetName www.reskit.net -Repeat

   Destination: www.reskit.net

Ping Source  Address         Latency BufferSize Status
                                (ms)       (B)
---- ------  -------         ------- ---------- ------
   1 SRV1    40.108.180.25        10         32 Success
   2 SRV1    40.108.180.25        10         32 Success
   3 SRV1    40.108.180.25        10         32 Success
   4 SRV1    40.108.180.25         9         32 Success
   5 SRV1    40.108.180.25        10         32 Success
   6 SRV1    40.108.180.25        10         32 Success
   7 SRV1    40.108.180.25         9         32 Success
```

Figure 2.34: Using infinite ping and stopping with Ctrl-C

In *steps 6* and *7*, you compare the speed of `Test-Connection` in Windows PowerShell and PowerShell 7. Running `Test-Connection` in PowerShell 7 looks like this:

```
PS C:\Foo> # 6. Checking speed of Test-Connection in PowerShell 7
PS C:\Foo> Measure-Command -Expression {
             Test-Connection -TargetName  8.8.8.8 -count 1}

Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 0
Milliseconds      : 12
Ticks             : 125724
TotalDays         : 1.45513888888889E-07
TotalHours        : 3.49233333333333E-06
TotalMinutes      : 0.00020954
TotalSeconds      : 0.0125724
TotalMilliseconds : 12.5724
```

Figure 2.35: Checking the speed of Test-Connection in PowerShell 7

In Windows PowerShell 5.1, the output is similar (although slower) and looks like this:

```
PS C:\Foo> # 7. Checking speed of Test-Connection in Windows PowerShell
PS C:\Foo> $Session = New-PSSession -UseWindowsPowerShell
PS C:\Foo> Invoke-Command -Session $Session -Scriptblock {
             Measure-Command -Expression {
               Test-Connection -ComputerName 8.8.8.8 -Count 1}
           }

Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 0
Milliseconds      : 195
Ticks             : 1951344
TotalDays         : 2.2585E-06
TotalHours        : 5.4204E-05
TotalMinutes      : 0.00325224
TotalSeconds      : 0.1951344
TotalMilliseconds : 195.1344
PSComputerName    : localhost
```

Figure 2.36: Checking the speed of Test-Connection in Windows PowerShell 5.1

There's more...

In the output from *step 1*, you can see that the results are formatted differently from Windows PowerShell, with improvements to the output.

Step 2 shows how, in PowerShell 7, you can use the `-IPv4` switch to use IPv4 explicitly. Similarly, if you want to use IPv6 specifically, you can use the `-IPv6` switch. Neither switch was available with Windows PowerShell.

In *step 3*, you determine the IP addresses for the host `Dns.Google` (8.8.8.8 and 8.8.4.4), which you then ping successfully. Note that this step both resolves the IP addresses into hostnames and performs the pings against each IP address. Google runs a free DNS service available to anyone on the internet. You can find out more at `https://developers.google.com/speed/public-dns`.

In *steps 6* and *7*, you compare the speed of the `Test-Connection` command between Windows PowerShell 5.1 and PowerShell 7.1, As you can see in the output from these steps, the command is considerably faster in PowerShell 7.

Using Select-String

The `Select-String` command, included with Windows PowerShell, has been improved in PowerShell 7. You use this command to search for strings either inside pipelined objects or within text files. This command is conceptually similar to the `grep` command in Linux. With PowerShell 7, the PowerShell team has added some excellent new features to this excellent command, which you look at in this recipe.

Getting ready

You run this recipe on `SRV1` after you have installed PowerShell 7 and Visual Studio Code, and once you have created a console profile file.

How to do it...

1. Getting a file of text to work with

```
$Source      = 'https://www.gutenberg.org/files/1661/1661-0.txt'
$Destination = 'C:\Foo\Sherlock.txt'
Start-BitsTransfer -Source $Source -Destination $Destination
```

2. Getting the book's contents

```
$Contents = Get-Content -Path $Destination
```

3. Checking the length of *The Adventures of Sherlock Holmes*

```
"The book is {0} lines long" -f $Contents.Length
```

4. Searching for "Watson" in the book's contents

```
$Match1 = $Contents | Select-String -Pattern 'Watson'
"Watson is found {0} times" -f $Match1.Count
```

5. Viewing the first few matches

```
$Match1 | Select-Object -First 5
```

6. Searching for "Dr. Watson" with a regular expression

```
$Contents | Select-String -Pattern 'Dr\. Watson'
```

7. Searching for "Dr. Watson" using a simple match

```
$Contents | Select-String -Pattern 'Dr. Watson' -SimpleMatch
```

8. Viewing the output when searching from files

```
Get-ChildItem -Path $Destination |
  Select-String -Pattern 'Dr\. Watson'
```

How it works...

In this recipe, you look at how you can use the `Select-String` cmdlet included with PowerShell 7. To investigate this cmdlet, you first download a text file.

In *step 1*, you use the `Start-BitsTransfer` command to download the text for a book, *The Adventures of Sherlock Holmes*, by Sir Arthur Conan Doyle, from the Project Gutenberg website. This step produces no output.

In *step 2*, you get the text from this book, which you store in the `$Contents` variable. This step produces no output. In *step 3*, you report the length of the book, which looks like this:

```
PS C:\Foo> # 3. Checking the length of The Adventures of Sherlock Holmes
PS C:\Foo> "The book is {0} lines long" -f $Contents.Length
The book is 12310 lines long
```

Figure 2.37: Checking the length of the book

In *step 4*, you search the book's contents to find all occurrences of "Watson", Sherlock Holmes' faithful companion. This results in 81 occurrences, and looks like this:

```
PS C:\Foo> # 4. Searching for "Watson" in book contents
PS C:\Foo> $Match1 = $Contents | Select-String -Pattern 'Watson'
PS C:\Foo> "Watson is found {0} times" -f $Match1.Count
Watson is found 81 times
```

Figure 2.38: Searching for occurrences of "Watson" in the book

In *step 5*, you use the `Select-Object` cmdlet to view the first five times the command finds the term "Watson" in the book's contents, which looks like this:

```
PS C:\Foo> # 5. Viewing first few matches
PS C:\Foo> $Match1 | Select-Object -First 5

"Wedlock suits you," he remarked. "I think, Watson, that you have put
fancy, Watson. And in practice again, I observe. You did not tell me
and fifty guineas apiece. There's money in this case, Watson, if there
Watson, who is occasionally good enough to help me in my cases. Whom
some good news for you. And good-night, Watson," he added, as the
```

Figure 2.39: Viewing the first five "Watson" matches in the book

With `Select-String`, you can specify a regular expression with which to match the contents. In *step 6*, you specify a regular expression pattern to search for the string "Dr. Watson". The output of this step looks like this:

```
PS C:\Foo> # 6. Searching for 'Dr. Watson' with a regular expression
PS C:\Foo> $Contents | Select-String -Pattern 'Dr\. Watson'

your narrative. I ask you not merely because my friend Dr. Watson has
my friend, Dr. Watson, before whom you can speak as freely as before
"This is my friend, Dr. Watson. He has been of most vital use to me in
Holmes. This is my intimate friend and associate, Dr. Watson, before
the basket-chair. This is my friend and colleague, Dr. Watson. Draw up
```

Figure 2.40: Searching for "Dr. Watson" with a regular expression pattern

As an alternative to using a regular expression to perform searching, `Select-String` also takes a simple match, as shown in the output of *step 7*:

```
PS C:\Foo> # 7. Searching for Dr. Watson using a simple match
PS C:\Foo> $Contents | Select-String -Pattern 'Dr. Watson' -SimpleMatch

your narrative. I ask you not merely because my friend Dr. Watson has
my friend, Dr. Watson, before whom you can speak as freely as before
"This is my friend, Dr. Watson. He has been of most vital use to me in
Holmes. This is my intimate friend and associate, Dr. Watson, before
the basket-chair. This is my friend and colleague, Dr. Watson. Draw up
```

Figure 2.41: Searching for "Dr. Watson" using a simple match

In the previous steps, you have used `Select-String` to search for the contents of a variable. Another valuable feature of `Select-String` is the ability to search for text in a file or even multiple files. You can see this in *step 8*, the output of which looks like this:

```
PS C:\Foo> # 8. Viewing output when searchng from files
PS C:\Foo> Get-ChildItem -Path $Destination |
             Select-String -Pattern 'Dr\. Watson'

Sherlock.txt:1196:your narrative. I ask you not merely because my friend Dr. Watson has
Sherlock.txt:2356:my friend, Dr. Watson, before whom you can speak as freely as before
Sherlock.txt:5337:"This is my friend, Dr. Watson. He has been of most vital use to me in
Sherlock.txt:6829:Holmes. This is my intimate friend and associate, Dr. Watson, before
Sherlock.txt:9081:the basket-chair. This is my friend and colleague, Dr. Watson. Draw up
```

Figure 2.42: Searching for "Dr. Watson" in the Sherlock.txt file

There's more...

In *steps 1* and *2*, you download a text file from Project Gutenberg, a free internet library of eBooks. This site contains a large number of free books in a variety of formats, including basic text. To find more free eBooks, visit the home page at `https://www.gutenberg.org/`, and to read more about the project, see `https://www.gutenberg.org/about/`.

An essential improvement to `Search-String` in PowerShell 7 is the highlighting of the selected string, as you can see in the outputs of *steps 5*, *6*, *7*, and *8*. From the command line, this makes viewing the output from `Select-String` much easier to consume. Also, the ability to search across multiple files, as shown in *step 8*, makes the `Select-String` cmdlet even more useful.

Exploring the error view and Get-Error

Since the very beginning, Windows PowerShell has done a great job in displaying the results of errors: a big blob of red text on a black background that contains full details about what went wrong. It was tremendous, but many new users found it a bit off-putting – there was too much information, some of which was not very useful in most cases.

PowerShell 7 now offers a more concise view of errors that reduces the amount of text and improves the format of the output. The result is shorter and more readable output. And, on those rare occasions when it might be necessary, you can Get-Error to get complete error details without having to parse through `$Error[0]`.

Getting ready

You run this recipe on `SRV1` after you have installed PowerShell 7 and/or Visual Studio Code, and once you have created a console profile file.

How to do it...

1. Creating a simple script

```
$SCRIPT = @'
  # divide by zero
  42/0
'@
$SCRIPTFILENAME = 'C:\Foo\ZeroDivError.ps1'
$SCRIPT | Out-File -Path $SCRIPTFILENAME
```

2. Running the script and seeing the default error view

```
& $SCRIPTFILENAME
```

3. Running the same line from the console

```
42/0
```

4. Viewing the `$ErrorView` variable

```
$ErrorView
```

5. Viewing the potential values of `$ErrorView`

```
$Type = $ErrorView.GetType().FullName
[System.Enum]::GetNames($Type)
```

6. Setting `$ErrorView` to `'NormalView'` and recreating the error

```
$ErrorView = 'NormalView'
& $SCRIPTFILENAME
```

7. Setting `$ErrorView` to `'CategoryView'` and recreating the error

```
$ErrorView = 'CategoryView'
& $SCRIPTFILENAME
```

8. Setting `$ErrorView` to its default value

```
$ErrorView = 'ConciseView'
```

How it works...

In *step 1*, you create a script that contains a (deliberate) divide-by-zero error. This step creates the file, but creates no other output.

In *step 2*, you run the script from within VS Code, and view the resulting error, which looks like this:

```
PS C:\Foo> # 2. Running the script and seeing the default error view
PS C:\Foo> & $SCRIPTFILENAME
RuntimeException: C:\Foo\ZeroDivError.ps1:2:3
Line |
   2 |    42/0
     |    ~~~~
     | Attempted to divide by zero.
```

Figure 2.43: Running the script and viewing the error

In *step 3*, you create a divide-by-zero error from the command line. The output from this step looks like this:

```
PS C:\Foo> # 3. Running the same line from the console
PS C:\Foo> 42/0
RuntimeException: Attempted to divide by zero.
```

Figure 2.44: Running the same line from the console

PowerShell 7 uses the built-in `$ErrorView` variable to hold the name of the error view PowerShell should use to display errors. In *step 4*, you view the current value of this variable, which looks like this:

```
PS C:\Foo> # 4. Viewing $ErrorView variable
PS C:\Foo> $ErrorView
ConciseView
```

Figure 2.45: Viewing the value of the $ErrorView variable

The `$ErrorView` variable can take one of three values, as you can see from the output of *step 5*:

```
PS C:\Foo> # 5. Viewing potential values of $ErrorView
PS C:\Foo> $Type = $ErrorView.GetType().FullName
PS C:\Foo> [System.Enum]::GetNames($Type)
NormalView
CategoryView
ConciseView
```

Figure 2.46: Viewing the potential values of $ErrorView

In *step 6*, you set the value of `$ErrorView` to display the error using the output generated by Windows PowerShell and then re-view the error, which looks like this:

```
PS C:\Foo> # 6. Setting $ErrorView to 'NormalView' and recreating the error
PS C:\Foo> $ErrorView = 'NormalView'
PS C:\Foo> & $SCRIPTFILENAME
Attempted to divide by zero.
At C:\Foo\ZeroDivError.ps1:2 char:3
+   42/0
+   ~~~~
+ CategoryInfo          : NotSpecified: (:) [], RuntimeException
+ FullyQualifiedErrorId : RuntimeException
```

Figure 2.47: Setting $ErrorView to NormalView and recreating the error

In *step 7*, you set `$ErrorView` to display the error using `CategoryView` and then recreate the error. The output from this step shows the category error view:

```
PS C:\Foo> # 7. Setting $ErrorView to 'CategoryView' and recreating the error
PS C:\Foo> $ErrorView = 'CategoryView'
PS C:\Foo> & $SCRIPTFILENAME
NotSpecified: (:) [], RuntimeException
```

Figure 2.48: Setting $ErrorView to CategoryView and recreating the error

In *step 8*, you reset the value of `$ErrorView` to the default value. This step creates no output.

There's more...

The concise error view you see in the output from *step 2* contains all the information from the standard view that you can see in the output from *step 7*, except for the omission of the error category information. And if you invoke the error directly from the command line, as shown in *step 3*, you see only the error message, which is easier on the eyes.

In *step 5*, you view the error category information. In most cases, this is not particularly useful.

In *step 8*, you reset the value of `$ErrorView`. Depending on what you are doing, this step may not be needed. You can just exit the PowerShell console (or VS Code), and the next time you start PowerShell, it resets the value back to the default (`ConciseView`). And if you should prefer the normal or category error views, you can always set a value to `$ErrorView` in your profile file.

Although not shown in this recipe, you can use the Get-Error cmdlet to show you complete error information about a specific error. For most IT Professionals, the basic error information provided by PowerShell 7 is more than adequate (and a great improvement over error output with Windows PowerShell).

Exploring experimental features

During the development of PowerShell Core and later with PowerShell 7, the PowerShell team have routinely added new features. Some of these new features could, at least in theory, break existing scripts and are called "experimental." PowerShell does not, by default, enable any of these features. As shown in this recipe, you must enable them explicitly. This approach to experimental features enables you to test these new features and provide the PowerShell team with feedback. Should you find a feature that breaks a script for you, disable it. If you turn on (or turn off) an experimental feature, you need to restart PowerShell 7.

In general, experimental features are not intended to be used in production since the experimental features, by design, can be breaking. Also, experimental features are not officially supported. That being said, so far, these features have been very stable and reliable.

In this recipe, you look at the experimental features available in PowerShell 7.1 as released. If you are using later versions (for example, a PowerShell 7.2 preview release), you may see different experimental features. For a fuller look at PowerShell's experimental features, see `https://docs.microsoft.com/powershell/module/microsoft.powershell.core/about/about_experimental_features`.

Getting ready

You run this recipe on `SRV1` after you install PowerShell 7 and/or Visual Studio Code, and once you have created a console profile file.

How to do it...

1. Discovering the experimental features

```
Get-ExperimentalFeature -Name * |
  Format-Table Name, Enabled, Description -Wrap
```

2. Examining the "command not found" result with no experimental features available

```
Foo
```

3. Enabling one experimental feature as the current user

```
Get-ExperimentalFeature -Name * |
  Select-Object -First 1 |
    Enable-ExperimentalFeature -Scope CurrentUser -Verbose
```

4. Enabling one experimental feature for all users

```
Get-ExperimentalFeature -Name * |
  Select-Object -Skip 1 -First 1 |
    Enable-ExperimentalFeature -Scope AllUsers -Verbose
```

5. Starting a new PowerShell console

   If you are using VS Code to run this recipe, enter *Ctrl* + *Shift* + `` ` `` to start a new terminal. If you are using the PowerShell 7 console, start a new copy of the console.

6. Examining the experimental features

```
Get-ExperimentalFeature
```

7. Examining output from the "command not found" suggestion feature

```
Foo
```

How it works...

In *step 1*, you use the `Get-ExperimentalFeature` cmdlet to discover the available experimental features and their current state, which (by default) looks like this:

```
PS C:\Foo> # 1. Discovering experimental features
PS C:\Foo> Get-ExperimentalFeature -Name * |
             Format-Table Name, Enabled, Description -Wrap

Name                                                 Enabled Description
----                                                 ------- -----------
PSCommandNotFoundSuggestion                            False Recommend potential commands based on fuzzy search on a
                                                             CommandNotFoundException
PSCultureInvariantReplaceOperator                      False Use culture invariant to-string convertor for lval in
                                                             replace operator
PSImplicitRemotingBatching                             False Batch implicit remoting proxy commands to improve
                                                             performance
PSNativePSPathResolution                               False Convert PSPath to filesystem path, if possible, for native
                                                             commands
PSNotApplyErrorActionToStderr                          False Don't have $ErrorActionPreference affect stderr output
PSSubsystemPluginModel                                 False A plugin model for registering and un-registering
                                                             PowerShell subsystems
Microsoft.PowerShell.Utility.PSManageBreakpointsInRunspace False Enables -BreakAll parameter on Debug-Runspace and Debug-Job
                                                             cmdlets to allow users to decide if they want PowerShell to
                                                             break immediately in the current location when they attach
                                                             a debugger. Enables -Runspace parameter on *-PSBreakpoint
                                                             cmdlets to support management of breakpoints in another
                                                             runspace.
PSDesiredStateConfiguration.InvokeDscResource          False Enables the Invoke-DscResource cmdlet and related features.
```

Figure 2.49: Discovering experimental features

To test out an experimental feature, in *step 2*, you run a non-existent command, with output such as this:

```
PS C:\Foo> # 2. Examining command not found result
PS C:\Foo> Foo
Foo: The term 'Foo' is not recognized as a name of a cmdlet, function, script file, or executable program.
Check the spelling of the name, or if a path was included, verify that the path is correct and try again
```

Figure 2.50: Examining the "command not found" result

In *step 3*, you enable the first feature, the `PSCommandNotFoundSuggestion` experimental feature for the current user, which looks like this:

```
PS C:\Foo> # 3. Enabling one experimental feature as current user
PS C:\Foo> Get-ExperimentalFeature -Name * |
             Select-Object -First 1 |
               Enable-ExperimentalFeature -Scope CurrentUser -Verbose

VERBOSE: Performing the operation "Enable-ExperimentalFeature" on target "PSCommandNotFoundSuggestion".
WARNING: Enabling and disabling experimental features do not take effect until next start of PowerShell.
```

Figure 2.51: Enabling an experimental feature for the current user

In *step 4*, you enable the second experimental feature, `PSCultureInvariantReplaceOperator`, which looks like this:

```
PS C:\Foo> # 4. Enabling one experimental feature for all users
PS C:\Foo> Get-ExperimentalFeature -Name * |
             Select-Object -Skip 1 -First 1 |
               Enable-ExperimentalFeature -Scope CurrentUser -Verbose
VERBOSE: Performing the operation "Enable-ExperimentalFeature" on target "PSCultureInvariantReplaceOperator".
WARNING: Enabling and disabling experimental features do not take effect until next start of PowerShell.
```

Figure 2.52: Enabling an experimental feature for all users

In *step 5*, you start a new version of PowerShell. This step produces no output as such.

In *step 6*, you examine the state of experimental features, noting that two new features are now available, which looks like this:

```
PS C:\Foo> # 6. Examining experimental features
PS C:\Foo> Get-ExperimentalFeature

Name                                Enabled Source                          Description
----                                ------- ------                          -----------
PSCommandNotFoundSuggestion            True PSEngine                        Recommend potential commands based on fuzzy sea…
PSCultureInvariantReplaceOperator      True PSEngine                        Use culture invariant to-string convertor for l…
PSImplicitRemotingBatching            False PSEngine                        Batch implicit remoting proxy commands to impro…
PSNativePSPathResolution              False PSEngine                        Convert PSPath to filesystem path, if possible,…
PSNotApplyErrorActionToStderr         False PSEngine                        Don't have $ErrorActionPreference affect stderr…
PSSubsystemPluginModel                False PSEngine                        A plugin model for registering and un-registeri…
Microsoft.PowerShell.Utility.PSMan…   False C:\program files\powershell\7\Modu… Enables -BreakAll parameter on Debug-Runspace a…
PSDesiredStateConfiguration.Invoke…   False C:\program files\powershell\7\Modu… Enables the Invoke-DscResource cmdlet and relat…
```

Figure 2.53: Examining experimental features

In *step 7*, you re-run the unknown command to observe the "command not found" suggestions, which look like this:

```
PS C:\Foo> # 7. Examining output from command not found suggestion feature
PS C:\Foo> foo
foo: The term 'foo' is not recognized as a name of a cmdlet, function, script file, or executable program.
Check the spelling of the name, or if a path was included, verify that the path is correct and try again.

Suggestion [4,General]: The most similar commands are: fl, ft, fw, gmo, nmo, oh, rmo, fc, fhx, F:.
PS C:\Foo>
```

Figure 2.54: Examining the output from the "command not found" suggestion feature

There's more...

In this recipe, you turn on two experimental features and examine one ("command not found" suggestions). In most cases, you should be safe to enable all of the experimental features, but it is always safer to turn them on one by one and test your scripts carefully.

3

Exploring Compatibility with Windows PowerShell

In this chapter, we cover the following recipes:

- Exploring compatibility with Windows PowerShell
- Using the Windows PowerShell compatibility solution
- Exploring compatibility solution limitations
- Exploring the module deny list
- Importing format XML
- Leveraging compatibility

Introduction

Microsoft built Windows PowerShell to work with Microsoft's .NET Framework. You can think of PowerShell as a layer on top of .NET. When you use `Get-ChildItem` to return file or folder details, the cmdlet invokes a .NET class to do much of the heavy lifting involved. In *Chapter 5*, *Exploring .NET*, you learn more about .NET.

As Windows PowerShell evolved, each new version took advantage of improvements in newer versions of .NET to provide additional features.

In 2014, Microsoft announced that they would release the .NET Framework as open source, to be known as .NET Core. Microsoft also decided to freeze development of Windows PowerShell, in favor of open sourcing Windows PowerShell. The first two initial versions were known as PowerShell Core. With the release of PowerShell 7.0, the team dropped the name "Core" and announced that future versions are to be known as just PowerShell. This book refers to the latest version as simply PowerShell 7.

As an IT professional, the critical question you should be asking is whether PowerShell 7 can run your specific workload. The answer, as ever, is "it depends." It depends on the specific Windows PowerShell features and modules/commands on which you rely. Since there are many product teams involved, a fully coordinated solution is not straightforward. As this book shows, there is very little you were able to do in Windows PowerShell that you cannot do in PowerShell 7.

At the time of publication, the latest released version of PowerShell 7.1.3, with a preview-8 version of PowerShell 7.2 also available. This is highly likey to change by the time you read this book.

Module compatibility

In terms of compatibility with Windows PowerShell, there are three sets of modules containing different commands to look at:

- Modules/commands shipped with Windows PowerShell and on which your scripts depend.
- Modules/commands included as part of a Windows feature – these contain commands to manage aspects of Windows services, usually as part of the **Remote Server Administration Tools** (**RSAT**), which you can load independently of the services themselves.
- Other third-party modules, such as the `NTFSSecurity` module. This book makes use of several external modules whose authors may, or may not, have updated their modules to run on .NET Core.

Windows PowerShell included modules that provide basic PowerShell functionality. For example, the `Microsoft.PowerShell.Management` module contains the `Get-Process` cmdlet. With PowerShell 7, the development team re-implemented the fundamental Windows PowerShell commands to provide good fidelity with Windows PowerShell. A few Windows PowerShell commands made use of proprietary APIs, so they could not be included, but the number is relatively small.

These core modules (`Microsoft.PowerShell.*`) now reside in the PowerShell installation folder, which simplifies side-by-side use of multiple versions of PowerShell. Through the use of the environment variable `PSModulePath`, when you call a cmdlet in PowerShell 7, you get the "right" cmdlet and supported underpinnings. It is important to note that these core modules were developed by the Windows PowerShell team initially and are not part of the open source PowerShell.

The Windows client and Windows Server also provide modules that enable you to manage your systems. Modules like the Active Directory module give you the tools to manage AD in your environment. These Windows modules live in the `C:\Windows\System32\WindowsPowerShell\v1.0\modules` folder.

The ownership of these operating system-related modules lies with the individual product teams, some of whom have not been motivated to update their modules to run with .NET Core. Like so many things, this is a work in progress. Some modules, such as the `ActiveDirectory` module, have been updated and are available natively in PowerShell 7. For a quick look at compatibility, view this document: `https://docs.microsoft.com/en-us/powershell/scripting/whats-new/module-compatibility?view=powershell-7.1`. It is worth noting that updating these OS modules can only be done by upgrading to a newer version of the OS, which is unfortunate.

The final set of commands to think about are third-party modules, possibly obtained from the PowerShell Gallery, Spiceworks, or a host of other online sources. As the recipes in this book demonstrate, some third-party modules work fine natively in the latest versions of PowerShell 7, while others do not and may never work.

In the *Exploring compatibility with Windows PowerShell* recipe, you look specifically at the new commands you have available in PowerShell 7, as well as where PowerShell and Windows PowerShell find core commands.

Since many of the Windows product teams and other module providers might not update their modules to work natively, the PowerShell team devised a compatibility solution that enables you to import and use Windows PowerShell modules more or less seamlessly in PowerShell 7. In the *Using the Windows PowerShell compatibility solution* recipe, you investigate the workings of the Windows PowerShell compatibility solution.

Incompatible modules

With the compatibility solution, the modules developed for Windows PowerShell work well, as demonstrated by many recipes in this book. However, there are some small and some more fundamental issues in a few cases. Some things do not and may never work inside PowerShell 7 and beyond. For instance, despite the compatibility solution, the `BestPractices`, `PSScheduledJob`, and `WindowsUpdate` modules do not work in PowerShell 7.

With Windows PowerShell you use `WindowsUpdatemodule` to manage **Windows Server Update Services** (**WSUS**). The WSUS's, `UpdateServices` module, makes use of object methods. With the compatibility mechanism, these methods are not available in a PowerShell 7 session. Additionally, the module makes use of the **Simple Object Access Protocol** (**SOAP**) protocol for which .NET Core provides no support (and is not likely to anytime soon). The other two modules make use of .NET types that are not provided with .NET Core and are not likely to be updated.

Other Windows PowerShell features that are not going to work in PowerShell 7 include:

- Windows PowerShell workflows: Workflows require the Windows Workflow Foundation component of .NET, which the .NET team has not ported to .NET Core.
- Windows PowerShell snap-ins: Snap-ins provided a way to add new cmdlets in PowerShell V2. Modules replaced snap-ins as an add-in mechanism in PowerShell V2, and are the sole method you use to add commands to PowerShell. You can convert some older snap-ins into modules by creating a module manifest, but most require the authors to re-engineer their modules.
- The WMI cmdlets: The `CIMCmdlets` module replaces the older WMI Cmdlets.
- **Desired State Configuration** (**DSC**): The core functions of DSC are not available to you from within PowerShell 7.
- The `WebAdministration` module's IIS provider: This means many IIS configuration scripts do not work in PowerShell 7, even with the compatibility solution.
- The `Add-Computer`, `Checkpoint-Computer`, `Remove-Computer`, and `Restore-Computer` commands from the `Microsoft.PowerShell.Management` module.

While there are a few Windows PowerShell features and commands you cannot use in PowerShell 7, even with the help of the Windows PowerShell compatibility solution, you can always use Windows PowerShell.

As noted above, there are just three Microsoft-authored modules that do not work in PowerShell 7, natively or via the compatibility mechanism. If you try to use these, you receive unhelpful error messages. To avoid an awful user experience (and no doubt tons of support calls), the PowerShell team devised a list of modules that PowerShell does not load. If you try to use these modules, you get an error message, as you see in the *Exploring the module deny list* recipe.

One downside to the compatibility mechanism is that any format XML included with the module does not get loaded into the PowerShell 7 session. The result is that PowerShell 7 does not format objects as nicely. Fortunately, there is a way around this, as you see in the *Importing format XML* recipe.

The Windows PowerShell compatibility solution makes use of a specially created PowerShell remoting session. In *Leveraging compatibility*, you look at that session and learn how you can take advantage of it.

Despite compatibility issues, it is more than worthwhile to move forward to PowerShell 7 and away from Windows PowerShell. One key reason is improved performance. You can use the `ForEach-Object -Parallel` construct to run script blocks in parallel without having to resort to workflows. If your script has to perform actions on a large number of computers, or you are processing a large array, the performance improvements in PowerShell 7 justify moving forward. On top of this, there are all the newly added features, which make life so much easier.

Exploring compatibility with Windows PowerShell

When you invoke a cmdlet, PowerShell has to load the module containing the cmdlet and can then run that cmdlet. By default, PowerShell uses the paths on the environment variable `$env:PSModulePath` to discover the modules and the cmdlets contained in those modules.

As this recipe shows, the set of paths held by `$env:PSModulePath` has changed between Windows PowerShell 5.1 and PowerShell 7.

In this recipe, you examine the paths that PowerShell uses by default to load modules. You also look at the new commands now provided in PowerShell 7.

Getting ready

You use `SRV1` for this recipe after you have loaded PowerShell 7.1 and VS Code.

How to do it...

1. Ensuring PowerShell remoting is fully enabled

```
Enable-PSRemoting -Force -WarningAction SilentlyContinue |
  Out-Null
```

2. Getting session using endpoint for Windows PowerShell 5.1

```
$SHT1 = @{
  ComputerName      = 'localhost'
  ConfigurationName = 'microsoft.powershell'
}
$SWP51 = New-PSSession @SHT1
```

3. Getting session using PowerShell 7.1 endpoint

```
$CNFName = Get-PSSessionConfiguration |
             Where-Object PSVersion -eq '7.1' |
               Select-Object -Last 1
$SHT2 = @{
  ComputerName      = 'localhost'
  ConfigurationName = $CNFName.Name
}
$SP71    = New-PSSession @SHT2
```

4. Defining a script block to view default module paths

```
$SBMP = {
  $PSVersionTable
  $env:PSModulePath -split ';'
}
```

5. Reviewing paths in Windows PowerShell 5.1

```
Invoke-Command -ScriptBlock $SBMP -Session $SWP51
```

6. Reviewing paths in PowerShell 7.1

```
Invoke-Command -ScriptBlock $SBMP -Session $SP71
```

7. Creating a script block to get commands in PowerShell

```
$SBC = {
  $ModPaths = $Env:PSModulePath -split ';'
  $CMDS = @()
  Foreach ($ModPath in $ModPaths) {
    if (!(Test-Path $Modpath)) {Continue}
    # Process modules found in an existing module path
    $Mods = Get-ChildItem -Path $ModPath -Directory
    foreach ($Mod in $Mods){
      $Name  = $Mod.Name
      $Cmds  += Get-Command -Module $Name
    }
  }
  $Cmds  # return all commands discovered
}
```

8. Discovering all 7.1 cmdlets

```
$CMDS71 = Invoke-Command -ScriptBlock $SBC -Session $SP71 |
            Where-Object CommandType -eq 'Cmdlet'
"Total commands available in PowerShell  7.1 [{0}]" -f $Cmds71.count
```

9. Discovering all 5.1 cmdlets

```
$CMDS51 = Invoke-Command -ScriptBlock $SBC -Session $SWP51 |
            Where-Object CommandType -eq 'Cmdlet'
"Total commands available in PowerShell  5.1 [{0}]" -f $Cmds51.count
```

10. Creating arrays of just cmdlet names

```
$Commands51 = $CMDS51 |
  Select-Object -ExpandProperty Name |
    Sort-Object -Unique
$Commands71 = $CMDS71 |
  Select-Object -ExpandProperty Name |
    Sort-Object -Unique
```

11. Discovering new commands in PowerShell 7.1

```
Compare-Object $Commands51 $Commands71  |
  Where-Object sideindicator -match '^=>'
```

12. Creating a script block to check core PowerShell modules

```
$CMSB = {
  $M = Get-Module -Name 'Microsoft.PowerShell*' -ListAvailable
  $M
  "$($M.count) modules found in $($PSVersionTable.PSVersion)"
}
```

13. Viewing core modules in Windows PowerShell 5.1

```
Invoke-Command -Session $SWP51 -ScriptBlock $CMSB
```

14. Viewing core modules in PowerShell 7.1

```
Invoke-Command -Session $SP71 -ScriptBlock $CMSB
```

How it works...

In *step 1*, you ensure that PowerShell remoting is enabled fully within PowerShell 7. This should not be necessary, but this step, which produces no output, ensures that the endpoints you use later in this recipe exist.

In *step 2*, you create a new PowerShell remoting session to a Windows PowerShell 5.1 endpoint on the local computer. The endpoint's configuration name, `microsoft.powershell`, is a well-known configuration name that runs Windows PowerShell 5.1.

With *step 3*, you then repeat the operation and create a new PowerShell remoting session to a PowerShell 7.1 endpoint. This step produces no output.

In *step 4*, you create a simple script block that displays the PowerShell version table and shows the endpoint's module paths.

In *step 5*, you view the version of PowerShell running (inside the remoting session) and that version's module paths. The output of this step looks like this:

```
PS C:\Foo> # 5. Reviewing paths in Windows PowerShell 5.1
PS C:\Foo> Invoke-Command -ScriptBlock $SBMP -Session $SWP51

Name                           Value
----                           -----
SerializationVersion           1.1.0.1
PSEdition                      Desktop
BuildVersion                   10.0.20270.1000
CLRVersion                     4.0.30319.42000
WSManStackVersion              3.0
PSCompatibleVersions           {1.0, 2.0, 3.0, 4.0, 5.0, 5.1.20270.1000}
PSRemotingProtocolVersion      2.3
PSVersion                      5.1.20270.1000

C:\Users\Administrator.RESKIT\Documents\WindowsPowerShell\Modules
C:\Program Files\WindowsPowerShell\Modules
C:\Windows\system32\WindowsPowerShell\v1.0\Modules
```

Figure 3.1: Inspecting the PowerShell 5.1 version and module paths

In *step 6*, you view the paths inside a PowerShell 7.1 endpoint, which looks like this:

```
PS C:\Foo> # 6. Reviewing paths in PowerShell 7.1
PS C:\Foo> Invoke-Command -ScriptBlock $SBMP -Session $SP71

Name                           Value
----                           -----
SerializationVersion           1.1.0.1
PSEdition                      Core
PSVersion                      7.1.1
GitCommitId                    7.1.1
WSManStackVersion              3.0
OS                             Microsoft Windows 10.0.20270
PSCompatibleVersions           {1.0, 2.0, 3.0, 4.0, 5.0, 5.1.10032.0, 6.0.0, 6.1.0, 6.2.0, 7.0.0, 7.1.1}
PSRemotingProtocolVersion      2.3
Platform                       Win32NT

C:\Users\Administrator.RESKIT\Documents\PowerShell\Modules
C:\Program Files\PowerShell\Modules
c:\program files\powershell\7\Modules
C:\Program Files\WindowsPowerShell\Modules
C:\Windows\system32\WindowsPowerShell\v1.0\Modules
```

Figure 3.2: Inspecting the PowerShell 7.1 version and module paths

In *step 7*, you create a script block gets the details of each module available in a given remoting endpoint. This step produces no output. In *step 8*, you run the script block in a PowerShell 7.1 remoting endpoint to discover the commands available, which looks like this:

```
PS C:\Foo> # 8. Discovering all 7.1 cmdlets
PS C:\Foo> $CMDS71 = Invoke-Command -ScriptBlock $SBC -Session $SP71 |
               Where-Object CommandType -eq 'Cmdlet'
PS C:\Foo> "Total commands available in PowerShell 7.1 [{0}]" -f $Cmds71.count
Total commands available in PowerShell  7.1 [668]
```

Figure 3.3: Discovering all cmdlets in PowerShell 7.1

In *step 9*, you run this script block inside the Windows PowerShell endpoint to produce output like the following:

```
PS C:\Foo> # 9. Discovering all 5.1 cmdlets
PS C:\Foo> $CMDS51 = Invoke-Command -ScriptBlock $SBC -Session $SWP51 |
                         Where-Object CommandType -eq 'Cmdlet'
PS C:\Foo> "Total commands available in PowerShell 5.1 [{0}]" -f $Cmds51.count
Total commands available in PowerShell 5.1 [594]
```

Figure 3.4: Discovering all cmdlets in PowerShell 5.1

In *step 10*, you create arrays to contain the names of the commands available. You compare these in *step 11*, to discover the commands in PowerShell 7.1 that are not in Windows PowerShell 5.1, which looks like this:

```
PS C:\Foo> # 11. Discovering new cmdlets in PowerShell 7.1
PS C:\Foo> Compare-Object $Commands51 $Commands71  |
             Where-Object SideIndicator -match '^=>'

InputObject               SideIndicator
-----------               -------------
ConvertFrom-Markdown      =>
ConvertFrom-SddlString    =>
Format-Hex                =>
Get-Error                 =>
Get-FileHash              =>
Get-MarkdownOption        =>
Get-Uptime                =>
Get-Verb                  =>
Import-PowerShellDataFile =>
Join-String               =>
New-Guid                  =>
New-TemporaryFile         =>
Remove-Alias              =>
Remove-Service            =>
Set-MarkdownOption        =>
Show-Markdown             =>
Start-ThreadJob           =>
Test-Json                 =>
```

Figure 3.5: Discovering cmdlets that are in PowerShell 7.1 but not 5.1

In *step 12*, you create a script block that you can use to discover the core PowerShell modules and their file storage locations. This step creates no output. In *step 13*, you run this script block inside the Windows PowerShell remoting session you created earlier, which looks like this:

```
PS C:\Foo> # 13. Viewing core modules in Windows PowerShell 5.1
PS C:\Foo> Invoke-Command -Session $SWP51 -ScriptBlock $CMSB

    Directory: C:\Program Files\WindowsPowerShell\Modules

ModuleType Version    PreRelease Name                                PSEdition ExportedCommands                                PSComputerName
---------- -------    ---------- ----                                --------- ----------------                                --------------
Script     1.0.1                 Microsoft.PowerShell.Operation.Val… Desk      {Get-OperationValidation, Invoke-OperationVa…  localhost

    Directory: C:\Windows\system32\WindowsPowerShell\v1.0\Modules

ModuleType Version    PreRelease Name                                PSEdition ExportedCommands                                PSComputerName
---------- -------    ---------- ----                                --------- ----------------                                --------------
Manifest   1.0.1.0               Microsoft.PowerShell.Archive        Desk      {Compress-Archive, Expand-Archive}              localhost
Manifest   3.0.0.0               Microsoft.PowerShell.Diagnostics    Core,Desk {Get-WinEvent, Get-Counter, Import-Counter, …   localhost
Manifest   3.0.0.0               Microsoft.PowerShell.Host           Desk      {Stop-Transcript, Start-Transcript}             localhost
Manifest   1.0.0.0               Microsoft.PowerShell.LocalAccounts  Core,Desk {Get-LocalGroupMember, algm, Get-LocalUser, …   localhost
Manifest   3.1.0.0               Microsoft.PowerShell.Management     Desk      {Remove-EventLog, Set-Service, Get-ComputerI…   localhost
Script     1.0                   Microsoft.PowerShell.ODataUtils     Desk      Export-ODataEndpointProxy                       localhost
Manifest   3.0.0.0               Microsoft.PowerShell.Security       Desk      {Get-Credential, Get-ExecutionPolicy, Unprot…   localhost
Manifest   3.1.0.0               Microsoft.PowerShell.Utility        Desk      {Get-PSCallStack, Update-TypeData, Add-Type,…   localhost
9 modules found in 5.1.20270.1000
```

Figure 3.6: Viewing core modules in PowerShell 5.1

You then rerun this script block in a PowerShell 7.1 endpoint, in *step 14*. The output from this step looks like this:

```
PS C:\Foo> # 14. Viewing core modules in PowerShell 7.1
PS C:\Foo> Invoke-Command -Session $SP71 -ScriptBlock $CMSB

    Directory: C:\program files\powershell\7\Modules

ModuleType Version    PreRelease Name                                PSEdition ExportedCommands                                PSComputerName
---------- -------    ---------- ----                                --------- ----------------                                --------------
Manifest   1.2.5                 Microsoft.PowerShell.Archive        Desk      {Compress-Archive, Expand-Archive}              localhost
Manifest   7.0.0.0               Microsoft.PowerShell.Diagnostics    Core      {Get-WinEvent, Get-Counter, New-WinEvent}       localhost
Manifest   7.0.0.0               Microsoft.PowerShell.Host           Core      {Start-Transcript, Stop-Transcript}             localhost
Manifest   7.0.0.0               Microsoft.PowerShell.Management     Core      {Convert-Path, New-PSDrive, New-ItemProperty, Re… localhost
Manifest   7.0.0.0               Microsoft.PowerShell.Security       Core      {Test-FileCatalog, Unprotect-CmsMessage, Set-Exe… localhost
Manifest   7.0.0.0               Microsoft.PowerShell.Utility        Core      {Format-Custom, Write-Verbose, Disable-PSBreakpo… localhost

    Directory: C:\Program Files\WindowsPowerShell\Modules

ModuleType Version    PreRelease Name                                PSEdition ExportedCommands                                PSComputerName
---------- -------    ---------- ----                                --------- ----------------                                --------------
Script     1.0.1                 Microsoft.PowerShell.Operation.Val… Desk      {Invoke-OperationValidation, Get-OperationValida… localhost

    Directory: C:\Windows\system32\WindowsPowerShell\v1.0\Modules

ModuleType Version    PreRelease Name                                PSEdition ExportedCommands                                PSComputerName
---------- -------    ---------- ----                                --------- ----------------                                --------------
Manifest   3.0.0.0               Microsoft.PowerShell.Diagnostics    Core,Desk {Get-WinEvent, Get-Counter, New-WinEvent, Import… localhost
Manifest   1.0.0.0               Microsoft.PowerShell.LocalAccounts  Core,Desk {Add-LocalGroupMember, nlg, Enable-LocalUser, rl… localhost
9 modules found in 7.1.1
```

Figure 3.7: Viewing core modules in PowerShell 7.1

There's more...

The code in *step 2* creates a PowerShell remoting endpoint to a well-known endpoint for Windows PowerShell 5. You cannot view Windows PowerShell remoting endpoint configurations from within PowerShell 7.1.

In *step 3*, you get a session that provides PowerShell 7.1 using a slightly different technique than you used in *step 2*. PowerShell 7.1 has multiple endpoints, especially if you have both a release version and a preview version loaded side by side. You can use `Get-PSSessionConfiguration` to view the available remoting endpoint configurations in your specific PowerShell 7 host. Thus, you do not see the Windows PowerShell remoting endpoints inside PowerShell 7. Although the `Get-PSSessionConfiguration` cmdlet does not return any Windows PowerShell configuration details, the remoting endpoints do exist, and you can use them, as you see in *step 2*.

In *steps 5* and *6*, you discover the default module paths in Windows PowerShell and PowerShell 7.1. As you can see, in PowerShell 7, there are five default module paths, versus just three in Windows PowerShell 5.1. The first three module folders with PowerShell 7 enable PowerShell 7 to pick up commands from PowerShell 7.1-specific folders (if they exist), and the last two modules enable you to access older Windows PowerShell commands that you do not have with PowerShell 7.

In *steps 13* and *14*, you discover the core modules inside Windows PowerShell. These core modules contain the basic internal PowerShell commands, such as `Get-ChildItem`, found in the `Microsoft.PowerShell.Management` module.

This recipe specifically makes use of PowerShell 7.1 By the time you read this, a later version may be available and this recipe would need to be updated accordingly to utilize later released version of PowerShell.

Using the Windows PowerShell compatibility solution

The PowerShell 7 Windows compatibility solution allows you to use older Windows PowerShell commands whose developers have not (yet) ported the commands to work natively in PowerShell 7. PowerShell 7 creates a special remoting session into a Windows PowerShell 5.1 endpoint, loads the modules into the remote session, then uses implicit remoting to expose proxy functions inside the PowerShell 7 session. This remoting session has a unique session name, `WinPSCompatSession`. Should you use multiple Windows PowerShell modules, PowerShell 7 loads them all into a single remoting session. Also, this session uses the "process" transport mechanism versus **Windows Remote Management** (**WinRM**). WinRM is the core transport protocol used with PowerShell remoting. The process transport is the transport used to run background jobs; it has less overhead than using WinRM, so is more efficient.

An example of the compatibility mechanism is using `Get-WindowsFeature`, a cmdlet inside the `ServerManager` module. You use the command to get details of features that are installed, or not, inside Windows Server. You use other commands in the `ServerManager` module to install and remove features.

Unfortunately, the Windows Server team has not yet updated this module to work within PowerShell Core. Via the compatibility solution, the commands in the `ServerManager` module enable you to add, remove, and view features. The Windows PowerShell compatibility mechanism allows you to use existing Windows PowerShell scripts in PowerShell 7, although with some very minor caveats.

When you invoke commands in PowerShell 7, PowerShell uses its command discovery mechanism to determine which module contains your desired command. In this case, that module is the `ServerManager` Windows PowerShell module. PowerShell 7 then creates the remoting session and, using implicit remoting, imports the commands in the module as proxy functions. You then invoke the proxy functions to accomplish your goal. For the most part, this is totally transparent. You use the module's commands, and they return the object(s) you request. One minor issue is that the compatibility mechanism does not import format XML for the Windows PowerShell module. The result is that the default output of some objects is not the same. There is a workaround for this, described in the *Importing format XML* recipe.

With implicit remoting, PowerShell creates a function inside a PowerShell 7 session with the same name and parameters as the actual command (in the remote session). You can view the function definition in the `Function` drive (`Get-Item Function:Get-WindowsFeature | Format-List -Property *`). The output shows the proxy function definition that PowerShell 7 creates when it imports the remote module.

When you invoke the command by name, for example, `Get-WindowsFeature`, PowerShell runs the function. The function then invokes the remote cmdlet using the steppable pipeline. Implicit remoting is a complex feature that is virtually transparent in operation. You can read more about implicit remoting at `https://www.techtutsonline.com/implicit-remoting-windows-powershell/`.

Getting ready

You run this recipe on `SRV1`, after installing PowerShell 7 and VS Code.

How to do it...

1. Discovering the `ServerManager` module

   ```
   Get-Module -Name ServerManager -ListAvailable
   ```

2. Discovering a command in the `ServerManager` module

   ```
   Get-Command -Name Get-WindowsFeature
   ```

3. Importing the module explicitly

```
Import-Module -Name ServerManager
```

4. Discovering the commands inside the module

```
Get-Module -Name ServerManager | Format-List
```

5. Using a command in the `ServerManager` module

```
Get-WindowsFeature -Name TFTP-Client
```

6. Running the command in a remoting session

```
$Session = Get-PSSession -Name WinPSCompatSession
Invoke-Command -Session $Session -ScriptBlock {
  Get-WindowsFeature -Name DHCP |
    Format-Table
}
```

7. Removing the `ServerManager` module from the current session

```
Get-Module -Name ServerManager |
  Remove-Module
```

8. Installing a Windows feature using module autoload

```
Install-WindowsFeature -Name TFTP-Client
```

9. Discovering the feature

```
Get-WindowsFeature -Name TFTP-Client
```

10. Viewing output inside Windows PowerShell session

```
Invoke-Command -Session $Session -ScriptBlock {
    Get-WindowsFeature -Name 'TFTP-Client' |
      Format-Table
}
```

How it works...

In *step 1*, you use the `Get-Module` command to discover the commands in the `ServerManager` module. Despite using the `-ListAvailable` switch, since this module does not work natively in PowerShell 7, this command returns no output.

In *step 2*, you use `Get-Command` to discover the `Get-WindowsFeature` command. The output looks like this:

```
PS C:\Foo> # 2. Discovering a command in the Server Manager module
PS C:\Foo> Get-Command -Name Get-ChildItem

CommandType  Name                Version   Source
-----------  ----                -------   ------
Function     Get-WindowsFeature  1.0       ServerManager
```

Figure 3.8: Discovering the Get-WindowsFeature command

In *step 3*, you import the `ServerManager` module explicitly, which looks like this:

```
PS C:\Foo> # 3. Importing the module explicitly
PS C:\Foo> Import-Module -Name ServerManager
WARNING: Module servermanager is loaded in Windows PowerShell using
WinPSCompatSession remoting session; please note that all input and
output of commands from this module will be deserialized objects. If
you want to load this module into PowerShell please use
'Import-Module -SkipEditionCheck' syntax.
```

Figure 3.9: Importing the ServerManager module explicitly

Now that PowerShell 7 has imported the module, you can see the output from `Get-Module`. The output from *step 4* looks like this:

```
PS C:\Foo> # 4. Discovering the module again
PS C:\Foo> Get-Module -Name ServerManager | Format-List

Name              : servermanager
Path              : C:\Users\Administrator\AppData\Local\Temp\1\remoteIpMoProxy_servermanager_2.0.0.0_localho
                    st_ab3b80b5-19af-49e2-b91d-55cf6f245c2f\remoteIpMoProxy_servermanager_2.0.0.0_localhost_a
                    b3b80b5-19af-49e2-b91d-55cf6f245c2f.psm1
Description       : Implicit remoting for
ModuleType        : Script
Version           : 1.0
PreRelease        :
NestedModules     : {}
ExportedFunctions : {Disable-ServerManagerStandardUserRemoting, Enable-ServerManagerStandardUserRemoting,
                    Get-WindowsFeature, Install-WindowsFeature, Uninstall-WindowsFeature}
ExportedCmdlets   :
ExportedVariables :
ExportedAliases   : {Add-WindowsFeature, Remove-WindowsFeature}
```

Figure 3.10: Discovering the ServerManager module

To illustrate that an older Windows PowerShell command works with PowerShell 7, in *step 5*, you invoke `Get-WindowsFeature`. This command discovers whether the TFTP Client feature is installed, which looks like this:

```
PS C:\Foo> # 5. Using a command in the ServerManager module
PS C:\Foo> Get-WindowsFeature -Name DHCP

Display Name      Name           Install State
------------      ----           -------------
                  TFTP-Client    Available
```

Figure 3.11: Invoking the Get-WindowsFeature command

In *step 6*, you rerun the `Get-WindowsFeature` command inside the Windows PowerShell remoting session, with output like this:

```
PS C:\Foo> # 6. Running the command in a remoting session
PS C:\Foo> $Session = Get-PSSession -Name WInPSCompatSession
PS C:\Foo> Invoke-Command -Session $Session -ScriptBlock {
             Get-WindowsFeature -Name DHCP |
               Format-Table
           }

Display Name         Name         Install State
------------         ----         -------------
[ ] TFTP Client      TFTP-Client  Available
```

Figure 3.12: Invoking Get-WindowsFeature in a remoting session

In *step 7*, you remove the `ServerManager` module from the current PowerShell session, producing no output. In *step 8*, you install the `TFTP-Client` feature, which looks like this:

```
PS C:\Foo> # 8. Installing a Windows feature using module autoload
PS C:\Foo> Install-WindowsFeature -Name TFTP-Client

Success Restart Needed Exit Code  Feature Result
------- -------------- ---------  --------------
True    No             Success    {TFTP Client}}
```

Figure 3.13: Installing TFTP-Client

In *step 9*, you rerun the `Get-WindowsFeature` command to check the state of the `TFTP-Client` feature, which looks like this:

```
PS C:\Foo> # 9. Discovering the feature
PS C:\Foo> Get-WindowsFeature -Name TFTP-Client

Display Name      Name          Install State
------------      ----          -------------
                  TFTP-Client       Installed
```

Figure 3.14: Checking the state of TFTP-Client

In *step 10*, you re-view the `TFTP-Client` feature inside the Windows PowerShell remoting session, which looks like this:

```
PS C:\Foo> # 10. Viewing output inside Windows PowerShell session
PS C:\Foo> Invoke-Command -Session $Session -ScriptBlock {
             Get-WindowsFeature -Name 'TFTP-Client' |
               Format-Table
           }

Display Name      Name          Install State
------------      ----          -------------
[X] TFTP Client   TFTP-Client       Installed
```

Figure 3.15: Viewing TFTP-Client inside the remoting session

There's more...

In *step 1*, you attempted to find the `ServerManager` module using the `-ListAvailable` switch. This module is not available natively in PowerShell 7, hence the lack of output. Even though the compatibility mechanism can find the module, by default, `Get-Module` does not display the module.

In *step 2*, you discover that the `Get-WindowsFeature` command comes from the `ServerManager` module, and you then load it explicitly in *step 3*. Loading the module using `Import-Module` generates the warning message you can see in the output. Note that in PowerShell 7, the `Get-WindowsFeature` command is a function, rather than a cmdlet. The implicit remoting process creates a proxy function (in the PowerShell 7 session), which then invokes the underlying command in the remote session. You can examine the function's definition to see how the proxy function invokes the remote function.

In *steps 9* and *10*, you view the output from `Get-WindowsFeature` inside PowerShell 7 and then inside the remoting session to Windows PowerShell. In *step 9*, you see a different output from *step 10*, which is the result of PowerShell 7 not importing the format XML for this module. You can see how to get around this minor issue in *Importing format XML* later in this chapter.

The Windows PowerShell compatibility mechanism in PowerShell 7 does an excellent job of supporting otherwise incompatible Windows PowerShell modules. But even with this, there are some commands and modules that are just not going to work.

Exploring compatibility solution limitations

In the previous recipe, you saw how you could use the Windows PowerShell compatibility mechanism built into PowerShell 7. This solution provides you with access to modules that their owners have not yet converted to run natively in PowerShell 7. This solution provides improved compatibility but does have some minor limitations.

The first limitation is discovery. You can't easily discover unsupported commands. You cannot, for example, use PowerShell 7's `Get-Module` to list the `ServerManager` module, even though you can use `Get-Command` to discover commands inside the module.

Another limitation of the compatibility solution is that PowerShell 7 does not import any display or type XML contained in the module. The result is that some commands may display output slightly differently. There are ways around this, including just running the entire command inside a remoting session.

Despite the compatibility solution, some Windows Server modules simply do not work at all in PowerShell 7. If you load one of these modules manually, the module may load, but some or all of the commands in the module do not work and often return error messages that are not actionable.

Getting ready

Run this in a new PowerShell session on `SRV1` after you have installed PowerShell 7 and VS Code.

How to do it...

1. Attempting to view a Windows PowerShell module

```
Get-Module -Name ServerManager -ListAvailable
```

2. Trying to load a module without edition check

```
Import-Module -Name ServerManager -SkipEditionCheck
```

3. Discovering the `Get-WindowsFeature` Windows PowerShell command

```
Get-Command -Name Get-WindowsFeature
```

4. Examining the Windows PowerShell compatibility remote session

```
$Session = Get-PSSession
$Session | Format-Table -AutoSize
```

5. Examining `Get-WindowsFeature` in the remote session

```
$SBRC = {Get-Command -Name Get-WindowsFeature}
Invoke-Command -Session $Session -ScriptBlock $SBRC
```

6. Invoking `Get-WindowsFeature` locally

```
Invoke-Command $SBRC
```

How it works...

In *step 1*, you use the `Get-Module` cmdlet to get details of the `ServerManager` module. However, since this module is not supported natively, `Get-Module` returns no output when you invoke it within a PowerShell 7 session, even though you use the `-ListAvailable` switch.

In *step 2*, you use the `-SkipEditionCheck` switch to instruct `Import-Module` to try to load the `ServerManager` module into the PowerShell 7 session, which looks like this:

```
PS C:\Foo> # 2. Trying to load a module without edition check
PS C:\Foo> Import-Module -Name ServerManager -SkipEditionCheck
Import-Module: Could not load type 'System.Diagnostics.Eventing.EventDescriptor' from
assembly 'System.Core, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089'.
```

Figure 3.16: Loading ServerManager without edition check

As you can see in the output, `Import-Module` errors out when it attempts to load the module in PowerShell 7, with an error message stating that `Import-Module` was not able to load the .NET type `System.Diagnostics.Eventing.EventDescriptor`. Since the `EventDescriptor` type is not available in .NET Core, you cannot use the `ServerManager` module natively. The only alternatives are for the Server Manager team to update their code, or for you to run the cmdlet in Windows PowerShell (either explicitly or via the compatibility mechanism).

In *step 3*, you use `Get-Command` to discover the `Get-WindowsFeature` command, which succeeds, as you can see in the output from this step:

```
PS C:\Foo> # 3. Discovering a Windows PowerShell command
PS C:\Foo> Get-Command -Name Get-WindowsFeature

CommandType     Name                Version    Source
-----------     ----                -------    ------
Function        Get-WindowsFeature  1.0        ServerManager
```

Figure 3.17: Discovering the Get-WindowsFeature command

In *step 4*, you use `Get-PSSession` to discover the Windows PowerShell compatibility remote session. The output of this step is as follows:

```
PS C:\Foo> # 4. Examining remote session
PS C:\Foo> $Session = Get-PSSession
PS C:\Foo> $Session | Format-Table -AutoSize

Id Name               Transport ComputerName ComputerType  State  ConfigurationName Availability
-- ----               --------- ------------ ------------  -----  ----------------- ------------
20 WinPSCompatSession Process   localhost    RemoteMachine Opened                       Available
```

Figure 3.18: Discovering the compatibility remote session

In *step 5*, you invoke the `Get-WindowsFeature` command inside the remoting session to view the command details inside Windows PowerShell 5.1, like this:

```
PS C:\Foo> # 5. Invoking Get-WindowsFeature in the remote session
PS C:\Foo> $SBRC = {Get-Command -Name Get-WindowsFeature}
PS C:\Foo> Invoke-Command -Session $Session -ScriptBlock $SBRC

CommandType     Name                Version    Source         PSComputerName
-----------     ----                -------    ------         --------------
Cmdlet          Get-WindowsFeature  2.0.0.0    ServerManager  localhost
```

Figure 3.19: Invoking Get-WindowsFeature in the remote session

With *step 6*, you run `Invoke-Command` to discover the details about this command inside your PowerShell 7 console. The output of this step is as follows:

```
PS C:\Foo> # 6. Invoking Get-WindowsFeature locally
PS C:\Foo> Invoke-Command $SBRC

CommandType  Name                 Version  Source
-----------  ----                 -------  ------
Function     Get-WindowsFeature   1.0      ServerManager
```

Figure 3.20: Invoking Get-WindowsFeature locally

There's more...

As you can see from the steps in this recipe, the Windows PowerShell compatibility solution imposes some minor restrictions on features Windows PowerShell users have grown accustomed to having. These limitations are relatively minor, and there are some workarounds.

As you can see in *step 1*, some modules are not directly discoverable in PowerShell 7. Attempting to force load a module into a PowerShell 7 session, as you attempt in *step 2*, fails (with a rather unhelpful message for the user).

In this case, when PowerShell 7 begins to load the module, PowerShell attempts to utilize a type (a .NET object) that does exist in the full .NET CLR, but not in .NET Core. This .NET type is not only not implemented in .NET Core, but the .NET team have effectively deprecated it. To read through the discussions on this, see `https://github.com/dotnet/core/issues/2933`. The discussion in this post is an outstanding example of the transparency of open source software where you can see the problem and trace its resolution.

The `ServerManager` module is one of many Windows team modules that require updating before it can be used natively in PowerShell 7. Hopefully, future releases of Windows Server might address these modules. But in the meantime, the Windows PowerShell compatibility mechanism provides an excellent solution to most of the non-compatible modules.

In *steps 5* and *6*, you examine the details of the `Get-WindowsFeature` command. Inside Windows PowerShell 5.1, this command is a cmdlet. When, in *step 6*, you view the command inside the PowerShell 7.1 console, you can see the command is a (proxy) function. If you view the function definition (`ls function:get-WindowsFeature.definition`), you can see how PowerShell uses the steppable pipeline to run the command in the remote session. For more background on the steppable pipeline, which has been a feature of Windows PowerShell since version 2, see `https://livebook.manning.com/book/windows-powershell-in-action-third-edition/chapter-10/327`.

Exploring the module deny list

During the development of PowerShell 7, it became clear that a few Windows PowerShell modules did not work with PowerShell 7 despite the compatibility solution. Worse, if you attempted to use them, the error messages that resulted were cryptic and non-actionable. One suggested solution was to create a list of modules that were known to not be usable within PowerShell 7. When `Import-Module` attempts to load any module on this list, the failure is more graceful with a cleaner message.

One possible issue that such a deny list might cause would be if the module owners were to release an updated module previously on the deny list that now works. To simplify this situation, PowerShell 7 stores the deny list in a configuration file in the `$PSHOME` folder.

In PowerShell 7.1, there are three modules that are in the deny list:

- `PSScheduledJob`: Windows PowerShell commands to manage the Windows Task Manager service. This module is installed in Windows Server by default.
- `BestPractices`: Windows Server commands to view, run, and view the results of best practice scans of core Windows Server services. This module is installed in Windows Server by default.
- `UpdateServices`: You use this module to manage the **Windows Server Update Services** (**WSUS**) You can install it along with WSUS, or as part of the RSAT tools. This module makes use of object methods to manage the WSUS service, and these methods are not available via the compatibility mechanism. This module also makes use of the **Simple Object Access Protocol** (**SOAP**), which is also not implemented in .NET Core. For this reason, enabling you to manage WSUS natively in PowerShell 7 requires a significant development effort. Microsoft has not committed to undertake this work at this time.

It might be tempting to edit the deny list and to remove modules from it. However, such actions have no practical value. You can import the `BestPractices` module explicitly into a PowerShell 7 session, but the cmdlets in the module fail with more cryptic and non-actionable error messages.

In this recipe, you look at PowerShell's module load deny list and discover how it works in practice.

Getting ready

You run this recipe on `SRV1`, a Windows Server Datacenter Edition server with PowerShell 7 and VS Code installed.

How to do it...

1. Getting the PowerShell configuration file

```
$CFFile = "$PSHOME/powershell.config.json"
Get-Item -Path $CFFile
```

2. Viewing contents

```
Get-Content -Path $CFFile
```

3. Attempting to load a module on the deny list

```
Import-Module -Name BestPractices
```

4. Loading the module, overriding edition check

```
Import-Module -Name BestPractices -SkipEditionCheck
```

5. Viewing the module definition

```
Get-Module -Name BestPractices
```

6. Attempting to use `Get-BpaModel`

```
Get-BpaModel
```

How it works...

In *step 1*, you locate and view file details of PowerShell's configuration file, `powershell.config.json`. This file is in the PowerShell installation folder. You can see the output in the following screenshot:

```
PS C:\Foo> # 1. Getting the PowerShell configuration file
PS C:\Foo> $CFFile = "$PSHOME/powershell.config.json"
PS C:\Foo> Get-Item -Path $CFFile

    Directory: C:\PSPreview

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a---          25/09/2020    16:38            406 powershell.config.json
```

Figure 3.21: Getting the PowerShell configuration file

In *step 2*, you view the contents of this file, which looks like this:

```
PS C:\Foo> # 2. Viewing contents
PS C:\Foo> Get-Content -Path $CFFile
{
    "WindowsPowerShellCompatibilityModuleDenyList":  [
                                                         "PSScheduledJob",
                                                         "BestPractices",
                                                         "UpdateServices"
                                                     ],
    "Microsoft.PowerShell:ExecutionPolicy":  "RemoteSigned"
}
```

Figure 3.22: Viewing the contents of the configuration file

In *step 3*, you attempt to use `Import-Module` to import a module you can see (in the output from *step 2*) is on the module deny list. The output of this step looks like this:

```
PS C:\Foo> # 3. Attempting to load a module in deny list
PS C:\Foo> Import-Module -Name BestPractices
Import-Module: Module 'BestPractices' is blocked from loading using Windows PowerShell
compatibility feature by a 'WindowsPowerShellCompatibilityModuleDenyList' setting in
PowerShell configuration file.
```

Figure 3.23: Attempting to load BestPractices, which is on the deny list

With PowerShell 7, you can force `Import-Module` to ignore the deny list and attempt to load the module in a PowerShell 7 session, as you can see in *step 4*. To do this, you specify the switch `-SkipEditionCheck`. This switch tells PowerShell to import the module without checking compatibility or attempting to load the module in the compatibility session.

With *step 5*, the import appears to work in that `Import-Module` does import the module, as you can see in the output from this step:

```
PS C:\Foo> # 5. Viewing the module definition
PS C:\Foo> Get-Module -Name BestPractices

ModuleType Version  PreRelease Name           ExportedCommands
---------- -------  ---------- ----           ----------------
Manifest   1.0                BestPractices  {Get-BpaModel, Get-BpaResult...}
```

Figure 3.24: Viewing the BestPractices module definition

As you can see in *step 6*, running commands in modules on the deny list is likely to fail. In this case, the cmdlet `Get-BpaModel` makes use of a .NET type that does not exist in .NET Core, rendering the cmdlet unable to function:

```
PS C:\Foo> # 6. Attempting to use Get-BpaModel
PS C:\Foo> Get-BpaModel
Get-BpaModel: Could not load type 'System.Diagnostics.Eventing.EventDescriptor'
from assembly 'System.Core, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089'.
```

Figure 3.25: Attempting to use Get-BpaModel

The compatibility mechanism means many older Windows PowerShell modules are going to be usable by your older scripts. However, some modules (just three, as it turns out) are not usable directly in PowerShell 7 or via the Windows PowerShell compatibility mechanism.

There's more...

In *step 2*, you view the configuration file. This file contains two sets of settings. The first creates the module deny list, and the other sets the starting execution policy. You can use the `Set-ExecutionPolicy` cmdlet to change the execution policy, in which case PowerShell writes an updated version of this file reflecting the updated execution policy. The module deny list in this configuration file contains the three module names whose contents are known not to work in PowerShell 7, as you saw in earlier recipes in this chapter.

As you can see in *steps 4* and *5*, you can import modules that are not compatible with .NET Core. The result is that, although you can import a non-compatible module, the commands in that module are not likely to work as expected. And when they fail, the error message is cryptic and not directly actionable.

In summary, the Windows PowerShell compatibility mechanism enables you to use a large number of additional commands in PowerShell 7. The mechanism is not perfect as it does impose a few limitations, but it is a functional approach in almost all cases. Furthermore, although you can, explicitly bypassing the compatibility mechanism is probably not very useful.

Importing format XML

PowerShell, ever since the very beginning, has displayed objects automatically and with nice-looking output. By default, PowerShell displays objects and properties for any object. It creates a table if the object to be displayed contains fewer than five properties, or it creates a list. PowerShell formats each property by calling the `.ToString()` method for each property.

You or the cmdlet developer can improve the output by using format XML. Format XML is custom-written XML that you store in a `format.ps1XML` file. The format XML file tells PowerShell precisely how to display a particular object type (as a table or a list), which properties to display, what headings to use (for tables), and how to display individual properties.

In Windows PowerShell, Microsoft included several format XML files that you can see in the Windows PowerShell home folder. You can view these by typing `Get-ChileItem $PSHOME/*.format.ps1xml`.

In PowerShell 7, the default format XML is inside the code implementing each cmdlet, instead of being a separate file as in Windows PowerShell. The result is that there are, by default, no `format.ps1XML` files in PowerShell 7's `$PSHOME` folder. As with Windows PowerShell, you can always write your own custom format XML files to customize how PowerShell displays objects by default.

The use of customized format XML is a great solution when you are displaying particular objects, typically from the command line, and the default display is not adequate for your purposes. You can implement format XML that tells PowerShell 7 to display objects as you want them displayed and import the format XML in your profile file.

An alternative to creating format XML is to pipe the objects to `Format-Table` or `Format-List`, specifying the properties that you wish to display. You can even use hash tables with `Format-Table` to define how properties are formatted. Depending on how often you display any given object, format XML can be useful to alter the information you see, especially at the console.

For more information about format XML, see `https://docs.microsoft.com/powershell/module/microsoft.powershell.core/about/about_format.ps1xml`.

Module developers often create format XML to help display objects generated by commands inside the module. The `ServerManager` module, for example, has format XML that enables the pretty output created with `Get-WindowsFeature`. This module is not directly supported by PowerShell 7. You can access the commands in the module thanks to the Windows PowerShell compatibility mechanism described earlier in this chapter. One restriction, by default, of the compatibility mechanism is that it does not import format XML (if it exists). The net result is that output can look different when you run scripts or issue console commands in PowerShell 7 versus Windows PowerShell.

The simple way around this is to import the format XML manually, as you see in this recipe.

Getting ready

You run this recipe on SRV1, a workgroup server running Windows Server Datacenter Edition and on which you have installed PowerShell 7 and VS Code.

How to do it...

1. Importing the ServerManager module

```
Import-Module -Name ServerManager
```

2. Checking a Windows feature

```
Get-WindowsFeature -Name Simple-TCPIP
```

3. Running this command in the compatibility session

```
$S = Get-PSSession -Name 'WinPSCompatSession'
Invoke-Command -Session $S -ScriptBlock {
    Get-WindowsFeature -Name Simple-TCPIP }
```

4. Running this command with formatting in the remote session

```
Invoke-Command -Session $S -ScriptBlock {
                    Get-WindowsFeature -Name Simple-TCPIP |
                      Format-Table}
```

5. Getting path to Windows PowerShell modules

```
$Paths = $env:PSModulePath -split ';'
foreach ($Path in $Paths) {
  if ($Path -match 'system32') {$S32Path = $Path; break}
}
"System32 path: [$S32Path]"
```

6. Displaying path to the format XML for the ServerManager module

```
$FXML = "$S32path/ServerManager"
$FF = Get-ChildItem -Path $FXML\*.format.ps1xml
"Format XML file: [$FF]"
```

7. Updating the format XML

```
Foreach ($F in $FF) {
  Update-FormatData -PrependPath $F.FullName}
```

8. Viewing the Windows Simple-TCPIP feature

```
Get-WindowsFeature -Name Simple-TCPIP
```

9. Adding `Simple-TCPIP` Services

```
Add-WindowsFeature -Name Simple-TCPIP
```

10. Examining the `Simple-TCPIP` feature

```
Get-WindowsFeature -Name Simple-TCPIP
```

11. Using the `Simple-TCPIP` Windows feature

```
Install-WindowsFeature Telnet-Client |
  Out-Null
Start-Service -Name simptcp
```

12. Using the Quote of the Day service

```
Telnet SRV1 qotd
```

How it works...

In *step 1*, inside a PowerShell 7.1 session, you import the `ServerManager` module, with output that looks like this:

```
PS C:\Foo> # 1.Importing the Server Manager module
PS C:\Foo> Import-Module -Name Server Manager
WARNING: Module ServerManager is loaded in Windows PowerShell using
WinPSCompatSession remoting session; please note that all input and
output of commands from this module will be deserialized objects.
If you want to load this module into PowerShell please use
'Import-Module -SkipEditionCheck' syntax
```

Figure 3.26: Importing the ServerManager module

Next, in *step 2*, you examine a Windows feature, the `Simple-TCPIP` services feature, using the `Get-WindowsFeature` command, which produces output that looks like this:

```
PS C:\Foo> # 2. Checking a Windows feature
PS C:\Foo> Get-WindowsFeature -Name Simple-TCPIP

Display Name     Name          Install State
------------     ----          -------------
                 Simple-TCPIP      Available
```

Figure 3.27: Examining the Simple-TCPIP feature

Note that, in the output, the **Display Name** column contains no information. PowerShell uses the format XML to populate the contents of this **Display Name** column. Since the compatibility mechanism does not import the format XML into the PowerShell session, you see a sub-optimal output.

In *step 3*, you use the Windows PowerShell compatibility remoting session to run the `Get-WindowsFeature` cmdlet in the remoting session. The output of this step looks like this:

```
PS C:\Foo> # 3 Running this command in the compatibility session
PS C:\Foo> $S = Get-PSSession -Name 'WinPSCompatSession'
PS C:\Foo> Invoke-Command -Session $S -ScriptBlock {
             Get-WindowsFeature -Name Simple-TCPIP }

Display Name     Name           Install State PSComputerName
------------     ----           ------------- --------------
                 Simple-TCPIP       Available localhost
```

Figure 3.28: Running Get-WindowsFeature in the compatibility session

In both *steps 2* and *3*, the `Get-WindowsFeature` cmdlet produces an object that the PowerShell 7 session formats without the benefit of format XML.

In *step 4*, you run the `Get-WindowsFeature` command to retrieve the feature details then perform formatting in the remote session, with output like this:

```
PS C:\Foo> # 4. Running this command with formatting in the remote session
PS C:\Foo> Invoke-Command -Session $S -ScriptBlock {
             Get-WindowsFeature -Name Simple-TCPIP |
               Format-Table}

Display Name                Name          Install State
------------                ----          -------------
[ ] Simple TCP/IP Services  Simple-TCPIP      Available
```

Figure 3.29: Running Get-WindowsFeature in the remote session

You see the nicely formatted **Display Name** filled, since Windows PowerShell performs the formatting of the output from the `Get-WindowsFeature` cmdlet fully within the compatibility session, where the format XML exists.

In *step 5*, you parse the `PSModulePath` Windows environment variable to discover the default path to the Windows PowerShell modules. This path holds the modules added by core Windows services and includes the `ServerManager` module. The output of these commands looks like this:

```
PS C:\Foo> # 5. Getting path to Windows PowerShell modules
PS C:\Foo> $Paths = $env:PSModulePath -split ';'
PS C:\Foo> foreach ($Path in $Paths) {
             if ($Path -match 'system32') {$S32Path = $Path; break}
           }
PS C:\Foo> "System32 path: [$S32Path]"
System32 path: [C:\Windows\system32\WindowsPowerShell\v1.0\Modules]
```

Figure 3.30: Getting the path to Windows PowerShell modules

In *step 6*, you discover the filename for the format XML for the `ServerManager` module, which produces output like this:

```
PS C:\Foo> # 6. Displaying path to the format XML for Server Manager module
PS C:\Foo> $FXML = "$S32path/ServerManager"
PS C:\Foo> $FF = Get-ChildItem -Path $FXML\*.format.ps1xml
PS C:\Foo> "Format XML file: [$FF]"
Format XML file: [C:\Windows\system32\WindowsPowerShell\v1.0\Modules\ServerManager\Feature.format.ps1xml]
```

Figure 3.31: Retrieving the path to the format XML for ServerManager

Most, but not all, Windows modules have a single format XML file.

In *step 7*, which produces no output, you update the format data for the current Windows PowerShell session. If a module has multiple format XML files, this approach ensures they are all imported.

In *step 8*, you view a Windows feature, the Simple TCP/IP service, which has the feature name `Simple-TCPIP`. The output from this command looks like this:

```
PS C:\Foo> # 8. Viewing the Windows Simple-TCPIP feature
PS C:\Foo> Get-WindowsFeature -Name Simple-TCPIP

Display Name                Name            Install State
------------                ----            -------------
[ ] Simple TCP/IP Services  Simple-TCPIP        Available
```

Figure 3.32: Viewing Simple-TCPIP

In *step 9*, you add the `Simple-TCPIP` Windows feature to `SRV1`, which produces output like this:

```
PS C:\Foo> # 9. Adding Simple-TCP Services
PS C:\Foo> Add-WindowsFeature -Name Simple-TCPIP
Success Restart Needed Exit Code   Feature Result
------- -------------- ---------   --------------
True    No             Success     {Simple TCP/IP Services
```

Figure 3.33: Adding Simple-TCPIP to SRV1

In the next step, *step 10*, you view the `Simple-TCPIP` Windows feature, which now looks like this:

```
PS C:\Foo> # 10. Examining Simple-TCPIP feature
PS C:\Foo> Get-WindowsFeature -Name Simple-TCPIP

Display Name                Name          Install State
------------                ----          -------------
[X] Simple TCP/IP Services  Simple-TCPIP      Installed
```

Figure 3.34: Viewing Simple-TCPIP

To test the `Simple-TCPIP` feature, you must install the Telnet client. Then you start the service. You perform both actions in *step 11*, which produces no output.

In *step 12*, you use the **Quote Of The Day** (**QOTD**) protocol. You can use QOTD for debugging connectivity issues, as well as measuring network performance. RFC 8965 defines the QOTD protocol; you can view the protocol definition at `https://tools.ietf.org/html/rfc865` and read more about using QOTD at `https://searchcio.techtarget.com/tip/Quote-of-the-Day-A-troubleshooting-networking-protocol`.

There's more...

In *step 1*, you import the `ServerManager` module. This ensures that the Windows compatibility remoting session is set up. If you have just run some of the earlier recipes in this chapter in the same PowerShell session, this step is redundant as the compatibility remoting session has already been set up and the `ServerManager` module is already loaded.

In *step 2*, you see that the output from `Get-WindowsFeature` does not populate the **Display Name** column. Likewise, in *step 3*. In both of these steps, PowerShell performs basic default formatting. In *step 4*, you get the details of the Windows feature and perform the formatting all in the remote session. As you can see, in the remote session, formatting makes use of format XML to produce superior output. In this case, the format XML populated the **Display Name** field and added an indication of whether the feature is installed (or not).

In *steps 5* and *6*, you discover the Windows PowerShell default modules folder and find the name of the format XML for this module. Having discovered the filename for the format XML, in *step 7*, you import this format information. With the format XML imported, in *step 8*, you run the `Get-WindowsFeature` cmdlet to view a Windows feature. Since you have imported the format XML, the result is the output that is the same as you saw in *step 4*.

This recipe uses the `Simple-TCPIP` feature to demonstrate how you can use a command in PowerShell 7.1 and get the same output you are used to from using Windows PowerShell. The simple TCIP/IP services provided by this function are very old-school protocols, such as QOTD. In production, these services are of potentially no value, and not adding them is a best practice.

Leveraging compatibility

In this chapter so far, you have looked at the issue of compatibility between Windows PowerShell and PowerShell 7. You have examined the new features in PowerShell 7 and looked at the Windows PowerShell compatibility mechanism.

The compatibility mechanism allows you to use incompatible Windows PowerShell cmdlets inside a PowerShell session. Incompatible Windows PowerShell cmdlets/modules rely on features that, while present in the full .NET CLR, are not available in .NET Core 5.0 (and are unlikely ever to be added to .NET Core). For example, the `Get-WindowsFeature` cmdlet uses a .NET type `System.Diagnostics.Eventing.EventDescriptor`, as you saw earlier. Although the cmdlet cannot run natively in PowerShell 7, the compatibility mechanism allows you to make use of the cmdlet's functionality.

When `Import-Module` begins loading an incompatible module, it checks to see if a remoting session with the name `WinPSCompatSession` exists. If that remoting session exists, PowerShell makes use of it. If the session does not exist, `Import-Module` creates a new remoting session with that name. PowerShell 7 then imports the module in the remote session and creates proxy functions in the PowerShell 7 session.

Once `Import-Module` has created the remoting session, PowerShell uses that single session for all future use, so loading multiple modules utilizes a single remoting session.

Getting ready

You run this recipe on `SRV1`, on which you have installed PowerShell 7 and VS Code. `SRV1` is a workgroup server running Windows Server Datacenter Edition.

How to do it...

1. Creating a session using the reserved name

```
$S1 = New-PSSession -Name WinPSCompatSession -ComputerName SRV1
```

2. Getting loaded modules in the remote session

```
Invoke-Command -Session $S1 -ScriptBlock {Get-Module}
```

3. Loading the `ServerManager` module in the remote session

```
Import-Module -Name ServerManager -WarningAction SilentlyContinue |
  Out-Null
```

4. Getting loaded modules in the remote session

```
Invoke-Command -Session $S1 -ScriptBlock {Get-Module}
```

5. Using `Get-WindowsFeature`

```
Get-WindowsFeature -Name PowerShell
```

6. Closing remoting sessions and removing module from current PS7 session

```
Get-PSSession | Remove-PSSession
Get-Module -Name ServerManager | Remove-Module
```

7. Creating a default compatibility remoting session

```
Import-Module -Name ServerManager -WarningAction SilentlyContinue
```

8. Getting the new remoting session

```
$S2 = Get-PSSession -Name 'WinPSCompatSession'
$S2
```

9. Examining modules in `WinPSCompatSession`

```
Invoke-Command -Session $S2 -ScriptBlock {Get-Module}
```

How it works...

In *step 1*, you create a new remoting session using the session name `WinPSCompatSession`. This step produces no output, but it does create a remoting session to a Windows PowerShell endpoint. PowerShell 7 holds the name of the endpoint that `New-PSSession` uses, unless you use the parameter `-ConfigurationName` to specify an alternate configuration name when you create the new remoting session.

In *step 2*, you run the `Get-Module` cmdlet to return the modules currently loaded in the remoting session you established in *step 1*. PowerShell generates no output for this step, indicating that there are no modules so far imported into the remoting session.

With *step 3*, you import the `ServerManager` module. Since this module is not compatible with PowerShell 7, PowerShell uses the Window PowerShell compatibility session you created earlier. `Import-Module` only checks to see if there is an existing remoting session with the name `WinPSCompatSession`. This step generates no output.

In *step 4*, you recheck the modules loaded into the remote session. As you can see in the output, this command discovers two modules loaded in the remoting session, one of which is the Windows PowerShell `ServerManager` module. The output from this step looks like this:

```
PS C:\Foo> # 4. Getting loaded modules in remote session
PS C:\Foo> Invoke-Command -Session $S1 -ScriptBlock {Get-Module}

ModuleType Version PreRelease Name                        ExportedCommands                        PSComputerName
---------- ------- ---------- ----                        ----------------                        --------------
Manifest   3.1.0.0            Microsoft.PowerShell.Utility {Remove-Variable, Remove-Event, Add-Me… SRV1
Script     2.0.0.0            ServerManager               {Uninstall-WindowsFeature, Get-Windows… SRV1
```

Figure 3.35: Getting loaded modules in the remote session

In *step 5*, you invoke `Get-WindowsFeature` to discover the PowerShell feature. As you can see in the output, this feature is both available and installed in `SRV1`:

```
PS C:\Foo> # 5. Using Get-WindowsFeature
PS C:\Foo> Get-WindowsFeature -Name PowerShell

Display Name   Name        Install State
------------   ----        -------------
               PowerShell      Installed
```

Figure 3.36: Invoking Get-WindowsFeature

In *step 6*, you close the remoting session that you created earlier in this recipe and remove all loaded modules. This step generates no output. In *step 7*, you import the (non-PowerShell 7-compatible) `ServerManager` module. As you have seen previously, this command creates a compatibility session, although there is no output from this step.

In *step 8*, you get and then display the remoting session, which produces output like this:

```
PS C:\Foo> # 8. Getting loaded modules in remote session
PS C:\Foo> $S2 = Get-Pssession -Name 'WinPSCompatSession'
PS C:\Foo> $S2

 Id Name            Transport ComputerName  ComputerType   State   ConfigurationName  Availability
 -- ----            --------- ------------  ------------   -----   -----------------  ------------
 14 WinPSCompatSes… Process   localhost     RemoteMachine  Opened                        Available
```

Figure 3.37: Getting loaded modules in the remote session

In *step 9*, you check to see the modules loaded in the Windows compatibility session (which you create in *step 7* by importing a module). The output from this step looks like this:

```
PS C:\Foo> # 9. Examining modules in WInPSCompatSession
PS C:\Foo> Invoke-Command -Session $S2 -ScriptBlock {Get-Module}

ModuleType Version  PreRelease Name                         ExportedCommands                                  PSComputerName
---------- -------  ---------- ----                         ----------------                                  --------------
Manifest   3.1.0.0             Microsoft.PowerShell.Utility {Remove-Variable, Remove-Event, Add-Member, Debug-Runspace… localhost
Script     2.0.0.0             ServerManager                {Uninstall-WindowsFeature, Get-WindowsFeature, Disable-Ser… localhost
```

Figure 3.38: Checking modules loaded in the Windows compatibility session

There's more...

In this recipe, you create two PowerShell remoting sessions. In *step 1*, you create it explicitly by using `New-PSSession`. Later, in *step 7*, you call `Import-Module` to import an incompatible module. `Import-Module` then creates a remoting session, since one does not exist.

As you can see in this recipe, so long as there is a remoting session with the name `WinPSCompatSession` in PowerShell 7, `Import-Module` uses that to attempt to load, in this case, the `ServerManager` module.

One potential use for the approach shown in this recipe might be when you wish to create a constrained endpoint for use with delegated administration that relies on older Windows PowerShell cmdlets. You could create the endpoint configuration details and then, in the scripts, create a remoting session using the customized endpoint and name the session `WinPSCompatSession`.

4

Using PowerShell 7 in the Enterprise

In this chapter, we cover the following recipes:

- Installing RSAT tools on Windows Server
- Exploring package management
- Exploring PowerShellGet and the PS Gallery
- Creating a local PowerShell repository
- Establishing a script signing environment
- Working with shortcuts and the PSShortcut module
- Working with archive files

Introduction

For many users, PowerShell and the commands that come with Windows Server and the Windows client are adequate for their use. But in larger organizations, there are additional things you need in order to manage your IT infrastructure. This includes tools that can make your job easier.

You need to create an environment in which you can use PowerShell to carry out the administration. This environment includes ensuring you have all the tools you need close to hand, and making sure the environment is as secure as possible. There are also techniques and tools that make life easier for an administrator in a larger organization. And of course, those tools can be very useful for any IT professional.

To manage Windows roles and features, as well as manage Windows itself with PowerShell, you can use modules of PowerShell commands. You can manage most Windows features with PowerShell, using the tools that come with the feature in question. You can install the tools with a feature – installing the `ActiveDirectory` module when you install Active Directory on a system.

You can also install the management tools separately and manage features remotely. The **Remote Server Administration Tools** (**RSAT**) allow you to manage Windows roles and features. In the *Installing RSAT tools on Windows Server* recipe, you investigate the RSAT tools and how you can install them in Windows Server.

Although the RSAT tools provide much excellent functionality, they do not allow you to do everything you might wish. To fill the gaps, the PowerShell community has created many additional third-party modules/commands which you can use to augment the modules provided by Microsoft. To manage these, you need package management, which you examine in *Exploring package management*. In the *Exploring PowerShellGet and the PS Gallery* recipe, you look at one source of modules and examine how to find and utilize modules contained in the PowerShell Gallery.

PowerShell enables you to use digital signing technology to sign a script and to ensure that your users can only run signed scripts. In *Establishing a script signing environment*, you learn how to sign scripts and use digitally signed scripts.

PowerShell makes use of Microsoft's Authenticode technology to enable you to sign both applications and PowerShell scripts. The signature is a cryptographic hash of the executable or script that's based on an X.509 code signing certificate. The key benefit is the signature provides cryptographic proof that the executable or script has not changed since it was signed. Also, you can use the digital signature to provide **non-repudiation** – that is, the only person who could have signed the file would be a person who had the signing certificate's private key.

In *Working with shortcuts and the PSShortcut module*, you learn how to create and manage shortcuts. A shortcut is a file that points to another file. You can have a link file (with the extension `.LNK`) which provides a shortcut to an executable program. You might have a shortcut to VS Code or PowerShell and place it on the desktop. The second type of shortcut, a URL shortcut, is a file (with a `.URL` extension). You might place a shortcut on a desktop which points to a specific website. PowerShell has no built-in mechanism for handling shortcuts. To establish a link shortcut, you can use the `Wscript.Shell` COM object. You can also use the `PSShortcut` module to create, discover, and manage both kinds of shortcut.

An archive is a file which contains other, usually compressed, files and folders. You can easily create new archive files and expand existing ones. Archives are very useful, both to hold multiple documents and to compress them. You might bundle together documents, scripts, and graphic files and send them as a single archive. You might also use compression to transfer log files – a 100 MB text log file might compress to as little as 3-4 MB, allowing you to send such log files via email. In *Working with archive files*, you create and use archive files.

Installing RSAT tools on Windows Server

The RSAT tools are fundamental to administering the roles and features you can install on Windows Server. Each feature in Windows Server can optionally have management tools, and most do. These tools can include PowerShell cmdlets, functions, and aliases, as well as GUI **Microsoft Management Console** (**MMC**) files. Some features also have older Win32 console applications. For the most part, you do not need the console applications since you can use the cmdlets, but that is not always the case. You may have older scripts that use those console applications.

You can also install the RSAT tools independently of a Windows Server feature on Windows Server. This recipe covers RSAT tool installation on Windows Server 2022.

You can also install the RSAT tools in Windows 10 and administer your servers remotely. The specific method of installing the RSAT tools varies with the specific version of Windows 10 you are using. For earlier Windows 10 editions, you can download the tools here: `https://www.microsoft.com/download/details.aspx?id=45520`.

In later editions of Windows 10, beginning with the Windows 10 October Update, you install the RSAT tools using the "Features on Demand" mechanism inside Windows 10. The URL in the previous paragraph provides fuller details about how to install the RSAT tools on Windows 10.

Getting ready

You run this recipe on `SRV1`, on which you have installed PowerShell 7 and VS Code. `SRV1` is a workgroup server running Windows Server 2022 Datacenter Edition.

How to do it...

1. Displaying counts of available PowerShell commands

```
$CommandsBeforeRSAT = Get-Command
$CmdletsBeforeRSAT = $CommandsBeforeRSAT  |
    Where-Object CommandType -eq 'Cmdlet'
$CommandCountBeforeRSAT = $CommandsBeforeRSAT.Count
$CmdletCountBeforeRSAT  = $CmdletsBeforeRSAT.Count
"On Host: [$(hostname)]"
"Total Commands available before RSAT installed
[$CommandCountBeforeRSAT]"
"Cmdlets available before RSAT installed
[$CmdletCountBeforeRSAT]"
```

2. Getting command types returned by `Get-Command`

```
$CommandsBeforeRSAT |
  Group-Object -Property CommandType
```

3. Checking the object type details

```
$CommandsBeforeRSAT |
  Get-Member |
    Select-Object -ExpandProperty TypeName -Unique
```

4. Getting the collection of PowerShell modules and a count of modules before adding the RSAT tools

```
$ModulesBefore = Get-Module -ListAvailable
```

5. Displaying a count of modules available before adding the RSAT tools

```
$CountOfModulesBeforeRSAT = $ModulesBefore.Count
"$CountOfModulesBeforeRSAT modules available"
```

6. Getting a count of features available on SRV1

```
Import-Module -Name ServerManager -WarningAction SilentlyContinue
$Features  = Get-WindowsFeature
$FeaturesI = $Features | Where-Object Installed
$RsatF     = $Features |
               Where-Object Name -Match 'RSAT'
$RSATFI    = $RSATF |
              Where-Object Installed
```

7. Displaying counts of features installed

```
"On Host [$(hostname)]"
"Total features available      [{0}]"  -f $Features.count
"Total features installed      [{0}]"  -f $FeaturesI.count
"Total RSAT features available [{0}]"  -f $RSATF.count
"Total RSAT features installed [{0}]"  -f $RSATFI.count
```

8. Adding all RSAT tools to SRV1

```
Get-WindowsFeature -Name *RSAT* |
  Install-WindowsFeature
```

9. Rebooting SRV1 (then logging on as the local administrator)

```
Restart-Computer -Force
```

10. Getting details of RSAT tools now installed on SRV1

```
$FSRV1A   = Get-WindowsFeature
$IFSRV1A  = $FSRV1A | Where-Object Installed
$RSFSRV1A = $FSRV1A | Where-Object Installed |
              Where-Object Name -Match 'RSAT'
```

11. Displaying counts of commands after installing the RSAT tools

```
"After Installation of RSAT tools on SRV1"
"$($IFSRV1A.count) features installed on SRV1"
"$($RSFSRV1A.count) RSAT features installed on SRV1"
```

12. Displaying RSAT tools on SRV1

```
$MODS = "$env:windir\system32\windowspowerShell\v1.0\modules"
$SMMOD = "$MODS\ServerManager"
Update-FormatData -PrependPath "$SMMOD\*.format.ps1xml"
Get-WindowsFeature |
  Where-Object Name -Match 'RSAT'
```

How it works...

In *step 1*, you use the `Get-Command` command to obtain all the commands inside all modules on `SRV1`. The step then displays a count of the total number of commands available on `SRV1` and how many actual cmdlets exist on `SRV1` before installing the RSAT tools. The output of this step looks like this:

```
PS C:\Foo> # 1. Displaying counts of available PowerShell commands
PS C:\Foo> $CommandsBeforeRSAT = Get-Command
PS C:\Foo> $CmdletsBeforeRSAT = $CommandsBeforeRSAT  |
             Where-Object CommandType -eq 'Cmdlet'
PS C:\Foo> $CommandCountBeforeRSAT = $CommandsBeforeRSAT.Count
PS C:\Foo> $CmdletCountBeforeRSAT  = $CmdletsBeforeRSAT.Count
PS C:\Foo> "On Host: [$(hostname)]"
PS C:\Foo> "Total Commands available before RSAT installed [$CommandCountBeforeRSAT]"
PS C:\Foo> "Cmdlets available before RSAT installed        [$CmdletCountBeforeRSAT]"
On Host: [SRV1]
Total Commands available before RSAT installed [1829]
Cmdlets available before RSAT installed        [594]
```

Figure 4.1: Displaying counts of available PowerShell commands

In *step 2*, you display a count of the types of commands available thus far on `SRV1`, which looks like this:

```
PS C:\Foo> # 2. Getting command types returned by Get-Command
PS C:\Foo> $CommandsBeforeRSAT |
             Group-Object -Property CommandType

Count Name      Group
----- ----      -----
   58 Alias     {Add-AppPackage, Add-AppPackageVolume, Add-AppProvisionedPackage, Add-ProvisionedAppPackag…
 1177 Function  {A:, Add-BCDataCacheExtension, Add-DnsClientDohServerAddress, Add-DnsClientNrptRule, Add-D…
  594 Cmdlet    {Add-AppxPackage, Add-AppxProvisionedPackage, Add-AppxVolume, Add-BitsFile, Add-Certificat…
```

Figure 4.2: Displaying command types returned by Get-Command

In PowerShell, when you use `Get-Command`, the cmdlet returns different objects to describe the different types of commands. As you saw in the previous step, there are three command types which PowerShell returns in different object classes. You can see the class names for those three command types in the output from *step 3*, which looks like this:

```
PS C:\Foo> # 3. Checking the object type details
PS C:\Foo> $CommandsBeforeRSAT |
             Get-Member |
               Select-Object -ExpandProperty TypeName -Unique
System.Management.Automation.AliasInfo
System.Management.Automation.FunctionInfo
System.Management.Automation.CmdletInfo
```

Figure 4.3: Checking the object type details

In *step 4*, which produces no output, you get all the modules available on SRV1. In *step 5*, you display a count of the number of modules available (79), which looks like this:

```
PS C:\Foo> # 5. Displaying a count of modules available before adding the RSAT tools
PS C:\Foo> $CountOfModulesBeforeRSAT = $ModulesBefore.count
PS C:\Foo> "$CountOfModulesBeforeRSAT modules available prior to adding RSAT"
79 modules available
```

Figure 4.4: Displaying an available module count

In *step 6*, you obtain counts of the features available and features installed, as well as the number of RSAT features available and installed. This step generates no output.

In *step 7*, you display the counts you obtained in *step 6*, which looks like this:

```
PS C:\Foo> # 7. Displaying counts of features installed
PS C:\Foo> "On Host [$(hostname)]"
PS C:\Foo> "Total features available      [{0}]"  -f $Features.count
PS C:\Foo> "Total features installed      [{0}]"  -f $FeaturesI.count
PS C:\Foo> "Total RSAT features available [{0}]"  -f $RSATF.count
PS C:\Foo> "Total RSAT features installed [{0}]"  -f $RSATFI.count

On Host [SRV1]
Total features available      [267]
Total features installed      [15]
Total RSAT features available [50]
Total RSAT features installed [0]
```

Figure 4.5: Displaying counts of features installed

In *step 8*, you get and install all the RSAT features in Windows Server. This process does take a bit of time, and generates output like this:

```
PS C:\Foo> # 8. Adding ALL RSAT tools to SRV1
PS C:\Foo> Get-WindowsFeature -Name *RSAT* |
             Install-WindowsFeature

Success Restart Needed Exit Code      Feature Result
------- -------------- ---------      --------------
True    Yes            SuccessRestar… {BitLocker Drive Encryption, RAS Connection …
WARNING: You must restart this server to finish the installation process.
```

Figure 4.6: Adding all RSAT tools

The installation of these tools requires a restart, as you can see in the preceding screenshot. Thus, in *step 9*, you restart the system. After the restart, you log into `SRV1` as an administrator to continue.

Now that you have added all the RSAT-related Windows features, you can get details of what you installed. In *step 10*, which creates no output, you get details of the features you just installed and the commands they contain. In *step 11*, you display the count of RSAT features not available on `SRV1`, which looks like:

```
PS C:\Foo> # 11. Displaying counts of commands after installing the RSAT tools
PS C:\Foo> "After Installation of RSAT tools on SRV1"
PS C:\Foo> "$($IFSRV1A.count) features installed on SRV1"
PS C:\Foo> "$($RSFSRV1A.count) RSAT features installed on SRV1"
After Installation of RSAT tools on SRV1
76 features installed on SRV1
50 RSAT features installed on SRV1
```

Figure 4.7: Displaying counts of commands after installing the RSAT tools

In *step 12*, you display the RSAT features you installed in an earlier step. The output of this step looks like this:

```
PS C:\Foo> # 12. Displaying RSAT tools on SRV1
PS C:\Foo> $MODS = "$env:windir\system32\windowspowerShell\v1.0\modules"
PS C:\Foo> $SMMOD = "$MODS\ServerManager"
PS C:\Foo> Update-FormatData -PrependPath "$SMMOD\*.format.ps1xml"
PS C:\Foo> Get-WindowsFeature |
             Where-Object Name -Match 'RSAT'

Display Name                                            Name                    Install State
------------                                            ----                    -------------
[X] Remote Server Administration Tools                  RSAT                        Installed
    [X] Feature Administration Tools                    RSAT-Feature-Tools          Installed
        [X] SMTP Server Tools                           RSAT-SMTP                   Installed
        [X] BitLocker Drive Encryption Administration … RSAT-Feature-Tools-Bit…     Installed
            [X] BitLocker Drive Encryption Tools        RSAT-Feature-Tools-Bit…     Installed
            [X] BitLocker Recovery Password Viewer      RSAT-Feature-Tools-Bit…     Installed
        [X] BITS Server Extensions Tools                RSAT-Bits-Server            Installed
        [X] DataCenterBridging LLDP Tools               RSAT-DataCenterBridgin…     Installed
        [X] Failover Clustering Tools                   RSAT-Clustering             Installed
            [X] Failover Cluster Management Tools       RSAT-Clustering-Mgmt        Installed
            [X] Failover Cluster Module for Windows Po… RSAT-Clustering-PowerS…     Installed
            [X] Failover Cluster Automation Server      RSAT-Clustering-Automa…     Installed
            [X] Failover Cluster Command Interface      RSAT-Clustering-CmdInt…     Installed
        [X] Network Load Balancing Tools                RSAT-NLB                    Installed
        [X] Shielded VM Tools                           RSAT-Shielded-VM-Tools      Installed
        [X] SNMP Tools                                  RSAT-SNMP                   Installed
        [X] Storage Migration Service Tools             RSAT-SMS                    Installed
        [X] Storage Replica Module for Windows PowerSh… RSAT-Storage-Replica        Installed
        [X] System Insights Module for Windows PowerSh… RSAT-System-Insights        Installed
        [X] WINS Server Tools                           RSAT-WINS                   Installed
    [X] Role Administration Tools                       RSAT-Role-Tools             Installed
        [X] AD DS and AD LDS Tools                      RSAT-AD-Tools               Installed
            [X] Active Directory module for Windows Po… RSAT-AD-PowerShell          Installed
            [X] AD DS Tools                             RSAT-ADDS                   Installed
                [X] Active Directory Administrative Ce… RSAT-AD-AdminCenter         Installed
                [X] AD DS Snap-Ins and Command-Line To… RSAT-ADDS-Tools             Installed
            [X] AD LDS Snap-Ins and Command-Line Tools  RSAT-ADLDS                  Installed
        [X] Hyper-V Management Tools                    RSAT-Hyper-V-Tools          Installed
        [X] Remote Desktop Services Tools               RSAT-RDS-Tools              Installed
            [X] Remote Desktop Gateway Tools            RSAT-RDS-Gateway            Installed
            [X] Remote Desktop Licensing Diagnoser Too… RSAT-RDS-Licensing-Dia…     Installed
        [X] Windows Server Update Services Tools        UpdateServices-RSAT         Installed
        [X] Active Directory Certificate Services Tools RSAT-ADCS                   Installed
            [X] Certification Authority Management Too… RSAT-ADCS-Mgmt              Installed
            [X] Online Responder Tools                  RSAT-Online-Responder       Installed
        [X] Active Directory Rights Management Service… RSAT-ADRMS                  Installed
        [X] DHCP Server Tools                           RSAT-DHCP                   Installed
        [X] DNS Server Tools                            RSAT-DNS-Server             Installed
        [X] Fax Server Tools                            RSAT-Fax                    Installed
        [X] File Services Tools                         RSAT-File-Services          Installed
            [X] DFS Management Tools                    RSAT-DFS-Mgmt-Con           Installed
            [X] File Server Resource Manager Tools      RSAT-FSRM-Mgmt              Installed
            [X] Services for Network File System Manag… RSAT-NFS-Admin              Installed
        [X] Network Controller Management Tools         RSAT-NetworkController      Installed
        [X] Network Policy and Access Services Tools    RSAT-NPAS                   Installed
        [X] Print and Document Services Tools           RSAT-Print-Services         Installed
        [X] Remote Access Management Tools              RSAT-RemoteAccess           Installed
            [X] Remote Access GUI and Command-Line Too… RSAT-RemoteAccess-Mgmt      Installed
            [X] Remote Access module for Windows Power… RSAT-RemoteAccess-Powe…     Installed
        [X] Volume Activation Tools                     RSAT-VA-Tools               Installed
```

Figure 4.8: Displaying installed RSAT features

There's more...

The output from *step 1* shows there are 1,829 total commands and 594 module-based cmdlets available on `SRV1`, before adding the RSAT tools. The actual number may vary, depending on what additional tools, features, or applications you might have added to `SRV1` or the Windows Server version itself.

In *steps 2* and *3*, you find the kinds of commands available and the object type name PowerShell uses to describe these different command types. When you have the class names, you can use your favourite search engine to discover more details about each of these command types.

Exploring package management

The `PackageManagement` PowerShell module provides tools that enable you to download and install software packages from a variety of sources. The module, in effect, implements a provider interface that software package management systems use to manage software packages.

You can use the cmdlets in the `PackageManagement` module to work with a variety of package management systems.

This module, in effect, is an API to package management providers such as `PowerShellGet`, discussed in the *Exploring PowerShellGet and the PS Gallery* recipe. The primary function of the `PackageManagement` module is to manage the set of software repositories in which package management tools can search, obtain, install, and remove packages. The module enables you to discover and utilize software packages from a variety of sources. The modules in the gallery vary in quality. Some are excellent and are heavily used by the community, while others are less useful or of lower quality. Ensure you look carefully at any third-party module you put into production.

This recipe explores the `PackageManagement` module from `SRV1`.

Getting ready

You run this recipe on `SRV1`, on which you have installed PowerShell 7 and VS Code. `SRV1` is a workgroup server running Windows Server Datacenter Edition.

How to do it...

1. Reviewing the cmdlets in the `PackageManagement` module

```
Get-Command -Module PackageManagement
```

2. Reviewing installed providers with `Get-PackageProvider`

```
Get-PackageProvider |
  Format-Table -Property Name,
                         Version,
                         SupportedFileExtensions,
                         FromTrustedSource
```

3. Examining available package providers

```
$PROVIDERS = Find-PackageProvider
$PROVIDERS |
    Select-Object -Property Name,Summary |
      Format-Table -AutoSize -Wrap
```

4. Discovering and counting available packages

```
$PAGKAGES = Find-Package
"Discovered {0:N0} packages" -f $PACKAGES.Count
```

5. Showing the first five packages discovered

```
$PAGKAGES  |
    Select-Object -First 5 |
      Format-Table -AutoSize -Wrap
```

6. Installing the `Chocolatier` provider

```
Install-PackageProvider -Name Chocolatier -Force
```

7. Verifying `Chocolatier` is in the list of installed providers

```
Get-PackageProvider |
  Select-Object -Property Name,Version
```

8. Discovering packages from `Chocolatier`

```
$Start = Get-Date
$CPackages = Find-Package -ProviderName Chocolatier -Name *
"$($CPackages.Count) packages available from Chocolatey"
$End = Get-Date
```

9. Displaying how long it took to find the packages from the `Chocolatier` provider

```
$Elapsed = $End - $Start
"Took {0:n3} seconds" -f $Elapsed.TotalSeconds
```

How it works...

In *step 1*, you use `Get-Command` to view the commands provided by the `PackageManagement` module, which looks like this:

```
PS C:\Foo> # 1. Reviewing the cmdlets in the PackageManagement module
PS C:\Foo> Get-Command -Module PackageManagement

CommandType     Name                               Version    Source
-----------     ----                               -------    ------
Cmdlet          Find-Package                       1.4.7      PackageManagement
Cmdlet          Find-PackageProvider               1.4.7      PackageManagement
Cmdlet          Get-Package                        1.4.7      PackageManagement
Cmdlet          Get-PackageProvider                1.4.7      PackageManagement
Cmdlet          Get-PackageSource                  1.4.7      PackageManagement
Cmdlet          Import-PackageProvider             1.4.7      PackageManagement
Cmdlet          Install-Package                    1.4.7      PackageManagement
Cmdlet          Install-PackageProvider            1.4.7      PackageManagement
Cmdlet          Register-PackageSource             1.4.7      PackageManagement
Cmdlet          Save-Package                       1.4.7      PackageManagement
Cmdlet          Set-PackageSource                  1.4.7      PackageManagement
Cmdlet          Uninstall-Package                  1.4.7      PackageManagement
Cmdlet          Unregister-PackageSource           1.4.7      PackageManagement
```

Figure 4.9: Viewing the commands provided by the PackageManagement module

In *step 2*, you use the `Get-PackageProvider` cmdlet to discover the installed package providers, which looks like this:

```
PS C:\Foo> # 2. Reviewing installed providers with Get-PackageProvider
PS C:\Foo> Get-PackageProvider |
             Format-Table -Property Name,
                                    Version,
                                    SupportedFileExtensions,
                                    FromTrustedSource

Name          Version SupportedFileExtensions FromTrustedSource
----          ------- ----------------------- -----------------
NuGet         3.0.0.1 {nupkg}                             False
PowerShellGet 2.2.5.0 {}                                  False
```

Figure 4.10: Reviewing the installed package providers

In *step 3*, you use `Find-PackageProvider` to discover any other providers you can use. The output looks like this:

```
PS C:\Foo> # 3. Examining available Package Providers
PS C:\Foo> $PROVIDERS = Find-PackageProvider
PS C:\Foo> $PROVIDERS |
             Select-Object -Property Name,Summary |
               Format-Table -AutoSize -Wrap

Name              Summary
----              -------
PowerShellGet     PowerShell module with commands for discovering, installing, updating and
                  publishing the PowerShell artifacts like Modules, DSC Resources, Role
                  Capabilities and Scripts.
ContainerImage    This is a PackageManagement provider module which helps in discovering,
                  downloading and installing Windows Container OS images.

                  For more details and examples refer to our project site at
                  https://github.com/PowerShell/ContainerProvider.
NanoServerPackage A PackageManagement provider to  Discover, Save and Install Nano Server
                  Packages on-demand
Chocolatier       Package Management (OneGet) provider that facilitates installing Chocolatey
                  packages from any NuGet repository.
WinGet            Package Management (OneGet) provider that facilitates installing WinGet
                  packages from any NuGet repository.
```

Figure 4.11: Examining available package providers

In *step 4*, you discover and count the packages you can find using `Find-Package`, with output like this:

```
PS C:\Foo> # 4. Discovering and counting available packages
PS C:\Foo> $PACKAGES = Find-Package
PS C:\Foo> "Discovered {0:N0} packages" -f $PACKAGES.Count
Discovered 6,009 packages
```

Figure 4.12: Discovering and counting available packages

To illustrate some of the packages you just discovered, in *step 5*, you view the first five packages, which looks like this:

```
PS C:\Foo> # 5. Showing first 5 packages discovered
PS C:\Foo> $PACKAGES  |
             Select-Object -First 5 |
              Format-Table -AutoSize -Wrap

Name                Version Source    Summary
----                ------- ------    -------
SpeculationControl  1.0.14  PSGallery This module provides the ability to query the speculation control
                                      settings for the system.
AzureRM.profile     5.8.3   PSGallery Microsoft Azure PowerShell - Profile credential management cmdlets
                                      for Azure Resource Manager
PSWindowsUpdate     2.2.0.2 PSGallery This module contain cmdlets to manage Windows Update Client.
NetworkingDsc       8.2.0   PSGallery DSC resources for configuring settings related to networking.
PackageManagement   1.4.7   PSGallery PackageManagement (a.k.a. OneGet) is a new way to discover and
                                      install software packages from around the web.
                                       It is a manager or multiplexor of existing package managers (also
                                      called package providers) that unifies Windows package management
                                      with a single Windows PowerShell interface. With PackageManagement,
                                      you can do the following.
                                        - Manage a list of software repositories in which packages can be
                                      searched, acquired and installed
                                        - Discover software packages
                                        - Seamlessly install, uninstall, and inventory packages from one or
                                      more software repositories
```

Figure 4.13: Viewing the first five packages discovered

In *step 6*, you install the `Chocolatier` package provider, which gives you access via the package management to the `Chocolatey` repository. `Chocolatey` is a third-party application repository, although not directly supported by Microsoft. For more information about the `Chocolatey` repository, see `https://chocolatey.org/`.

With *step 7*, you review the packaged providers now available on `SRV1` to ensure Chocolatier is included, which looks like this:

```
PS C:\Foo> # 7.  Verifying Chocolatier is in the list of installed providers
PS C:\Foo> Get-PackageProvider |
             Select-Object -Property Name,Version

Name          Version
----          -------
Chocolatier   1.2.0.0  <--
NuGet         3.0.0.1
PowerShellGet 2.2.5.0
```

Figure 4.14: Verifying Chocolatier is installed

In *step 8*, you discover the packages available via the `Chocolatier` provider and display a count of packages. In this step, you also capture the date and time between the start and finish of finding packages. The output from this step looks like this:

```
PS C:\Foo> # 8. Discovering packages from Chocolatier
PS C:\Foo> $CPackages = Find-Package -ProviderName Chocolatier -Name *
PS C:\Foo> "$($CPackages.Count) packages available from Chocolatey
5961 packages available from Chocolatey
```

Figure 4.15: Discovering Chocolatier packages

In *step 9*, you display the time taken to find the packages on `Chocolatey`, which looks like this:

```
PS C:\Foo> # 9.Displaying how long it took to find the packages from the Chocolatier provider
PS C:\Foo> $Elapsed = $End - $Start
PS C:\Foo> "Took {0:n3} seconds" -f $Elapsed.TotalSeconds
Took 49.337 seconds
```

Figure 4.16: Displaying time taken to find the packages on Chocolatey

There's more...

In *step 4*, you obtain a list of packages (and display a count of discovered packages) and then, in *step 5*, display the first five. The packages you see when you run this step are most likely to change – the package repositories are in a state of near-constant change. If you are looking for packages, the approach in these two steps is helpful. You download the list of packages and store it locally. Then, you discover more about the existing packages without incurring the long time it takes to retrieve the list of packages from any given repository. Later, in *step 9*, you display how long it took to obtain the list of packages from `Chocolatey`. In your environment, this time may vary from that shown here, but it illustrates the usefulness of getting a list of all packages first before diving into discovery.

In *step 6*, you install another package provider, `Chocolatier`. This provider gives you access, via the package management commands, to the `Chocolatey` repository. `Chocolatey` provides you with access to common application platforms and is much like `apt-get` in Linux (but for Windows). As always, be careful when obtaining applications or application components from any third-party repository, since your vendors and partners may not provide full support in the case of an incident.

In *step 9*, you view how long it took to discover packages from one repository. This step shows how long it could take, and the time may vary. Note that the first time you execute this step, the code prompts you to install `choco.exe`.

Exploring PowerShellGet and the PS Gallery

In a perfect world, PowerShell would come with a command that performed every single action any IT professional should ever need or want. But, as Jeffrey Snover (the inventor of PowerShell) says: "To Ship is To Choose." And that means PowerShell itself, as well as some Windows features, may not have every command you need. And that is where the PowerShell community comes in.

Ever since V1 was shipped (and probably before!), community members have been providing add-ons. Some attempts were, being kind, sub-optimal, but still better than nothing. As PowerShell and the community matured, the quality of these add-ons grew.

Getting ready

You run this recipe on `SRV1`, on which you have installed PowerShell 7 and VS Code. `SRV1` is a workgroup server running Windows Server Datacenter Edition.

How to do it...

1. Reviewing the commands available in the `PowerShellGet` module

   ```
   Get-Command -Module PowerShellGet
   ```

2. Discovering `Find-*` cmdlets in the PowerShellGet module

   ```
   Get-Command -Module PowerShellGet -Verb Find
   ```

3. Getting all commands, modules, DSC resources, and scripts:

   ```
   $COM = Find-Command
   $MOD = Find-Module
   $DSC = Find-DscResource
   $SCR = Find-Script
   ```

4. Reporting on results

   ```
   "On Host [$(hostname)]"
   "Commands found:          [{0:N0}]"  -f $COM.count
   "Modules found:           [{0:N0}]"  -f $MOD.count
   "DSC Resources found:     [{0:N0}]"  -f $DSC.count
   "Scripts found:           [{0:N0}]"  -f $SCR.count
   ```

5. Discovering NTFS-related modules

```
$MOD |
  Where-Object Name -match NTFS
```

6. Installing the `NTFSSecurity` module

```
Install-Module -Name NTFSSecurity -Force
```

7. Reviewing module commands

```
Get-Command -Module NTFSSecurity
```

8. Testing the `Get-NTFSAccess` cmdlet

```
Get-NTFSAccess -Path C:\Foo
```

9. Creating a download folder

```
$DLFLDR = 'C:\Foo\DownloadedModules'
$NIHT = @{
  ItemType    = 'Directory'
  Path        = $DLFLDR
  ErrorAction = 'SilentlyContinue'
}
New-Item @NIHT | Out-Null
```

10. Downloading the `CountriesPS` module

```
Save-Module -Name CountriesPS -Path $DLFLDR
```

11. Checking downloaded module

```
Get-ChildItem -Path $DLFLDR -Recurse |
  Format-Table -Property Fullname
```

12. Importing the `CountriesPS` module

```
$ModuleFolder = "$DLFLDR\CountriesPS"
Get-ChildItem -Path $ModuleFolder -Filter *.psm1 -Recurse |
    Select-Object -ExpandProperty FullName -First 1 |
        Import-Module -Verbose
```

13. Checking commands in the module

```
Get-Command -Module CountriesPS
```

14. Using the `Get-Country` command

```
Get-Country -Name 'United Kingdom'
```

How it works...

In *step 1*, you examine the commands within the `PowerShellGet` module, which looks like this:

```
PS C:\Foo> # 1. Reviewing the commands available in the PowerShellGet module
PS C:\Foo> Get-Command -Module PowerShellGet

CommandType     Name                                     Version    Source
-----------     ----                                     -------    ------
Function        Find-Command                             2.2.5      PowerShellGet
Function        Find-DscResource                         2.2.5      PowerShellGet
Function        Find-Module                              2.2.5      PowerShellGet
Function        Find-RoleCapability                      2.2.5      PowerShellGet
Function        Find-Script                              2.2.5      PowerShellGet
Function        Get-CredsFromCredentialProvider          2.2.5      PowerShellGet
Function        Get-InstalledModule                      2.2.5      PowerShellGet
Function        Get-InstalledScript                      2.2.5      PowerShellGet
Function        Get-PSRepository                         2.2.5      PowerShellGet
Function        Install-Module                           2.2.5      PowerShellGet
Function        Install-Script                           2.2.5      PowerShellGet
Function        New-ScriptFileInfo                       2.2.5      PowerShellGet
Function        Publish-Module                           2.2.5      PowerShellGet
Function        Publish-Script                           2.2.5      PowerShellGet
Function        Register-PSRepository                    2.2.5      PowerShellGet
Function        Save-Module                              2.2.5      PowerShellGet
Function        Save-Script                              2.2.5      PowerShellGet
Function        Set-PSRepository                         2.2.5      PowerShellGet
Function        Test-ScriptFileInfo                      2.2.5      PowerShellGet
Function        Uninstall-Module                         2.2.5      PowerShellGet
Function        Uninstall-Script                         2.2.5      PowerShellGet
Function        Unregister-PSRepository                  2.2.5      PowerShellGet
Function        Update-Module                            2.2.5      PowerShellGet
Function        Update-ModuleManifest                    2.2.5      PowerShellGet
Function        Update-Script                            2.2.5      PowerShellGet
Function        Update-ScriptFileInfo                    2.2.5      PowerShellGet
```

Figure 4.17: Reviewing the PowerShellGet module commands

In *step 2*, you discover the commands in the `PowerShellGet` module, which enable you to find resources. These, naturally, use the verb `Find`. The output of this step looks like this:

```
PS C:\Foo> # 2. Discovering Find-* cmdlets in PowerShellGet module
PS C:\Foo> Get-Command -Module PowerShellGet -Verb Find

CommandType  Name                  Version   Source
-----------  ----                  -------   ------
Function     Find-Command          2.2.5     PowerShellGet
Function     Find-DscResource      2.2.5     PowerShellGet
Function     Find-Module           2.2.5     PowerShellGet
Function     Find-RoleCapability   2.2.5     PowerShellGet
Function     Find-Script           2.2.5     PowerShellGet
```

Figure 4.18: Discovering Find commands in the PowerShellGet module

In *step 3*, you use several `Find-*` commands to find key resources in the PowerShell Gallery, which produces no output. In *step 4*, you view a count of each of these resource types:

```
PS C:\Foo> # 4. Reporting on results
PS C:\Foo> "On Host [$(hostname)]"
PS C:\Foo> "Commands found:          [{0:N0}]"  -f $COM.count
PS C:\Foo> "Modules found:           [{0:N0}]"  -f $MOD.count
PS C:\Foo> "DSC Resources found:     [{0:N0}]"  -f $DSC.count
PS C:\Foo> "Scripts found:           [{0:N0}]"  -f $SCR.count

On Host [SRV1]
Commands found:          [103,650]
Modules found:           [6,204]
DSC Resources found:     [1,771]
Scripts found:           [1,241]
```

Figure 4.19: Displaying a count of resource types

In *step 5*, you use the returned list of modules to discover any NTFS-related modules, like this:

```
PS C:\Foo> # 5. Discovering NTFS-related modules
PS C:\Foo> $MOD |
             Where-Object Name -match NTFS

Version   Name                      Repository    Description
-------   ----                      ----------    -----------
4.2.6     NTFSSecurity              PSGallery     Windows PowerShell Module for managing file
                                                  and folder security on NTFS volumes
1.4.1     cNtfsAccessControl        PSGallery     The cNtfsAccessControl module contains DSC
                                                  resources for NTFS access control management.
1.0       NTFSPermissionMigration   PSGallery     This module is used as a wrapper to the popular
                                                  icacls utility to save permissions to a file
```

Figure 4.21: Discovering NTFS-related modules

In *step 6*, you install the `NTFSSecurity` module, which produces no output. In *step 7*, you review the commands in the `NTFSSecurity` module, which produces output like this:

```
PS C:\Foo> # 7. Reviewing module commands
PS C:\Foo> Get-Command -Module NTFSSecurity

CommandType  Name                              Version   Source
-----------  ----                              -------   ------
Cmdlet       Add-NTFSAccess                    4.2.6     NTFSSecurity
Cmdlet       Add-NTFSAudit                     4.2.6     NTFSSecurity
Cmdlet       Clear-NTFSAccess                  4.2.6     NTFSSecurity
Cmdlet       Clear-NTFSAudit                   4.2.6     NTFSSecurity
Cmdlet       Copy-Item2                        4.2.6     NTFSSecurity
Cmdlet       Disable-NTFSAccessInheritance     4.2.6     NTFSSecurity
Cmdlet       Disable-NTFSAuditInheritance      4.2.6     NTFSSecurity
Cmdlet       Disable-Privileges                4.2.6     NTFSSecurity
Cmdlet       Enable-NTFSAccessInheritance      4.2.6     NTFSSecurity
Cmdlet       Enable-NTFSAuditInheritance       4.2.6     NTFSSecurity
Cmdlet       Enable-Privileges                 4.2.6     NTFSSecurity
Cmdlet       Get-ChildItem2                    4.2.6     NTFSSecurity
Cmdlet       Get-DiskSpace                     4.2.6     NTFSSecurity
Cmdlet       Get-FileHash2                     4.2.6     NTFSSecurity
Cmdlet       Get-Item2                         4.2.6     NTFSSecurity
Cmdlet       Get-NTFSAccess                    4.2.6     NTFSSecurity
Cmdlet       Get-NTFSAudit                     4.2.6     NTFSSecurity
Cmdlet       Get-NTFSEffectiveAccess           4.2.6     NTFSSecurity
Cmdlet       Get-NTFSHardLink                  4.2.6     NTFSSecurity
Cmdlet       Get-NTFSInheritance               4.2.6     NTFSSecurity
Cmdlet       Get-NTFSOrphanedAccess            4.2.6     NTFSSecurity
Cmdlet       Get-NTFSOrphanedAudit             4.2.6     NTFSSecurity
Cmdlet       Get-NTFSOwner                     4.2.6     NTFSSecurity
Cmdlet       Get-NTFSSecurityDescriptor        4.2.6     NTFSSecurity
Cmdlet       Get-NTFSSimpleAccess              4.2.6     NTFSSecurity
Cmdlet       Get-Privileges                    4.2.6     NTFSSecurity
Cmdlet       Move-Item2                        4.2.6     NTFSSecurity
Cmdlet       New-NTFSHardLink                  4.2.6     NTFSSecurity
Cmdlet       New-NTFSSymbolicLink              4.2.6     NTFSSecurity
Cmdlet       Remove-Item2                      4.2.6     NTFSSecurity
Cmdlet       Remove-NTFSAccess                 4.2.6     NTFSSecurity
Cmdlet       Remove-NTFSAudit                  4.2.6     NTFSSecurity
Cmdlet       Set-NTFSInheritance               4.2.6     NTFSSecurity
Cmdlet       Set-NTFSOwner                     4.2.6     NTFSSecurity
Cmdlet       Set-NTFSSecurityDescriptor        4.2.6     NTFSSecurity
Cmdlet       Test-Path2                        4.2.6     NTFSSecurity
```

Figure 4.21: Reviewing the NTFSSecurity module commands

In preparation for downloading another module, in *step 9,* you create a new folder to hold the downloaded module. In *step 10*, you download the `CountriesPS` module. Neither of these steps generates output.

In *step 11,* you examine the files that make up the module, which looks like this:

```
PS C:\Foo\> # 11. Checking downloaded module
PS C:\Foo\> Get-ChildItem -Path $DLFLDR -Recurse |
             Format-Table -Property Fullname

FullName
--------
C:\Foo\DownloadedModules\CountriesPS
C:\Foo\DownloadedModules\CountriesPS\1.0.0.0
C:\Foo\DownloadedModules\CountriesPS\1.0.0.0\Public
C:\Foo\DownloadedModules\CountriesPS\1.0.0.0\CountriesPS.psd1
C:\Foo\DownloadedModules\CountriesPS\1.0.0.0\CountriesPS.psm1
C:\Foo\DownloadedModules\CountriesPS\1.0.0.0\Public\Get-Country.ps1
```

Figure 4.22: Examining the CountriesPS module files

In *step 12*, you find the `CountriesPS` module and import it. Because you use the `-Verbose` switch, `Import-Module` produces the additional output you can see here:

```
PS C:\Foo> # 12. Importing the CountriesPS module
PS C:\Foo> $ModuleFolder = "$DLFLDR\CountriesPS"
PS C:\Foo> Get-ChildItem -Path $ModuleFolder -Filter *.psm1 -Recurse |
             Select-Object -ExpandProperty FullName -First 1 |
               Import-Module -Verbose
VERBOSE: Importing function 'Get-Country'.
```

Figure 4.23: Importing the CountriesPS module using the -Verbose switch

In *step 13*, you use `Get-Command` to check the commands available in the `CountriesPS` module, which looks like this:

```
PS C:\Foo\> # 13. Checking commands in the module
PS C:\Foo\> Get-Command -Module CountriesPS

CommandType  Name         Version   Source
-----------  ----         -------   ------
Function     Get-Country  0.0       CountriesPS
```

Figure 4.24: Checking commands available in the CountriesPS module

In the final step in this recipe, *step 14*, you use the `Get-Country` command to return country details for the United Kingdom, which looks like this:

```
PS C:\Foo> # 14. Using the Get-Country command
PS C:\Foo> Get-Country -Name 'United Kingdom'

name          : United Kingdom
topLevelDomain : {.uk}
alpha2Code    : GB
alpha3Code    : GBR
callingCodes  : {44}
capital       : London
altSpellings  : {GB, UK, Great Britain}
region        : Europe
subregion     : Northern Europe
population    : 64800000
latlng        : {54, -2}
demonym       : British
area          : 242900
gini          : 34
timezones     : {UTC-08:00, UTC-05:00, UTC-04:00, UTC-03:00, UTC-02:00, UTC, UTC+01:00, UTC+02:00, UTC+06:00}
borders       : {IRL}
nativeName    : United Kingdom
numericCode   : 826
currencies    : {GBP}
languages     : {en}
translations  : @{de=Vereinigtes Königreich; es=Reino Unido; fr=Royaume-Uni; ja=イギリス; it=Regno Unito}
relevance     : 2.5
```

Figure 4.25: Using the Get-Country command for the United Kingdom

There's more...

In *step 2*, you discover the commands within the `PowerShellGet` module that enable you to find resources in the PS Gallery. There are five types of resources supported:

- **Command**: These are individual commands within the gallery. Using `Find-Command` can be useful to help you discover the name of a module that might contain a command.
- **Module**: These are PowerShell modules; some may not work in PowerShell 7.
- **DSC resource**: These are Windows PowerShell DSC resources. PowerShell 7 does not provide the rich DSC functions and features available with Windows PowerShell.
- **Script**: This is an individual PowerShell script.
- **Role capability**: This was meant for packages that enhance Windows roles, but is not used.

In *steps 3* and *4*, you discover the number of commands, modules, DSC resources, and scripts available in the gallery. Since there is constant activity, the numbers of PowerShell resources you discover are likely different from what you see in this book.

In *step 5*, you search the PS Gallery for modules whose name include the string **"NTFS"**. You could also use the `Find-Command` cmdlet in the PowerShell to look for specific commands that might contain the characters **"NTFS"**.

In *steps 6* through *8*, you make use of the `NTFSSecurity` module in the PowerShell gallery. This module, which you use in later chapters of this book, allows you to manage NTFS **Access Control Lists** (**ACLs**). This module is an excellent example of a useful set of commands that the PowerShell development team could have included, but did not, inside Windows PowerShell or PowerShell 7. But with the `PowerShellGet` module, you can find, download, and leverage the modules in the PowerShell Gallery.

In *steps 9* through *14*, you go through the process of downloading and testing the `CountriesPS` module. These steps show how you can download and use a module without necessarily installing it. The approach shown in these steps is useful when you are examining modules in the Gallery for possible use. The module's command, `Get-Country`, uses a REST interface to the `https://restcountries.eu/` website. The GitHub repository has a set of examples to show you some ways to use `Get-Country`, which you can see at `https://github.com/lazywinadmin/CountriesPS`.

The two modules you examined in this recipe are a tiny part of the PowerShell Gallery. As you discovered, there are thousands of modules, commands, and scripts. It would be fair to say that some of those objects are not of the highest quality and may be no use to you. Others are excellent additions to your module collection, as many recipes in this book demonstrate.

For most IT pros, the PowerShell Gallery is the go-to location for obtaining useful modules that avoid you having to reinvent the wheel. In some cases, you may develop a particularly useful module and then publish it to the PS Gallery to share with others. See `https://docs.microsoft.com/powershell/scripting/gallery/concepts/publishing-guidelines` for guidelines regarding publishing to the PS Gallery. And, while you are looking at that page, consider implementing best practices suggested in any production script you develop.

Creating a local PowerShell repository

In the *Exploring PowerShellGet and the PS Gallery* recipe, you saw how you could download PowerShell modules and more from the PS Gallery. You can install them or save them for investigation. One nice feature is that after you install a module using `Install-Module`, you can later use `Update-Module` to update it.

An alternative to using a public repository is to create a private internal repository. You can then use the commands in the `PowerShellGet` module to find, install, and manage your modules. A private repository allows you to create your modules and put them into a local repository for your IT professionals, developers, or other users to access.

There are several ways of setting up an internal repository. One approach would be to use a third-party tool such as **ProGet** from Inedo (see `https://inedo.com/` for details on ProGet).

A simple way to create a repository is to set up an SMB file share. Then, you use the `Register-PSRepository` command to enable each system to use the `PowerShellGet` commands to view this share as a PowerShell repository. After you create the share and register the repository, you can publish your modules to the new repository using the `Publish-Module` command.

Once you set up a repository, you just need to ensure you use `Register-PSRepository` on any system that wishes to use this new repository, as you can see in this recipe.

Getting ready

You run this recipe on `SRV1`, on which you have installed PowerShell 7 and VS Code. `SRV1` is a workgroup server running Windows Server Datacenter Edition.

How to do it...

1. Creating a repository folder

```
$LPATH = 'C:\RKRepo'
New-Item -Path $LPATH -ItemType Directory | Out-Null
```

2. Sharing the folder

```
$SMBHT = @{
  Name        = 'RKRepo'
  Path        = $LPATH
  Description = 'Reskit Repository.'
  FullAccess  = 'Everyone'
}
New-SmbShare @SMBHT
```

3. Registering the repository as trusted (on `SRV1`)

```
$Path = '\\SRV1\RKRepo'
$REPOHT = @{
  Name               = 'RKRepo'
```

```
  SourceLocation     = $Path
  PublishLocation    = $Path
  InstallationPolicy = 'Trusted'
}
Register-PSRepository @REPOHT
```

4. Viewing configured repositories

```
Get-PSRepository
```

5. Creating an HW module folder

```
$HWDIR = 'C:\HW'
New-Item -Path $HWDIR -ItemType Directory | Out-Null
```

6. Creating an elementary module

```
$HS = @"
Function Get-HelloWorld {'Hello World'}
Set-Alias GHW Get-HelloWorld
"@
$HS | Out-File $HWDIR\HW.psm1
```

7. Testing the module locally

```
Import-Module -Name $HWDIR\HW.PSM1 -Verbose
GHW
```

8. Creating a PowerShell module manifest for the new module

```
$NMHT = @{
  Path              = "$HWDIR\HW.psd1"
  RootModule        = 'HW.psm1'
  Description       = 'Hello World module'
  Author            = 'DoctorDNS@Gmail.com'
  FunctionsToExport = 'Get-HelloWorld'
  ModuleVersion     = '1.0.1'}
New-ModuleManifest @NMHT
```

9. Publishing the module

```
Publish-Module -Path $HWDIR -Repository RKRepo -Force
```

10. Viewing the results of publishing

```
Find-Module -Repository RKRepo
```

11. Checking the repository's home folder

```
Get-ChildItem -Path $LPATH
```

How it works...

In *step 1*, you create a folder on `SRV1` that you plan to use to hold the repository. There is no output from this step. In *step 2*, you create a new SMB share, `RKRepo`, on `SRV1`, which looks like this:

```
PS C:\Foo> # 2. Sharing the folder
PS C:\Foo> $SMBHT = @{
             Name        = 'RKRepo'
             Path        = $LPATH
             Description = 'Reskit Repository'
             FullAccess  = 'Everyone'
           }
PS C:\Foo> New-SmbShare @SMBHT

Name   ScopeName Path      Description
----   --------- ----      -----------
RKRepo *         C:\RKRepo Reskit Repository
```

Figure 4.26: Sharing the folder

Before the commands in the `PowerShellGet` module can use this share as a repository, you must register the repository. You must perform this action on any host that is to use this repository via the commands in the `PowerShellGet` module. In *step 3*, you register the repository, which produces no output.

In *step 4*, you use `Get-PSRepository` to view the repositories you have available, which looks like this:

```
PS C:\Foo> # 4. Viewing configured repositories
PS C:\Foo> Get-PSRepository

Name       InstallationPolicy  SourceLocation
----       ------------------  --------------
PSGallery  Untrusted           https://www.powershellgallery.com/api/v2
RKRepo     Trusted             \\SRV1\RKRepo
```

Figure 4.27: Viewing available repositories

To illustrate how you can utilize a repository, you create a module programmatically. In *step 5*, you create a new folder to hold your working copy of the module. In *step 6,* you create a script module and save it into the working folder. These two steps produce no output.

In *step 7*, you test the HW module by importing the `.PSM1` file directly from the working folder and then using the `GHW` alias. To view the actions that `Import-Module` takes upon importing your new module, you specify the `-Verbose` switch. The output of this step looks like this:

```
PS C:\Foo> # 7. Testing the module locally
PS C:\Foo> Import-Module -Name $HWDIR\HW.PSM1 -Verbose
PS C:\Foo> GHW
VERBOSE: Loading module from path 'C:\HWTEST\HW.PSM1'.
VERBOSE: Exporting function 'Get-HelloWorld'.
VERBOSE: Exporting alias 'GHW'.
VERBOSE: Importing function 'Get-HelloWorld'.
VERBOSE: Importing alias 'GHW'.

Hello World
```

Figure 4.28: Testing the HW module locally

Before you can publish a module to your repository, you must create a module manifest to accompany the script module file. In *step 8*, you use `New-ModuleManifest` to create a manifest for this module. With *step 9*, you publish your output. Both steps produce no output.

In *step 10*, you browse the newly created repository using `Find-Module` and specify the `RKRepo` repository. The output of this step looks like this:

```
PS C:\> # 10. Viewing the results of publishing
PS C:\> Find-Module -Repository RKRepo

Version  Name   Repository  Description
-------  ----   ----------  -----------
1.0.1    HW     RKRepo      Hello World module
```

Figure 4.29: Browsing the newly created repository

In *step 11*, you examine the folder holding the repository. You can see the module's NuGet package in the output, which looks like this:

```
PS C:\> # 11. Checking the repository's home folder
PS C:\> Get-ChildItem -Path $LPATH

    Directory: C:\RKRepo

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a---          27/10/2020    14:58           3461 HW.1.0.1.nupkg
```

Figure 4.30: Checking the repository's home folder

There's more...

In *step 1*, you created a folder on the `C:\` drive to hold the repository's contents. In production, you should consider creating the folder on separate high availability volumes.

In *step 4*, you review the repositories available. By default, you have the PS Gallery available, albeit as an untrusted repository. You also see the `RKRepo` repository. Since you have registered the repository, you see it shown as trusted.

In *step 11*, you examine the folder holding the `RKRepo` repository. After you publish the HW module, the folder contains a file which holds the HW module. The file is stored with a `nupkg` extension, indicating that it is a NuGet package. A NuGet package is a single ZIP file that contains compiled code, scripts, a PowerShell module manifest, and more. The PowerShell Gallery is effectively a set of NuGet packages. For more information on NuGet, see `https://docs.microsoft.com/nuget/what-is-nuget`.

Establishing a script signing environment

You can often find that it is essential to know that an application, or a PowerShell script, has not been modified since it was released. You can use **Windows Authenticode Digital Signatures** for this. Authenticode is a Microsoft code-signing technology that identifies the publisher of Authenticode-signed software. Authenticode also verifies that the software has not been tampered with since it was signed and published.

You can also use Authenticode to digitally sign your script using a PowerShell command. You can then ensure PowerShell only runs digitally signed scripts by setting an execution policy of `AllSigned` or `RemoteSigned`.

After you digitally sign your PowerShell script, you can detect whether any changes were made in the script since it was signed. And by using PowerShell's execution policy, you can force PowerShell to test the script to ensure the digital signature is still valid and only run scripts that succeed. You can set PowerShell to do this either for all scripts (by setting the execution policy to `AllSigned`) or only for scripts you downloaded from a remote site (by setting the execution policy to `RemoteSigned`). Setting the execution policy to `AllSigned` also means that your profile files must be signed, or they do not run.

This sounds a beautiful thing, but it is worth remembering that even if you have the execution policy set to `AllSigned`, it's trivial to run any non-signed script. Simply bring your script into VS Code, select all the text in the script, and then run that selected script. And if an execution policy of `RemoteSigned` is blocking a particular script, you can use the `Unblock-File` cmdlet to, in effect, turn a remote script into a local one. Script signing just makes it a bit harder, but not impossible, to run a script which has no signature or whose signature fails.

Signing a script is simple once you have a digital certificate issued by a **Certificate Authority** (**CA**). You have three options for getting an appropriate code-signing certificate:

- Use a well-known public CA such as Digicert (see `https://www.digicert.com/code-signing` for details of their code-signing certificates)
- Deploy an internal CA and obtain the certificate from your organization's CA
- Use a self-signed certificate

Public certificates are useful but generally not free. You can easily set up your own CA or use self-signed certificates. Self-signed certificates are great for testing out signing scripts and then using them, but possibly inappropriate for production use. All three of these methods can give you a certificate that you can use to sign PowerShell scripts.

This recipe shows how to sign and use digitally signed scripts. The mechanisms in this recipe work on any of the three sources of signing key listed above. For simplicity, you use a self-signed certificate for this recipe.

Getting ready

You run this recipe on SRV1, on which you have installed PowerShell 7 and VS Code. SRV1 is a workgroup server running Windows Server Datacenter Edition.

How to do it...

1. Creating a script-signing self-signed certificate

```
$CHT = @{
  Subject           = 'Reskit Code Signing'
  Type              = 'CodeSigning'
  CertStoreLocation = 'Cert:\CurrentUser\My'
}
New-SelfSignedCertificate @CHT | Out-Null
```

2. Displaying the newly created certificate

```
$Cert = Get-ChildItem -Path Cert:\CurrentUser\my -CodeSigningCert
$Cert |
  Where-Object {$_.SubjectName.Name -match $CHT.Subject}
```

3. Creating and viewing a simple script

```
$Script = @"
  # Sample Script
  'Hello World from PowerShell 7!'
  "Running on [$(Hostname)]"
"@
$Script | Out-File -FilePath C:\Foo\Signed.ps1
Get-ChildItem -Path C:\Foo\Signed.ps1
```

4. Signing your new script

```
$SHT = @{
  Certificate = $cert
  FilePath    = 'C:\Foo\Signed.ps1'
}
Set-AuthenticodeSignature @SHT
```

5. Checking the script after signing

```
Get-ChildItem -Path C:\Foo\Signed.ps1
```

6. Viewing the signed script

```
Get-Content -Path C:\Foo\Signed.ps1
```

7. Testing the signature

```
Get-AuthenticodeSignature -FilePath C:\Foo\Signed.ps1 |
  Format-List
```

8. Running the signed script

```
C:\Foo\Signed.ps1
```

9. Set the execution policy to `AllSigned`

```
Set-ExecutionPolicy -ExecutionPolicy AllSigned -Scope Process
```

10. Running the signed script

```
C:\Foo\Signed.ps1
```

11. Copying certificate to current user Trusted Root store

```
$DestStoreName  = 'Root'
$DestStoreScope = 'CurrentUser'
$Type   = 'System.Security.Cryptography.X509Certificates.X509Store'
$MHT = @{
  TypeName = $Type
  ArgumentList  = ($DestStoreName, $DestStoreScope)
}
$DestStore = New-Object  @MHT
$DestStore.Open(
  [System.Security.Cryptography.X509Certificates.OpenFlags]::ReadWrite)
$DestStore.Add($Cert)
$DestStore.Close()
```

12. Checking the signature

```
Get-AuthenticodeSignature -FilePath C:\Foo\Signed.ps1 |
  Format-List
```

13. Running the signed script

```
C:\Foo\Signed.ps1
```

14. Copying certificate to Trusted Publisher store

```
$DestStoreName  = 'TrustedPublisher'
$DestStoreScope = 'CurrentUser'
$Type   = 'System.Security.Cryptography.X509Certificates.X509Store'
$MHT = @{
  TypeName = $Type
```

```
  ArgumentList  = ($DestStoreName, $DestStoreScope)
}
$DestStore = New-Object  @MHT
$DestStore.Open(
  [System.Security.Cryptography.X509Certificates.OpenFlags]::ReadWrite)
$DestStore.Add($Cert)
$DestStore.Close()
```

15. Running the signed script

```
C:\Foo\Signed.ps1
```

How it works...

In *step 1*, you create a new self-signed code signing certificate and store the certificate in the current user `My` certificate store. Because you pipe the output from `New-SelfSignedCertificate` to `Out-Null`, this step produces no output.

In *step 2*, you retrieve the code-signing certificate from the current user's certificate store, then view the certificate, which looks like this:

```
PS C:\Foo> # 2. Displaying the newly created certificate
PS C:\Foo> $Cert = Get-ChildItem -Path Cert:\CurrentUser\my -CodeSigningCert
PS C:\Foo> $Cert |
             Where-Object {$_.SubjectName.Name -match $CHT.Subject}

   PSParentPath: Microsoft.PowerShell.Security\Certificate::CurrentUser\my

Thumbprint                                Subject              EnhancedKeyUsageList
----------                                -------              --------------------
F0D830F1764CDB36122C6971AB7E917E7225D7C8  CN=Reskit Code Signing  Code Signing
```

Figure 4.31: Displaying the newly created certificate

In *step 3*, you create a simple script that outputs two lines of text, one of which includes the hostname. You can see this script in the following screenshot:

```
PS C:\Foo> # 3. Creating and viewing a simple script
PS C:\Foo> $Script = @"
             # Sample Script
             'Hello World from PowerShell 7!'
             "Running on [$(Hostname)]"
           "@
PS C:\Foo> $Script | Out-File -FilePath C:\Foo\Signed.ps1
PS C:\Foo> Get-ChildItem -Path C:\Foo\Signed.ps1

    Directory: C:\Foo

Mode         LastWriteTime   Length Name
----         -------------   ------ ----
-a---  28/10/2020    09:37       75 Signed.ps1
```

Figure 4.32: Creating and viewing a simple script

Now that you have a script, in *step 4*, you sign the script with the newly created self-signed code-signing certificate. The output from this step looks like this:

```
PS C:\Foo> # 4. Signing your new script
PS C:\Foo> $SHT = @{
             Certificate = $cert
             FilePath    = 'C:\Foo\Signed.ps1'
           }
PS C:\Foo> Set-AuthenticodeSignature @SHT

    Directory: C:\foo

SignerCertificate                         Status        StatusMessage                              Path
-----------------                         ------        -------------                              ----
4510B4F754B1704E7FF72779D7FBDEA6E44D88AE  UnknownError  A certificate chain processed, but termin… signed.ps1
```

Figure 4.33: Signing the new script

In *step 5*, you view the signed script, which looks like this:

```
PS C:\Foo> # 5. Checking script after signing
PS C:\Foo> Get-ChildItem -Path C:\Foo\Signed.ps1

    Directory: C:\Foo

Mode         LastWriteTime   Length Name
----         -------------   ------ ----
-a---  28/10/2020    09:39     2133 Signed.ps1
```

Figure 4.34: Checking the script after signing

In *step 6*, you view the script, including the script's digital signature, which looks like this:

```
PS C:\Foo> # 6. Viewing the signed script
PS C:\Foo> Get-Content -Path C:\Foo\Signed.ps1.

  # Sample Script
  'Hello World from PowerShell 7!'
  "Running on [SRV1]"

# SIG # Begin signature block
# MIIFeQYJKoZIhvcNAQcCoIIFajCCBWYCAQExCzAJBgUrDgMCGgUAMGkGCisGAQQB
# gjcCAQSgWzBZMDQGCisGAQQBgjcCAR4wJgIDAQAABBAfzDtgWUsITrck0sYpfvNR
# AgEAAgEAAgEAAgEAAgEAMCEwCQYFKw4DAhoFAAQUyPNDYU86IzmhHEACQWjqdKz7
# Z9KgggMQMIIDDDCCAfSgAwIBAgIQfGR5i0PPkY1B+bDt+q3dRzANBgkqhkiG9w0B
# AQsFADAeMRwwGgYDVQQDDBNSZXNraXQgQ29kZSBTaWduaW5nMB4XDTIwMTAyODA5
# MjM0N1oXDTIxMTAyODA5NDM0N1owHjEcMBoGA1UEAwwTUmVza2l0IENvZGUgU2ln
# bmluZzCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBALCcyou98FneWeHA
# f5eilGj9JoWL3hGBZV/ftuY1hd9+WyHw4qQijfK4aUxFBu8SrPMUUDp8xvFxv0/I
# aqlsisJXw+TxWEpBTheiStgafdDDx/nEyUi0PpRiHFNCq7zFITnrE/10NtLxSYFK
# cXLx3qjMstqoezxVqtvg8DhTmzPK3GW5DurCoQ+I4NJknw5gfiwh/XyCptTJMSah
# +7fCj4fRk/SCQKCfP10a7dtiXrI4v5VAEvg5faxNelno2JElDoGjONgQsIQbBlH7
# WQjaWlCtBXyxWYi2DZnhxeFbmzCx0c/JC+oY6O60r6Qpsed0zXVMrc58U6zrv1vU
# +OtWUY0CAwEAAaNGMEQwDgYDVR0PAQH/BAQDAgeAMBMGA1UdJQQMMAoGCCsGAQUF
# BwMDMB0GA1UdDgQWBBSkINUNq+xNYwNo3U+8zc2ILQZGVTANBgkqhkiG9w0BAQsF
# AAOCAQEAHoe9W8Gq4dqHhsH0jk+WvzdgrKy2DOrq1OrVn1wuBmL5CcseNhTWdqDV
# xnsQn3TKRoKYc0Inpf4cVwuzuAoFN8l8g+TkYNdE+sz36OmVYXFGWXj87sGPCADa
# qj6MtUadacG+Sa82GVD5uponQlQ6La2FplwzPqvaRKJFq7LhbLEJVotDB2Eiz9IX
# GVC8cUy8H/qM/bu2Xqwh4OaGF15Bwbxeev2Ye1e8lrtMhAS/KBJnl9qi9Pgf6tzc
# UW+hI5uQ5lDuhbHxs3AXSZDoFBjKMAAa3YcxVpv0Dm3+85nX7fsjHcOnTxWYSI0W
# fJs1PjUgfaymmYdsUq+fTer+s9o3VTGCAdMwggHPAgEBMDIwHjEcMBoGA1UEAwwT
# UmVza2l0IENvZGUgU2lnbmluZwIQfGR5i0PPkY1B+bDt+q3dRzAJBgUrDgMCGgUA
# oHgwGAYKKwYBBAGCNwIBDDEKMAigAoAAoQKAADAZBgkqhkiG9w0BCQMxDAYKKwYB
# BAGCNwIBBDAcBgorBgEEAYI3AgELMQ4wDAYKKwYBBAGCNwIBFTAjBgkqhkiG9w0B
# CQQxFgQU1j62dOU9ZLPKJw6/OtX3L51s3UkwDQYJKoZIhvcNAQEBBQAEggEAUqTw
# 3qhygbK7bbDM97V9U6zO3H8WXuxgPvpvahpzS8SZ0EKt+Kc9LxYFcKsmKv6YT5Dh
# +bKu3dYK0gctW39dBQb+tiOW18kTB7PlJKWaYd/bO/BtV5DbMTEKR3tBnm7eSOOW
# aYDDUsQifxOnHlcpFY8aUjoJP4AcA456qnqe6dSu+SgCcNU3anAT60+Dv9viL7sU
# gL+Q9QIfLSg2t824Kh0eMjNEmJDitRY/ekCIkO2JEE6D2S09wIyJhv0hEmHENpqx
# TFrqwuBxwxkNzeaxEp3NrcAbKKuYGnzuEogk4xqPnLkeEoPRl0z2c6LD03A38wJj
# sbvY1Pitzybp+vmbmQ==
# SIG # End signature block
```

Figure 4.35: Viewing the signed script

In *step 7*, you use the `Get-AuthenticodeSignature` cmdlet to test the digital signature, which looks like this:

```
PS C:\Foo> # 7. Testing the signature
PS C:\Foo> Get-AuthenticodeSignature -FilePath C:\Foo\Signed.ps1 |
             Format-List

SignerCertificate      : [Subject]
                           CN=Reskit Code Signing

                         [Issuer]
                           CN=Reskit Code Signing

                         [Serial Number]
                           23FBA3EA5E75CBBA41C34A283CDDC425

                         [Not Before]
                           28/10/2020 09:23:47

                         [Not After]
                           28/10/2021 10:43:47

                         [Thumbprint]
                           4510B4F754B1704E7FF72779D7FBDEA6E44D88AE

TimeStamperCertificate :
Status                 : UnknownError
StatusMessage          : A certificate chain processed, but terminated in a root certificate
                         which is not trusted by the trust provider.
Path                   : C:\Foo\Signed.ps1
SignatureType          : Authenticode
IsOSBinary             : False
```

Figure 4.36: Testing the signature using Get-AuthenticodeSignature

In *step 8*, you run the signed script, which looks like this:

```
PS C:\Foo> # 8. Running the signed script
PS C:\Foo> C:\Foo\Signed.ps1
Hello World from PowerShell 7!
Running on [SRV1]
```

Figure 4.37: Running the signed script

In *step 9*, you set the execution policy to `AllSigned` to configure PowerShell to ensure that any script you run must be signed in the current session. There is no output from this step.

In *step 10*, you attempt to run the script, resulting in an error, which looks like this:

```
PS C:\Foo> # 10. Running the signed script
PS C:\Foo> C:\Foo\Signed.ps1
C:\Foo\Signed.ps1: File C:\Foo\Signed.ps1 cannot be loaded. A certificate chain processed,
but terminated in a root certificate which is not trusted by the trust provider..
```

Figure 4.38: Attempting to run the script

In *step 11*, you copy the certificate to the current user's Trusted Root store. For security reasons, PowerShell pops up a confirmation dialog which you must agree to in order to complete the copy, which looks like this:

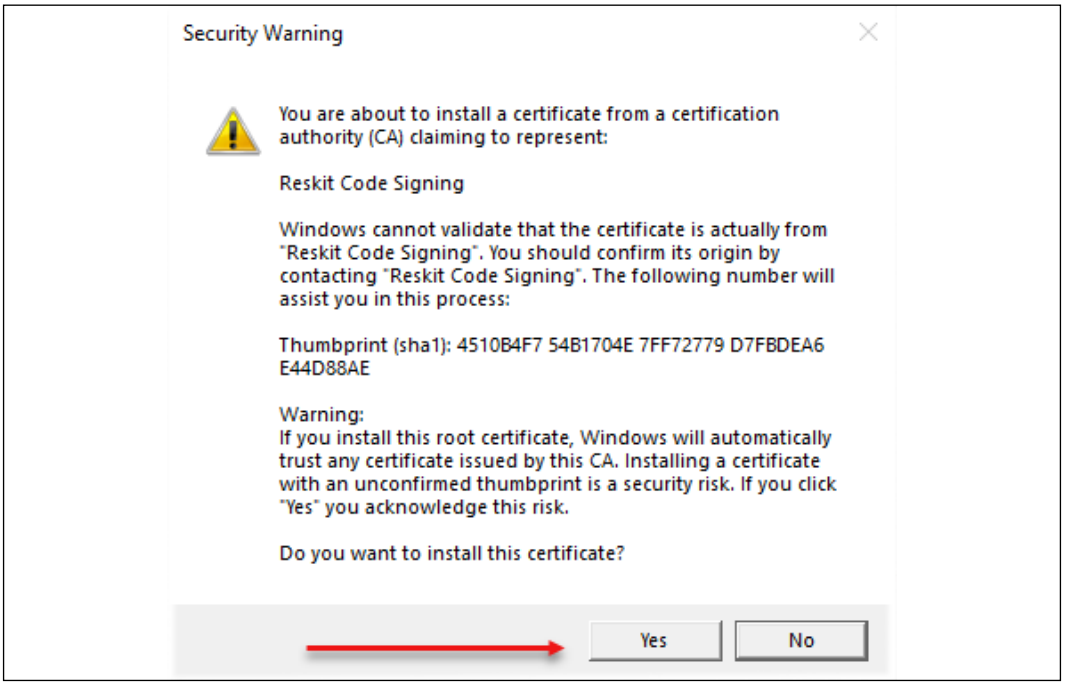


Figure 4.39: Copying the certificate to the current user's Trusted Root store

In *step 12*, you recheck the signature of the file, which looks like this:

```
PS C:\Foo> # 12. Checking the signature
PS C:\Foo> Get-AuthenticodeSignature -FilePath C:\Foo\Signed.ps1 |
             Format-List


SignerCertificate      : [Subject]
                           CN=Reskit Code Signing

                         [Issuer]
                           CN=Reskit Code Signing

                         [Serial Number]
                           7C64798B43CF918D41F9B0EDFAADDD47

                         [Not Before]
                           28/10/2020 09:23:47

                         [Not After]
                           28/10/2021 10:43:47

                         [Thumbprint]
                           4510B4F754B1704E7FF72779D7FBDEA6E44D88AE

TimeStamperCertificate :
Status                 : Valid
StatusMessage          : Signature verified.
Path                   : C:\Foo\Signed.ps1
SignatureType          : Authenticode
IsOSBinary             : False
```

Figure 4.40: Checking the script's digital signature

In *step 13*, you attempt to run the script again, but with a different result, like this:

```
PS C:\Foo> # 13. Running the signed script
PS C:\Foo> C:\Foo\Signed.ps1
Do you want to run software from this untrusted publisher?
File C:\Foo\Signed.ps1 is published by CN=Reskit Code Signing and is
not trusted on your system. Only run scripts from trusted publishers.
[V] Never run [D] Do not run [R] Run once [A] Always run [?] Help (default is "Do not run"):
```

Figure 4.41: Running the signed script

Although the certificate is from a (now) trusted CA, it is not from a trusted publisher. To resolve that, you copy the certificate into the Trusted Publishers store, in *step 14*, which generates no output.

Now that you have the certificate in both the trusted root store and the trusted publisher store (for the user), in *step 15*, you rerun the script, with output like this:

```
PS C:\Foo> # 15. Running the signed script
PS C:\Foo> C:\Foo\Signed.ps1
Hello World from PowerShell 7!
Running on [SRV1]
```

Figure 4.42: Running the script

There's more...

In this recipe, you begin by creating a code signing certificate and using that to sign a script. By default, Windows and PowerShell do not trust self-signed code signing certificates.

To enable PowerShell to trust the signature, you copy the code-signing certificate to the current user's root CA store, which has the effect of making the code-signing certificate trusted for the current user. You do this in *step 11*, and note that a pop-up is generated. You must also copy the certificate to the trusted publisher store, which you do in *step 14*.

In a production environment, you would obtain a code-signing certificate from a trusted CA and manage the trusted certificate stores carefully. For enterprise environments, you could set up a CA and ensure users auto-enrol for root CA certificates. With auto-enrolment, PowerShell (and Windows) can trust the certificates issued by the CA.

As an alternative, you can use a third-party CA, such as DigiCert, to obtain code-signing certificates which, by default, are trusted by Windows. Microsoft's Trusted Root Program helps to distribute trusted root CA certificates. For more information on DigiCert's code signing certificates, see `https://digicert.leaderssl.com/suppliers/digicert/products#codesigning-products`. For details on Microsoft's Trusted Root program, see `https://docs.microsoft.com/security/trusted-root/program-requirements`.

Working with shortcuts and the PSShortcut module

A shortcut is a file that contains a pointer to another file or URL. You can place a shell link shortcut to some executable program, such as PowerShell, on your Windows desktop. When you click the shortcut in Windows Explorer, Windows runs the target program. You can also create a shortcut to a URL.

Shell link shortcuts have the extension `.LNK`, while URL shortcuts have the `.URL` extension. Internally, a file shortcut has a binary structure which is not directly editable. For more details on the internal format, see `https://docs.microsoft.com/openspecs/windows_protocols/ms-shllink/`.

The URL shortcut is a text document that you can edit with VS Code or Notepad. For more details on the URL shortcut file format, see `http://www.lyberty.com/encyc/articles/tech/dot_url_format_-_an_unofficial_guide.html`.

There are no built-in commands to manage shortcuts in PowerShell 7. As you saw earlier in this book, you can use older COM objects to create shortcuts. A more straightforward way is to use the `PSShortcut` module, which you can download from the PowerShell Gallery.

In this recipe, you discover shortcuts on your system and create shortcuts both to an executable file and a URL.

Getting ready

You run this recipe on `SRV1`, on which you have installed PowerShell 7 and VS Code. `SRV1` is a workgroup server running Windows Server Datacenter Edition.

How to do it...

1. Finding the `PSShortcut` module

```
Find-Module -Name '*shortcut'
```

2. Installing the `PSShortcut` module

```
Install-Module -Name PSShortcut -Force
```

3. Reviewing the `PSShortcut` module

```
Get-Module -Name PSShortCut -ListAvailable |
  Format-List
```

4. Discovering commands in the `PSShortcut` module

```
Get-Command -Module PSShortcut
```

5. Discovering all shortcuts on `SRV1`

```
$SHORTCUTS = Get-Shortcut
"Shortcuts found on $(hostname): [{0}]" -f $SHORTCUTS.Count
```

6. Discovering `PWSH` shortcuts

```
$SHORTCUTS | Where-Object Name -match '^PWSH'
```

7. Discovering URL shortcut

```
$URLSC = Get-Shortcut -FilePath *.url
$URLSC
```

8. Viewing the content of the shortcut

```
$URLSC | Get-Content
```

9. Creating a URL shortcut

```
$NEWURLSC  = 'C:\Foo\Google.url'
$TARGETURL = 'https://google.com'
New-Item -Path $NEWURLSC | Out-Null
Set-Shortcut -FilePath $NEWURLSC -TargetPath $TARGETURL
```

10. Using the URL shortcut

```
& $NEWURLSC
```

11. Creating a file shortcut

```
$CMD  = Get-Command -Name notepad.exe
$NP   = $CMD.Source
$NPSC = 'C:\Foo\NotePad.lnk'
New-Item -Path $NPSC | Out-Null
Set-Shortcut -FilePath $NPSC -TargetPath $NP
```

12. Using the shortcut

```
& $NPSC
```

How it works...

In *step 1*, you use the `Find-Module` command to discover modules in the PowerShell Gallery whose name ends with "shortcut". The output from this step looks like this:

```
PS C:\Foo> # 1. Finding the PSShortcut module
PS C:\Foo> Find-Module -Name '*ShortCut*'

Version     Name                          Repository   Description
-------     ----                          ----------   -----------
2.0.0       DSCR_Shortcut                 PSGallery    PowerShell DSC Resource to create shortc…
2.1.0       Remove-iCloudPhotosShortcut   PSGallery    A powershell module for removing iCloud …
1.0.6       PSShortcut                    PSGallery    This module eases working with Windows s…
```

Figure 4.43: Finding the PSShortcut module

In *step 2*, you use the `Install-Module` command to install the `PSShortcut` module. This step produces no output.

Once you have installed the `PSShortcut` module, in *step 3*, you use the `Get-Module` command to find out more about the `PSShortcut` module. The output of this step looks like this:

```
PS C:\Foo> # 3. Reviewing PSShortcut module
PS C:\Foo> Get-Module -Name PSShortCut -ListAvailable |
             Format-List

Name              : PSShortcut
Path              : C:\Users\Administrator\Documents\PowerShell\Modules\PSShortcut\1.0.6\PSShortcut.psd1
Description       : This module eases working with Windows shortcuts (LNK and URL) files.
ModuleType        : Script
Version           : 1.0.6
PreRelease        :
NestedModules     : {}
ExportedFunctions : {Get-Shortcut, Set-Shortcut}
ExportedCmdlets   :
ExportedVariables :
ExportedAliases   :
```

Figure 4.44: Reviewing the PSShortcut module

In *step 4*, you discover the commands provided by the `PSShortcut` module, which looks like this:

```
PS C:\Foo> # 4. Discovering commands in PSShortcut module
PS C:\Foo> Get-Command -Module PSShortcut

CommandType  Name          Version  Source
-----------  ----          -------  ------
Function     Get-Shortcut  1.0.6    PSShortcut
Function     Set-Shortcut  1.0.6    PSShortcut
```

Figure 4.45: Discovering commands in the PSShortcut module

In *step 5*, you use `Get-Shortcut` to find all the link file shortcuts on `SRV1` and save them in a variable. You then display a count of how many you found, with output that looks like this:

```
PS C:\Foo> # 5. Discovering all shortcuts on SRV1
PS C:\Foo> $SHORTCUTS = Get-Shortcut
PS C:\Foo> "Shortcuts found on $(hostname): [{0}]" -f $SHORTCUTS.Count
Shortcuts found on SRV1: [249]
```

Figure 4.46: Discovering all shortcuts on SRV1

In *step 6*, you examine the set of link file shortcuts on `SRV1` to find those that point to `PWSH` (that is, a shortcut to PowerShell 7), which looks like this:

```
PS C:\Foo> # 6. Discovering PWSH shortcuts
PS C:\Foo> $SHORTCUTS | Where-Object Name -match '^PWSH'

    Directory: C:\Users\Administrator\AppData\Roaming\Microsoft\
               Internet Explorer\Quick Launch\User Pinned\TaskBar

Mode         LastWriteTime   Length Name
----         -------------   ------ ----
-a---  05/10/2020    16:25     1030 pwsh.lnk
-a---  05/10/2020    16:25      603 pwshdaily.lnk
-a---  05/10/2020    16:25      588 pwshpreview.lnk
```

Figure 4.47: Discovering PWSH shortcuts

In *step 7*, you use `Get-Shortcut` to discover any URL shortcuts on `SRV1`, which looks like this:

```
PS C:\Foo> # 7. Discovering URL shortcuts
PS C:\Foo> $URLSC = Get-Shortcut -FilePath *.url
PS C:\Foo> $URLSC

    Directory: C:\Users\Administrator\Favorites

Mode          LastWriteTime   Length Name
----          -------------   ------ ----
-a---   05/10/2020    13:37      208 Bing.url
```

Figure 4.48: Discovering URL shortcuts on SRV1

In *step 8*, you examine the contents of the `.URL` file, which looks like this:

```
PS C:\Foo> # 8. Viewing content of shortcut
PS C:\Foo> $URLSC | Get-Content
[{000214A0-0000-0000-C000-000000000046}]
Prop3=19,2
[InternetShortcut]
IDList=
URL=http://go.microsoft.com/fwlink/p/?LinkId=255142
IconIndex=0
IconFile=%ProgramFiles%\Internet Explorer\Images\bing.ico
```

Figure 4.49: Examining the contents of the .URL file

In *step 9*, you create a new shortcut to `Google.com`, which produces no output. In *step 10*, you execute the shortcut. PowerShell then runs the default browser (that is, Internet Explorer) and navigates to the URL in the file, which looks like this:

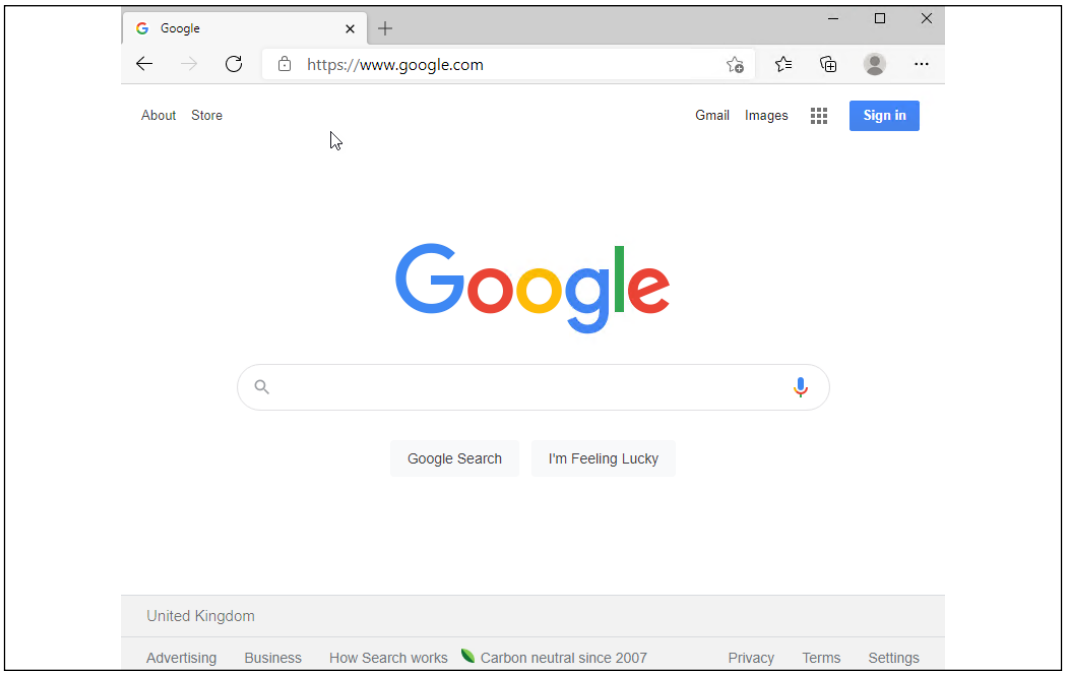


Figure 4.50: Executing the shortcut to Google.com

In *step 11*, you create a link shortcut, which generates no output. In *step 12*, you execute the shortcut, which brings up Notepad, like this:

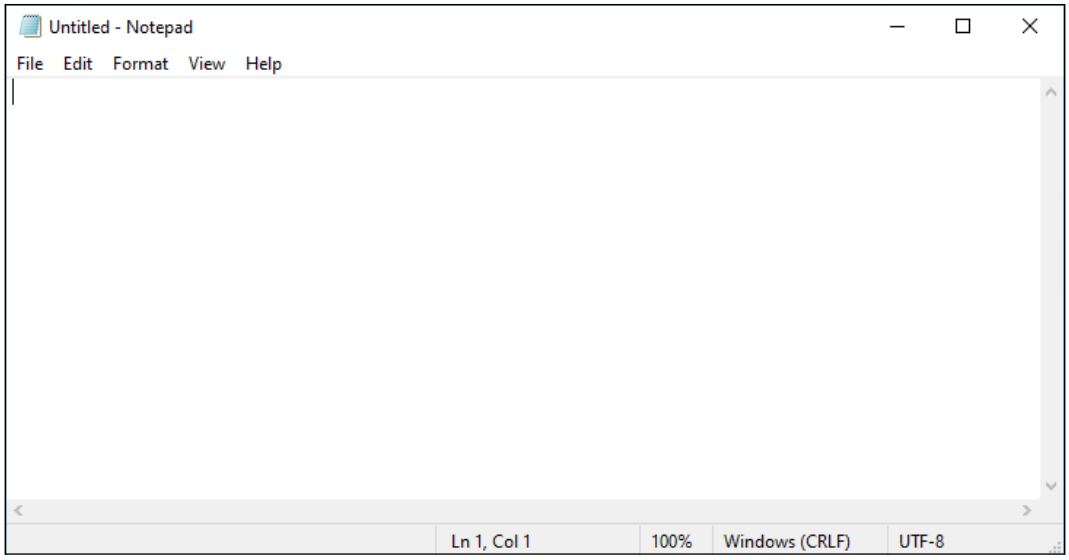


Figure 4.51: Executing the shortcut to Notepad

There's more...

In *step 7*, you find URL shortcuts and as you can see, there is just one: to Bing. The Windows installer created this when it installed Windows Server on this host.

In *step 8*, you examine the contents of a URL shortcut. Unlike link shortcut files, which have a binary format and are not fully readable, URL shortcuts are text files.

Working with archive files

Since the beginning of the PC era, users have employed a variety of file compression mechanisms. An early method used the ZIP file format, initially implemented by PKWare's PKZip program, which quickly became a near-standard for data transfer. Later, a Windows version, WinZip, became popular, and with Windows 98 Microsoft provided built-in support for `.ZIP` archive files. Today, Windows supports ZIP files up to 2 GB in total size. You can file more information about the ZIP file format at `https://en.wikipedia.org/wiki/Zip_(file_format)`.

Numerous developers have, over the years, provided alternative compression schemes and associated utilities, including WinRAR and 7-Zip. WinZip and WinRAR are both excellent programs, but are commercial. 7-Zip is a freeware tool that is also popular. All three offer their own compression mechanisms (with associated file extensions) and support the others as well.

For details on WinZip, see `https://www.winzip.com/win/en`; for information on WinRAR, see `https://www.win-rar.com`; and for more on 7-Zip, see `https://www.7-zip.org`. Each of the compression utilities offered by these groups also supports compression mechanisms from other environments, such as TAR.

In this recipe, you look at PowerShell 7's built-in commands for managing archive files. The commands work only with ZIP files. You can find a PowerShell module for 7-Zip at `https://github.com/thoemmi/7Zip4Powershell`, although the module is old and has not been updated in some time.

Getting ready

You run this recipe on `SRV1`, on which you have installed PowerShell 7 and VS Code. `SRV1` is a Windows Server host running Windows Server Datacenter Edition.

How to do it...

1. Getting the archive module

```
Get-Module -Name Microsoft.Powershell.Archive -ListAvailable
```

2. Discovering commands in the archive module

```
Get-Command -Module Microsoft.PowerShell.Archive
```

3. Making a new folder

```
$NIHT = @{
  Name        = 'Archive'
  Path        = 'C:\Foo'
  ItemType    = 'Directory'
  ErrorAction = 'SilentlyContinue'
}
New-Item @NIHT | Out-Null
```

4. Creating files in the archive folder

```
$Contents = "Have a Nice day with PowerShell and Windows Server" * 1000
1..100 |
  ForEach-Object {
    $FName = "C:\Foo\Archive\Archive_$_.txt"
    New-Item -Path $FName -ItemType File  | Out-Null
    $Contents | Out-File -FilePath $FName
}
```

5. Measuring size of the files to archive

```
$Files = Get-ChildItem -Path 'C:\Foo\Archive'
$Count = $Files.Count
$LenKB = (($Files | Measure-Object -Property length -Sum).Sum)/1mb
"[{0}] files, occupying {1:n2}mb" -f $Count, $LenKB
```

6. Compressing a set of files into an archive

```
$AFILE1 = 'C:\Foo\Archive1.zip'
Compress-Archive -Path $Files -DestinationPath "$AFile1"
```

7. Compressing a folder containing files

```
$AFILE2 = 'C:\Foo\Archive2.zip'
Compress-Archive -Path "C:\Foo\Archive" -DestinationPath $AFile2
```

8. Viewing the archive files

```
Get-ChildItem -Path $AFILE1, $AFILE2
```

9. Viewing archive content with Windows Explorer

```
explorer.exe $AFILE1
```

10. Viewing the second archive with Windows Explorer

```
explorer.exe $AFILE2
```

11. Making a new output folder

```
$Opath = 'C:\Foo\Decompressed'
$NIHT2 = @{
  Path        = $Opath
  ItemType    = 'Directory'
  ErrorAction = 'SilentlyContinue'
}
New-Item @NIHT2 | Out-Null
```

12. Decompressing the `Archive1.zip` archive

```
Expand-Archive -Path $AFILE1 -DestinationPath $Opath
```

13. Measuring the size of the decompressed files

```
$Files = Get-ChildItem -Path $Opath
$Count = $Files.Count
$LenKB = (($Files | Measure-Object -Property length -Sum).Sum)/1mb
"[{0}] decompressed files, occupying {1:n2}mb" -f $Count, $LenKB
```

How it works...

In *step 1*, you use `Get-Module` to examine the `Microsoft.PowerShell.Archive` module, which looks like this:

```
PS C:\Foo> # 1.  Getting archive module
PS C:\Foo> Get-Module -Name Microsoft.Powershell.Archive -ListAvailable

    Directory: C:\program files\powershell\7\Modules

ModuleType Version  PreRelease Name                         PSEdition ExportedCommands
---------- -------  ---------- ----                         --------- ----------------
Manifest   1.2.5               Microsoft.PowerShell.Archive Desk      {Compress-Archive, Expand-Archive}
```

Figure 4.52: Examining the archive module

In *step 2,* you use `Get-Command` to discover the commands in the `Microsoft.PowerShell.Archive` module, which looks like this:

```
PS C:\Foo> # 2. Discovering commands in archive module
PS C:\Foo> Get-Command -Module Microsoft.PowerShell.Archive

CommandType  Name              Version   Source
-----------  ----              -------   ------
Function     Compress-Archive  1.2.5     Microsoft.PowerShell.Archive
Function     Expand-Archive    1.2.5     Microsoft.PowerShell.Archive
```

Figure 4.53: Discovering commands in the archive module

In *step 3*, you create a new folder which, in *step 4,* you populate with 100 text files. These two steps produce no output.

In *step 5*, you use `Get-ChildItem` to get all the files in the archive folder and measure the size of all the files. The output looks like this:

```
PS C:\Foo> # 5. Measuring size of the files to archive
PS C:\Foo> $Files = Get-ChildItem -Path 'C:\Foo\Archive'
PS C:\Foo> $Count = $Files.Count
PS C:\Foo> $LenKB = (($Files | Measure-Object -Property length -Sum).Sum)/1mb
PS C:\Foo> "[{0}] files, occupying {1:n2}mb" -f $Count, $LenKB
[100] files, occupying 4.77mb
```

Figure 4.54: Measuring the size of all archive files

In *step 6*, you compress the set of files you created in *step 5*. This step compresses a set of files into an archive file, which produces no output. In *step 7*, you compress a folder and its contents. This step creates a root folder in the archive file which holds the archived (and compressed) file. This step also produces no output.

In *step 8*, you use `Get-ChildItem` to view the two archive files, which looks like this:

```
PS C:\Foo> # 8. Viewing the archive files
PS C:\Foo> Get-ChildItem -Path $AFILE1, $AFILE2

    Directory: C:\Foo

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a---          04/11/2020    10:43          49306 Archive1.zip
-a---          04/11/2020    10:45          50906 Archive2.zip
```

Figure 4.55: Viewing the archive files

In *step 9*, you use Windows Explorer to view the files in the first archive file, which shows the individual files you compressed into the archive. The output from this step looks like this:

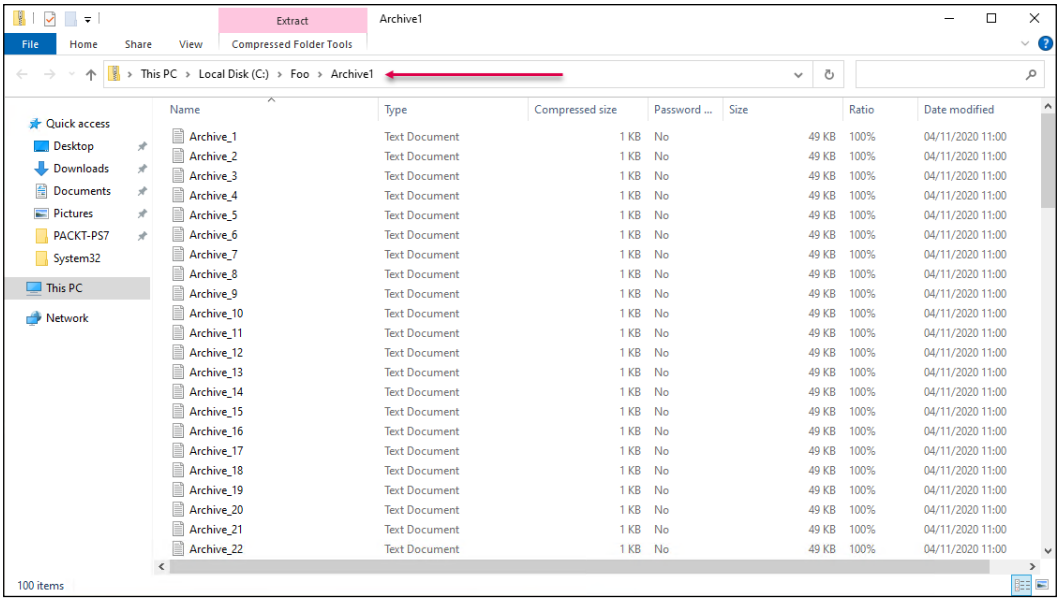


Figure 4.56: Using Windows Explorer to view the first archive

In *step 10*, you use Windows Explorer to view the second archive file, which looks like this:

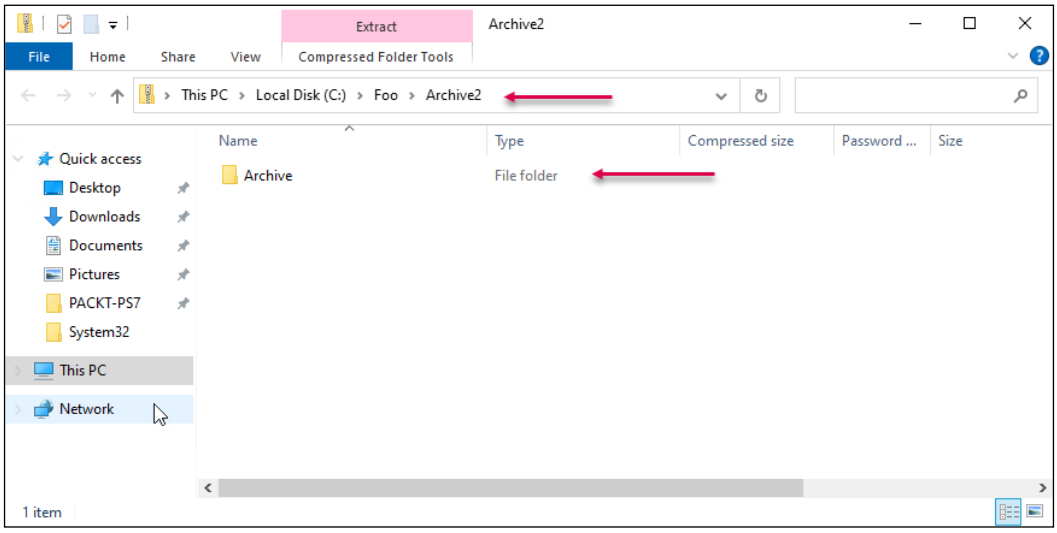


Figure 4.57: Using Windows Explorer to view the second archive

In *step 11*, you create a new folder (`C:\Foo\Decompressed`), producing no output. In *step 12*, you use `Expand-Archive` to decompress the files in `Archive1.ZIP` to the folder created in the previous step, which also produces no output.

In *step 13*, you measure the size of the decompressed files, which looks like this:

```
PS C:\Foo> # 13. Measuring the size of the decompressed files
PS C:\Foo> $Files = Get-ChildItem -Path $Opath
PS C:\Foo> $Count = $Files.Count
PS C:\Foo> $LenKB = (($Files | Measure-Object -Property length -Sum).Sum)/1mb
PS C:\Foo> "[{0}] decompressed files, occupying {1:n2}mb" -f $Count, $LenKB
[100] decompressed files, occupying 4.77mb
```

Figure 4.58: Measuring the size of the decompressed files

There's more...

In *steps 6* and *7*, you compress 100 files which you created earlier in the recipe. The difference between these two steps is that the first step just compresses a set of files, while the second creates an archive with a root folder containing the 100 files. You can see the resulting differences in file sizes in *step 8*, where the second archive is somewhat larger than the first owing to the presence of the root folder.

In *step 12*, you expand the first archive, and in *step 13*, you can see it contains the same number of files and has the same total file size as the 100 files you initially compressed.

5
Exploring .NET

In this chapter, we cover the following recipes:

- Exploring .NET assemblies
- Examining .NET classes
- Leveraging .NET methods
- Creating a C# extension
- Creating a PowerShell cmdlet

Introduction

Microsoft first launched the Microsoft .NET Framework in June 2000, amid a frenzy of marketing zeal, with the code name Next Generation Windows Services. Microsoft seemed to add the .NET moniker to every product. There was Windows .NET Server (renamed Windows Server 2003), Visual Studio .NET, and even MapPoint .NET. As is often the case, over time, .NET provided features which were superseded by later and newer features based on advances in technology. For example, **Simple Object Access Protocol** (**SOAP**) and XML-based web services have given way to **Representation State Transfer** (**REST**) and **JavaScript Object Notation** (**JSON**).

Microsoft made considerable improvements to .NET with each release and added new features in response to customer feedback. .NET started as closed source as the .NET Framework. Microsoft then transitioned .NET to open source, aka .NET Core. PowerShell 7.0 is based on .NET Core 3.1.

An issue was that, over time, .NET became fragmented across different OSes and the web. To resolve this, Microsoft created .NET 5.0 and dropped the "Core" moniker. The intention going forward is to move to a single .NET across all platforms and form factors, thus simplifying application development. See `https://www.zdnet.com/article/microsoft-takes-a-major-step-toward-net-unification-with-net-5-0-release/` for some more details on this.

It is important to note that neither .NET 5.0 nor PowerShell 7.1 have **long-term support** (**LTS**). The next LTS version of .NET is .NET 6.0 and of PowerShell is PowerShell 7.2, both of which are not meant to be released until late in 2021. PowerShell 7.1 is built on top of, and takes advantage of, the latest version of .NET Core, now known as .NET 5.0.

For the application developer, .NET is primarily an **Application Program Interface** (**API**) and a programming model with associated tools and runtime. .NET is an environment in which developers can develop rich applications and websites across multiple platforms.

For IT professionals, .NET provides the underpinnings of PowerShell but is otherwise transparent. You use cmdlets to carry out actions. These cmdlets use .NET to carry out their work without the user being aware there is .NET involved. You can use `Get-Process` without worrying how that cmdlet actually does its work. But that only gets you so far; there are some important .NET features that have no cmdlets. For example, creating an Active Directory cross-forest trust is done by using the appropriate .NET classes and methods (and is very simple too!). .NET is an important depth tool, helping you to go beyond the cmdlets provided in Windows and Windows applications.

Here is a high-level illustration of the components of .NET:

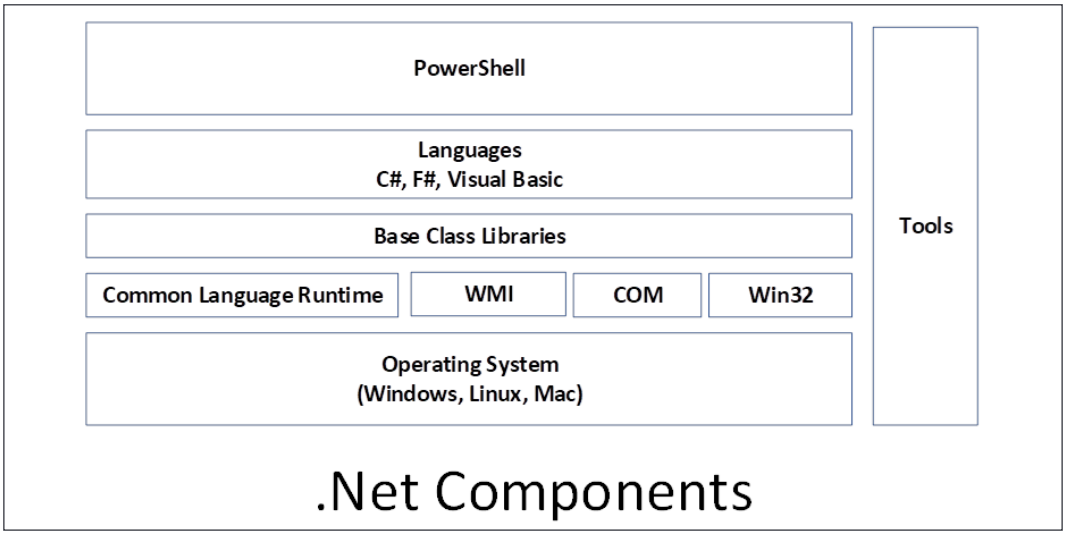


Figure 5.1: .NET components

The key components of .NET in this diagram are:

- **Operating System**: The .NET story begins with the operating system. The operating system provides the fundamental operations of your computer. .NET leverages the OS components. The .NET team supports .NET 5.0, the basis for PowerShell, on Windows, Linux, and Apple Mac. .NET also runs on the ARM platform, which enables you to get PowerShell on ARM. For a full list of operating systems that support PowerShell 7, see `https://github.com/dotnet/core/blob/master/release-notes/5.0/5.0-supported-os.md`.
- **Common Language Runtime** (**CLR**): The CLR is the core of .NET, the managed code environment in which all .NET applications run (including PowerShell). The CLR manages memory, threads, objects, resource allocation/de-allocation, and garbage collection. For an overview of the CLR, see `https://docs.microsoft.com/dotnet/standard/clr/`. For PowerShell users, the CLR "just works."
- **Base Class Libraries** (**BCLs**): .NET comes with a large number of base class libraries, the fundamental built-in features that developers use to build .NET applications. Each BCL is a DLL which contains .NET classes and types. When you install PowerShell, the installer adds the .NET Framework components into PowerShell's installation folder. PowerShell developers use these libraries to implement PowerShell cmdlets. You can also call classes in the BCL directly from within PowerShell. It is the BCLs that enable you to reach into .NET.
- **WMI, COM, Win32**: These are traditional application development technologies which you can access via PowerShell. Within PowerShell, you can run programs that use these technologies, as well as access them from the console or via a script. Your scripts can be a mixture of cmdlets, .NET, WMI, COM, and Win32 if necessary.
- **Languages**: This is the language that a cmdlet and application developer uses to develop PowerShell cmdlets. You can use an extensive variety of languages, based on your preferences and skill set. Most cmdlet developers use C#, although using any .NET supported language would work, such as VB.Net. The original PowerShell development team designed the PowerShell language based on C#. You can describe PowerShell as on the glide scope down to C#. This is useful since there is a lot of documentation with examples in C#.
- **PowerShell**: PowerShell sits on top of these other components. From PowerShell, you can use cmdlets developed using a supported language, and you can use both BCL and WMI/COM/Win32 as well.

For the most part, moving to .NET 5 with respect to PowerShell is more or less transparent, although not totally. One thing that's changed in .NET 5 is it drops support for a lot of WinRT APIs. The .NET team did this to boost cross-platform support. However, this means that any module that relied on these APIs, such as the Appx module (`https://github.com/PowerShell/PowerShell/issues/13138`), is no longer compatible in PowerShell 7.1 (when they were in PS7), although they can be used via the Windows PowerShell compatibility solution described in *Chapter 3*, *Exploring Compatibility with Windows PowerShell*.

In this chapter, you examine the assemblies that make up PowerShell 7.1. You then look at both the classes provided by .NET and how to leverage the methods provided by .NET. You also look at creating a simple C# extension and creating a PowerShell cmdlet.

Exploring .NET assemblies

With .NET, an assembly holds compiled code which .NET can run. An assembly can either be a **Dynamic Link Library** (**DLL**) or an executable. Cmdlets and .NET classes are contained in DLLs, as you see in this recipe. Each assembly also contains a manifest which describes what is in the assembly, along with compiled code.

Most PowerShell modules and commands make use of assemblies of compiled code. When PowerShell loads any module, the module manifest (the `.PSD1` file) lists the assemblies which make up the module. For example, the `Microsoft.PowerShell.Management` module provides many core PowerShell commands, such as `Get-ChildItem` and `Get-Process`. This module's manifest lists a nested module (that is, `Microsoft.PowerShell.Commands.Management.dll`) as the assembly containing the actual commands.

A great feature of PowerShell is the ability for you to invoke a .NET class method directly or to obtain a static .NET class value. The syntax for calling a .NET method or a .NET field, demonstrated in numerous recipes in this book, is to enclose the class name in square brackets, and then follow it with two colon characters (`::`), followed by the name of the method or static field.

In this recipe, you examine the assemblies loaded into PowerShell 7 and compare that with the behavior in Windows PowerShell. The recipe illustrates some of the differences between how PowerShell 7 and Windows PowerShell co-exist with .NET. You also look at a module and the assembly that implements the commands in the module.

Getting ready

You run this recipe on `SRV1`, a workgroup server on which you have installed PowerShell 7 and VS Code.

How to do it...

1. Counting loaded assemblies

```
$Assemblies = [System.AppDomain]::CurrentDomain.GetAssemblies()
"Assemblies loaded: {0:n0}" -f $Assemblies.Count
```

2. Viewing the first 10

```
$Assemblies | Select-Object -First 10
```

3. Checking assemblies in Windows PowerShell

```
$SB = {
  [System.AppDomain]::CurrentDomain.GetAssemblies()
}
$PS51 = New-PSSession -UseWindowsPowerShell
$AssIn51 = Invoke-Command -Session $PS51 -ScriptBlock $SB
"Assemblies loaded in Windows PowerShell: {0:n0}" -f $Assin51.Count
```

4. Viewing `Microsoft.PowerShell` assemblies

```
$AssIn51 |
  Where-Object FullName -Match "Microsoft\.Powershell" |
    Sort-Object -Property Location
```

5. Exploring the `Microsoft.PowerShell.Management` module

```
$Mod = Get-Module -Name Microsoft.PowerShell.Management -ListAvailable
$Mod  | Format-List
```

6. Viewing the module manifest

```
$Manifest = Get-Content -Path $Mod.Path
$Manifest | Select-Object -First 20
```

7. Discovering the module's assembly

```
Import-Module -Name Microsoft.PowerShell.Management
$Match = $Manifest | Select-String Modules
$Lube = $Match.Line
$DLL = ($Line -Split '"')[1]
Get-Item -Path $PSHOME\$DLL
```

8. Viewing associated loaded assembly

```
$Assemblies2 = [System.AppDomain]::CurrentDomain.GetAssemblies()
$Assemblies2 | Where-Object Location -match $DLL
```

9. Getting details of a PowerShell command inside a module DLL

```
$Commands = $Assemblies2
  Where-Object Location -match Commands.Management\.dll
$Commands.GetTypes() |
  Where-Object Name -match "Addcontentcommand$"
```

How it works...

In *step 1*, you use the `GetAssemblies()` method to return all the assemblies currently loaded by PowerShell. Then you output a count of the assemblies currently loaded, which looks like this:

```
PS C:\Foo> # 1. Counting loaded assemblies
PS C:\Foo> $Assemblies = [System.AppDomain]::CurrentDomain.GetAssemblies()
PS C:\Foo> "Assemblies loaded: {0:n0}" -f $Assemblies.Count
Assemblies loaded: 138
```

Figure 5.2: Counting the loaded assemblies

In *step 2*, you look at the first 10 assemblies returned, which looks like this:

```
PS C:\Foo> # 2. Viewing first 10
PS C:\Foo> $Assemblies | Select-Object -First 10

GAC    Version        Location
---    -------        --------
False  v4.0.30319     C:\Program Files\PowerShell\7\System.Private.CoreLib.dll
False  v4.0.30319     C:\Program Files\PowerShell\7\pwsh.dll
False  v4.0.30319     C:\Program Files\PowerShell\7\System.Runtime.dll
False  v4.0.30319     C:\Program Files\PowerShell\7\Microsoft.PowerShell.ConsoleHost.dll
False  v4.0.30319     C:\Program Files\PowerShell\7\System.Management.Automation.dll
False  v4.0.30319     C:\Program Files\PowerShell\7\System.Threading.Thread.dll
False  v4.0.30319     C:\Program Files\PowerShell\7\System.Runtime.InteropServices.dll
False  v4.0.30319     C:\Program Files\PowerShell\7\System.Threading.dll
False  v4.0.30319     C:\Program Files\PowerShell\7\Microsoft.Win32.Primitives.dll
False  v4.0.30319     C:\Program Files\PowerShell\7\System.Diagnostics.Process.dll
```

Figure 5.3: Viewing the first 10 assemblies

In *step 3*, you examine the assemblies loaded into Windows PowerShell 5.1, which looks like this:

```
PS C:\Foo> # 3. Checking assemblies in Windows PowerShell
PS C:\Foo> $SB = {
             [System.AppDomain]::CurrentDomain.GetAssemblies()
           }
PS C:\Foo> $PS51 = New-PSSession -UseWindowsPowerShell
PS C:\Foo> $Assin51 = Invoke-Command -Session $PS51 -ScriptBlock $SB
PS C:\Foo> "Assemblies loaded in Windows PowerShell: {0:n0}" -f $Assin51.Count
Assemblies loaded in Windows PowerShell: 16
```

Figure 5.4: Checking the assemblies in Windows PowerShell 5.1

With *step 4*, you examine the `Microsoft.PowerShell.*` assemblies in PowerShell 5.1, which looks like this:

```
PS C:\Foo> # 4. Viewing Microsoft.PowerShell assemblies
PS C:\Foo> $Assin51 |
             Where-Object FullName -Match "Microsoft\.Powershell" |
               Sort-Object -Property Location

GAC    Version        Location                                                                PSComputerName
---    -------        --------                                                                --------------
True   v4.0.30319     C:\Windows\Microsoft.Net\assembly\GAC_MSIL\Microsoft.PowerShell.Consol… localhost
True   v4.0.30319     C:\Windows\Microsoft.Net\assembly\GAC_MSIL\Microsoft.PowerShell.Securi… localhost
```

Figure 5.5: Viewing the Microsoft.PowerShell assemblies in Windows PowerShell

In *step 5*, you examine the details of the `Microsoft.PowerShell.Management` module, which contains numerous core commands in PowerShell. The output of this step looks like this:

```
PS C:\Foo> # 5. Exploring the Microsoft.PowerShell.Management module
PS C:\Foo> $Mod =
             Get-Module -Name Microsoft.PowerShell.Management -ListAvailable
PS C:\Foo> $Mod  | Format-List


Name              : Microsoft.PowerShell.Management
Path              : C:\program files\powershell\7\Modules\Microsoft.PowerShell.Manag
                    ement\Microsoft.PowerShell.Management.psd1
Description       :
ModuleType        : Manifest
Version           : 7.0.0.0
PreRelease        :
NestedModules     : {Microsoft.PowerShell.Commands.Management}
ExportedFunctions :
ExportedCmdlets   : {Add-Content, Clear-Content, Get-Clipboard, Set-Clipboard,
                    Clear-ItemProperty, Join-Path, Convert-Path, Copy-ItemProperty,
                    Get-ChildItem, Get-Content, Get-ItemProperty, Get-ItemPropertyValue,
                    Move-ItemProperty, Get-Location, Set-Location, Push-Location,
                    Pop-Location, New-PSDrive, Remove-PSDrive, Get-PSDrive, Get-Item,
                    New-Item, Set-Item, Remove-Item, Move-Item, Rename-Item, Copy-Item,
                    Clear-Item, Invoke-Item, Get-PSProvider, New-ItemProperty, Split-Path,
                    Test-Path, Test-Connection, Get-Process, Stop-Process, Wait-Process,
                    Debug-Process, Start-Process, Remove-ItemProperty, Rename-ItemProperty,
                    Resolve-Path, Get-Service, Stop-Service, Start-Service,
                    Suspend-Service, Resume-Service, Restart-Service, Set-Service,
                    New-Service, Remove-Service, Set-Content, Set-ItemProperty,
                    Restart-Computer, Stop-Computer, Rename-Computer, Get-ComputerInfo,
                    Get-TimeZone, Set-TimeZone, Get-HotFix, Clear-RecycleBin}
ExportedVariables :
ExportedAliases   : {gcb, gin, gtz, scb, stz}
```

Figure 5.6: Exploring the Microsoft.PowerShell.Management module in PowerShell 7.1

In *step 6*, you view the manifest for the `Microsoft.PowerShell.Management` module. The following screenshot shows the first 20 lines of the manifest:

```
PS C:\Foo> # 6. Viewing module manifest
PS C:\Foo> $MANIFEST = Get-Content -Path $MOD.Path
PS C:\Foo> $MANIFEST | Select-Object -First 20
@{
GUID="EEFCB906-B326-4E99-9F54-8B4BB6EF3C6D"
Author="PowerShell"
CompanyName="Microsoft Corporation"
Copyright="Copyright (c) Microsoft Corporation."
ModuleVersion="7.0.0.0"
CompatiblePSEditions = @("Core")
PowerShellVersion="3.0"
NestedModules="Microsoft.PowerShell.Commands.Management.dll"
HelpInfoURI = 'https://aka.ms/powershell71-help'
FunctionsToExport = @()
AliasesToExport = @("gcb", "gin", "gtz", "scb", "stz")
CmdletsToExport=@("Add-Content",
    "Clear-Content",
    "Get-Clipboard",
    "Set-Clipboard",
    "Clear-ItemProperty",
    "Join-Path",
    "Convert-Path",
    "Copy-ItemProperty",
```

Figure 5.7: Viewing the Microsoft.PowerShell.Management module manifest

In *step 7*, you extract the name of the DLL implementing the `Microsoft.PowerShell.Management` module and examine the location on disk, which looks like this:

```
PS C:\Foo> # 7. Discovering the module's assembly
PS C:\Foo> Import-Module -Name Microsoft.PowerShell.Management
PS C:\Foo> $Match = $Manifest | Select-String Modules
PS C:\Foo> $Line = $Match.Line
PS C:\Foo> $DLL = ($Line -Split '"')[1]
PS C:\Foo> Get-Item -Path $PSHOME\$DLL

    Directory: C:\Program Files\PowerShell\7

Mode          LastWriteTime    Length Name
----          -------------    ------ ----
-a---   06/11/2020    02:33   1119624 Microsoft.PowerShell.Commands.Management.dll
```

Figure 5.8: Discovering the Microsoft.PowerShell.Management module's assembly

In *step 8*, you find the assembly that contains the cmdlets in the `Microsoft.PowerShell.Management` module, which looks like this:

```
PS C:\Foo> # 8. Viewing associated loaded assembly
PS C:\Foo> $Assemblies2 = [appdomain]::CurrentDomain.GetAssemblies()
PS C:\Foo> $Assemblies2 | Where-Object Location -match $DLL

GAC   Version     Location
---   -------     --------
False v4.0.30319  C:\Program Files\PowerShell\7-preview\Microsoft.PowerShell.Commands.Management.dll
```

Figure 5.9: Viewing the associated loaded assembly

In *step 9*, you discover the name of the class that implements the `Add-Content` command, which looks like this:

```
PS C:\Foo> # 9. Getting details of a command
PS C:\Foo> $Commands  = $Assemblies2
               Where-Object Location -match Commands.Management\.dll
PS C:\Foo> $Commands.GetTypes() |
             Where-Object Name -match "Addcontentcommand$"

IsPublic IsSerial Name                BaseType
-------- -------- ----                --------
True     False    AddContentCommand   Microsoft.PowerShell.Commands.WriteContentCommandBase
```

Figure 5.10: Getting details of a command

There's more...

In this recipe, you have seen the .NET assemblies used by PowerShell. These consist of both the .NET Framework assemblies (that is, the BCLs) and the assemblies which implement PowerShell commands. For example, you find the `Add-Content` cmdlet in the `Microsoft.PowerShell.Management` module.

In *step 1*, you use the `GetAssemblies()` method to return all the assemblies currently loaded by PowerShell 7.1. As you can see, the syntax is different from calling PowerShell cmdlets.

In *steps 3* and *4*, you obtain and view the assemblies loaded by Windows PowerShell 5.1. As you can see, different assemblies are loaded by Windows PowerShell.

In *step 6*, you view the first 20 lines of the module manifest for the `Microsoft.PowerShell.Management` module. The output cuts off the complete list of cmdlets exported by the module and the long digital signature for the module manifest. In *Figure 5.7*, you can see that the command `Add-Content` is implemented within `Microsoft.PowerShell.Management.dll`.

In *step 7*, you discover the DLL, which implements inter alia the `Add-Content` cmdlet and in *step 8*, you can see that the assembly is loaded. In *step 9*, you discover that the `Add-Content` command is implemented by the `AddContentCommand` class within the assembly's DLL. For the curious, navigate to `https://github.com/PowerShell/PowerShell/blob/master/src/Microsoft.PowerShell.Commands.Management/commands/management/AddContentCommand.cs`, where you can read the source code for this cmdlet.

Examining .NET classes

With .NET, a class defines an object. Objects and object occurrences are fundamental to PowerShell, where cmdlets produce and consume objects. For example, the `Get-Process` command returns objects of the class `System.Diagnostics.Process`. If you use `Get-ChildItem` to return files and folders, the output is a set of objects based on the `System.IO.FileInfo` and `System.IO.DirectoryInfo` classes.

In most cases, your console activities and scripts make use of the objects created automatically by PowerShell commands. But you can also use the `New-Object` command to create occurrences of any class as necessary. This book shows numerous examples of creating an object using `New-Object`.

Within .NET, you have two kinds of object definitions: .NET **classes** and .NET **types**. A type defines a simple object that lives, at runtime, on your CPU's stack. Classes, being more complex, live in the global heap. The global heap is a large area of memory which .NET uses to hold object occurrences during a PowerShell session. In almost all cases, the difference between type and class is not overly relevant to IT professionals using PowerShell.

After a script or even a part of a script has run, .NET can tidy up the global heap in a process known as **garbage collection** (**GC**). The garbage collection process is also not important for IT professionals in most cases. The scripts you see in this book, for example, are not generally impacted by the garbage collection process, nor are most production scripts. For more information on the garbage collection process in .NET, see `https://docs.microsoft.com/dotnet/standard/garbage-collection/`.

There are cases where the GC process can impact on performance. For example, the `System.Array` class creates objects of fixed length. If you add an item to an array, .NET creates a new copy of the array (including the addition) and removes the old one. If you are adding a few items to the array, the performance hit is negligible. But if you are adding millions of occurrences, the performance can suffer significantly. To avoid this, you can just use the `ArrayList` class, which supports adding/removing items from an array without the performance penalty. For more information on GC and performance, see `https://docs.microsoft.com/dotnet/standard/garbage-collection/performance`.

.NET 5.0 features many improvements to the garbage collection process. You can read more about the GC improvements in .NET 5.0 here: `https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-5/`.

In .NET, occurrences of every class or type can include members, including properties, methods, and events. A **property** is an attribute of an occurrence of a class. An occurrence of the `System.IO.FileInfo` object, for example, has a `FullName` property. A **method** is effectively a function you can call which can do something to an object occurrence. You look at .NET methods in more details in the *Leveraging .NET methods* recipe. An **event** is something that can happen to an object occurrence, such when an event is generated when a Windows process has completed. .NET events are not covered in this book, although using WMI events is described in *Chapter 15*, *Managing with Windows Management Instrumentation*, in the *Managing WMI events* recipe.

You can quickly determine an object's class (or type) by piping the output of any cmdlet, or an object, to the `Get-Member` cmdlet. The `Get-Member` cmdlet uses a feature of .NET, **reflection**, to look inside and give you a definitive statement of what that object contains. This feature is invaluable – instead of guessing where, in some piece of string output, your script can find the full name of a file, you can discover the `FullName` property, a string, or the `Length` property, which is unambiguously an integer. Reflection and the `Get-Member` cmdlet help you to discover the properties and other members of a given object.

.NET classes can have **static properties** and **static methods**. Static properties/methods are aspects of the class of a whole as opposed to a specific class instance. A static property is a fixed constant value, such as the maximum and minimum values for a 32-bit signed integer or the value of pi. A static method is one that is independent of any specific instance. For example, the `Parse()` method of the INT32 class can parse a string to ensure it is a value 32-bit signed integer. In most cases, you use static methods to create object instances or to do something related to the class.

In this recipe, you look at some everyday objects created automatically by PowerShell. You also examine the static fields of the `[Int]` .NET class.

Getting ready

You run this recipe on `SRV1`, a workgroup host running Windows Server Datacenter edition. This host has PowerShell and VS Code installed.

How to do it...

1. Creating a `FileInfo` object

```
$FILE = Get-ChildItem -Path $PSHOME\pwsh.exe
$FILE
```

2. Discovering the underlying class

```
$TYPE = $FILE.GetType().FullName
".NET Class name: $TYPE"
```

3. Getting member types of the `FileInfo` object

```
$File |
  Get-Member |
    Group-Object -Property MemberType |
      Sort-Object -Property Count -Descending
```

4. Discovering properties of a Windows service

```
Get-Service |
  Get-Member -MemberType Property
```

5. Discovering the underlying type of an integer

```
$I = 42
$IntType  = $I.GetType()
$TypeName = $IntType.FullName
$BaseType = $IntType.BaseType.Name
".Net Class name      : $TypeName"
".NET Class Base Type : $BaseType"
```

6. Looking at process objects

```
$PWSH = Get-Process -Name pwsh |
  Select-Object -First 1
$PWSH |
  Get-Member |
    Group-Object -Property MemberType |
      Sort-Object -Property Count -Descending
```

7. Looking at static properties of a class

```
$Max = [Int32]::MaxValue
$Min = [Int32]::MinValue
"Minimum value [$Min]"
"Maximum value [$Max]"
```

How it works...

In *step 1*, you use the `Get-ChildItem` cmdlet to return an object representing the PowerShell 7 executable file, with output like this:

```
PS C:\Foo> # 1. Creating a FileInfo object
PS C:\Foo> $FILE = Get-ChildItem -Path $PSHOME\pwsh.exe
PS C:\Foo> $FILE

    Directory: C:\Program Files\PowerShell\7

Mode         LastWriteTime  Length Name
----         -------------  ------ ----
-a---  06/11/2020    02:33  280456 pwsh.exe
```

Figure 5.11: Creating a FileInfo object

You can determine the class name of an object by using the `GetType()` method. This method is present on every object and returns information about the object's type.

In *step 2*, you discover and display a full class name, which looks like this:

```
PS C:\Foo> # 2. Discovering the underlying class
PS C:\Foo> $TYPE = $FILE.GetType().FullName
PS C:\Foo> ".NET Class: $TYPE"
.NET Class: System.IO.FileInfo
```

Figure 5.12: Discovering the underlying class

In *step 3*, you use `Get-Member` to display the different member types contained within a `FileInfo` object, with output like this:

```
PS C:\Foo> # 3. Getting member types of FileInfo object
PS C:\Foo> $File |
             Get-Member |
               Group-Object -Property MemberType |
                 Sort-Object -Property Count -Descending

Count Name             Group
----- ----             -----
   21 Method           {System.IO.StreamWriter AppendText(), System.IO.FileInfo…
   15 Property         {System.IO.FileAttributes Attributes {get;set;}, datetim…
    6 NoteProperty     {string PSChildName=pwsh.exe, PSDriveInfo PSDrive=C, boo…
    4 CodeProperty     {System.String LinkType{get=GetLinkType;}, System.String…
    2 ScriptProperty   {System.Object BaseName {get=if ($this.Extension.Length …
```

Figure 5.13: Getting member types of the FileInfo object

In *step 4*, you examine objects returned from the `Get-Service` cmdlet. The output of this step looks like this:

```
PS C:\Foo> # 4. Discovering properties of a Windows service
PS C:\Foo> Get-Service |
             Get-Member -MemberType Property

   TypeName: System.Service.ServiceController#StartupType

Name                  MemberType Definition
----                  ---------- ----------
BinaryPathName        Property   System.String {get;set;}
CanPauseAndContinue   Property   bool CanPauseAndContinue {get;}
CanShutdown           Property   bool CanShutdown {get;}
CanStop               Property   bool CanStop {get;}
Container             Property   System.ComponentModel.IContainer Container {get;}
DelayedAutoStart      Property   System.Boolean {get;set;}
DependentServices     Property   System.ServiceProcess.ServiceController[] DependentServices {get;}
Description           Property   System.String {get;set;}
DisplayName           Property   string DisplayName {get;set;}
MachineName           Property   string MachineName {get;set;}
ServiceHandle         Property   System.Runtime.InteropServices.SafeHandle ServiceHandle {get;}
ServiceName           Property   string ServiceName {get;set;}
ServicesDependedOn    Property   System.ServiceProcess.ServiceController[] ServicesDependedOn {get;}
ServiceType           Property   System.ServiceProcess.ServiceType ServiceType {get;}
Site                  Property   System.ComponentModel.ISite Site {get;set;}
StartType             Property   System.ServiceProcess.ServiceStartMode StartType {get;}
StartupType           Property   Microsoft.PowerShell.Commands.ServiceStartupType {get;set;}
Status                Property   System.ServiceProcess.ServiceControllerStatus Status {get;}
UserName              Property   System.String {get;set;}
```

Figure 5.14: Discovering properties of a Windows service

In *step 5*, you examine the details of an integer, a `System.Int32` data type. You use the `GetType()` method, which is present on all .NET objects, to return the variable's type information, including the full class name of the class. The output looks like this:

```
PS C:\Foo> # 5. Discovering the underlying type of an integer
PS C:\Foo> $I = 42
PS C:\Foo> $IntType  = $I.GetType()
PS C:\Foo> $TypeName = $IntType.Name
PS C:\Foo> $BaseType = $IntType.BaseType.Name
PS C:\Foo> ".Net Class name      : $TypeName"
PS C:\Foo> ".NET Class Base Type : $BaseType"
.NET Class name      : System.Int32
.NET Class base type : ValueType
```

Figure 5.15: Examining type information of an integer

In *step 6*, you examine the member types of a `System.Diagnostics.Process` object. The `Get-Process` cmdlet returns objects of this class. The output of this step looks like this:

```
PS C:\Foo> # 6. Looking at process objects
PS C:\Foo> $PWSH = Get-Process -Name pwsh |
             Select-Object -First 1
PS C:\Foo> $PWSH |
             Get-Member |
               Group-Object -Property MemberType |
                 Sort-Object -Property Count -Descending

  Count Name             Group
----- ----             -----
   52 Property         {int BasePriority {get;}, System.ComponentModel.IContainer Container {ge…
   19 Method           {void BeginErrorReadLine(), void BeginOutputReadLine(), void CancelError…
    8 ScriptProperty   {System.Object CommandLine {get=…
    7 AliasProperty    {Handles = Handlecount, Name = ProcessName, NPM = NonpagedSystemMemorySi…
    4 Event            {System.EventHandler Disposed(System.Object, System.EventArgs), System.D…
    2 PropertySet      {PSConfiguration {Name, Id, PriorityClass, FileVersion}, PSResources {Na…
    1 CodeProperty     {System.Object Parent{get=GetParentProcess;}}
    1 NoteProperty     {string __NounName=Process}
```

Figure 5.16: Looking at member types of a process object

In *step 7*, you examine two static values of the `System.Int32` class. These two fields hold the largest and smallest values, respectively, that a 32-bit signed integer can hold. The output looks like this:

```
PS C:\Foo> # 7. Looking at static properties within a class
PS C:\Foo> $Max = [Int32]::MaxValue
PS C:\Foo> $Min = [Int32]::MinValue
PS C:\Foo> "Minimum value [$Min]"
PS C:\Foo> "Maximum value [$Max]"
Minimum value [-2147483648]
Maximum value [2147483647]
```

Figure 5.17: Examining static values of the System.Int32 class

There's more...

In *step 1*, you create an object representing `pwsh.exe`. This object's full type name is `System.IO.FileInfo`. In .NET, classes live in namespaces and, in general, namespaces equate to DLLs. In this case, the object is in the `System.IO` namespace, and the class is contained in `System.IO.FileSystem.DLL`. You can discover namespace and DLL details by examining the class's documentation – in this case, `https://docs.microsoft.com/dotnet/api/system.io.fileinfo?view=net-5.0`.

As you use your favourite search engine to learn more about .NET classes that might be useful, note that many sites describe the class without the namespace, simply as the `FileInfo` class, while others spell out the class as `System.IO.FileInfo`. If you are going to be using .NET class names in your scripts, a best practice is to spell out the full class name.

Leveraging .NET methods

With .NET, a method is some action that a .NET object occurrence, or the class, can perform. These methods form the basis for many PowerShell cmdlets. For example, you can stop a Windows process by using the `Stop-Process` cmdlet. The cmdlet then uses the `Kill()` method of the associated process object. As a general best practice, you should use cmdlets wherever possible. You should only use .NET classes and methods directly where there is no alternative.

.NET methods can be beneficial for performing operations which have no PowerShell cmdlets. And it can be useful too from the command line; for example, when you wish to kill a process. IT professionals are all too familiar with processes that are not responding and need to be killed, something you can do at the GUI using Task Manager. Or with PowerShell, you can use the `Stop-Process` cmdlet, as discussed above. At the command line, where brevity is useful, you can use `Get-Process` to find the process you want to stop and pipe the output to each process's `Kill()` method. PowerShell then calls the object's `Kill()` method. Of course, to help IT professionals, the PowerShell team created the `Kill` alias to the `Stop-Process` cmdlet. In practice, it's four characters to type versus 12 (or eight if you are using tab completion to its best effect). At the command line, piping to the `kill` method (where you don't even have to use the open/close parentheses!) is just faster and risks fewer typos. It is a good shortcut at the command line – but avoid it in production code.

Another great example is encrypting files. Windows supports the NTFS **Encrypting File System** (**EFS**) feature. EFS enables you to encrypt or decrypt files on your computer with the encryption based on X.509 certificates. For details on the EFS and how it works, see `https://docs.microsoft.com/windows/win32/fileio/file-encryption`.

At the time of writing, there are no cmdlets to encrypt or decrypt files. The `System.IO.FileInfo` class, however, has two methods you can use: `Encrypt()` and `Decrypt()`. These methods encrypt and decrypt a file based on EFS certificates. You can use these .NET methods to encrypt or decrypt a file without having to use the GUI.

As you saw in the *Examining .NET classes* recipe, you can pipe any object to the `Get-Member` cmdlet to discover the methods for the object. Discovering the specific property names and property value types is simple and easy – no guessing or prayer-based text parsing, so beloved by Linux admins.

Getting ready

You run this recipe on SRV1 after loading PowerShell 7.1 and VS Code.

How to do it...

1. Starting Notepad

```
notepad.exe
```

2. Obtaining methods on the Notepad process

```
$Notepad = Get-Process -Name Notepad
$Notepad | Get-Member -MemberType method
```

3. Using the Kill() method

```
$Notepad |
  ForEach-Object {$_.Kill()}
```

4. Confirming the Notepad process is destroyed

```
Get-Process -Name Notepad
```

5. Creating a new folder and some files

```
$Path = 'C:\Foo\Secure'
New-Item -Path $Path -ItemType directory -ErrorAction SilentlyContinue  |
  Out-Null
1..3 | ForEach-Object {
  "Secure File" | Out-File "$Path\SecureFile$_.txt"
}
```

6. Viewing files in the $Path folder

```
$Files = Get-ChildItem -Path $Path
$Files | Format-Table -Property Name, Attributes
```

7. Encrypting the files

```
$Files| ForEach-Object Encrypt
```

8. Viewing file attributes

```
Get-ChildItem -Path $Path |
  Format-Table -Property Name, Attributes
```

9. Decrypting and viewing the files

```
$Files| ForEach-Object {
  $_.Decrypt()
}
Get-ChildItem -Path $Path |
  Format-Table -Property Name, Attributes
```

How it works...

In *step 1*, which produces no output, you start `Notepad.exe`. This creates a process which you can examine and use.

In *step 2*, you obtain Notepad's process object and examine the methods available to you. The output looks like this:

```
PS C:\Foo> # 2. Obtaining methods on the Notepad process
PS C:\Foo> $Notepad = Get-Process -Name Notepad
PS C:\Foo> $Notepad | Get-Member -MemberType Method

   TypeName: System.Diagnostics.Process

Name                      MemberType Definition
----                      ---------- ----------
BeginErrorReadLine        Method     void BeginErrorReadLine()
BeginOutputReadLine       Method     void BeginOutputReadLine()
CancelErrorRead           Method     void CancelErrorRead()
CancelOutputRead          Method     void CancelOutputRead()
Close                     Method     void Close()
CloseMainWindow           Method     bool CloseMainWindow()
Dispose                   Method     void Dispose(), void IDisposable.Dispose()
Equals                    Method     bool Equals(System.Object obj)
GetHashCode               Method     int GetHashCode()
GetLifetimeService        Method     System.Object GetLifetimeService()
GetType                   Method     type GetType()
InitializeLifetimeService Method     System.Object InitializeLifetimeService()
Kill                      Method     void Kill(), void Kill(bool entireProcessTree)
Refresh                   Method     void Refresh()
Start                     Method     bool Start()
ToString                  Method     string ToString()
WaitForExit               Method     void WaitForExit(), bool WaitForExit(int milliseconds)
WaitForExitAsync          Method     System.Threading.Tasks.Task WaitForExitAsync(System.Threading.Cancellati…
WaitForInputIdle          Method     bool WaitForInputIdle(), bool WaitForInputIdle(int milliseconds)
```

Figure 5.18: Examining methods on the Notepad process

In *step 3*, you use the `Kill()` method in the `System.Diagnostics.Process` object to stop the Notepad process. This step produces no output. In *step 4*, you confirm that you have stopped the Notepad process, with output like this:

```
PS C:\Foo> # 4. Confirming Notepad process is destroyed
PS C:\Foo> Get-Process -Name Notepad
Get-Process:
Line |
   2 |  Get-Process -Name Notepad
     |  ~~~~~~~~~~~~~~~~~~~~~~~~~
     | Cannot find a process with the name "Notepad". Verify the process name
       and call the cmdlet again.
```

Figure 5.19: Confirming the Notepad process has been destroyed

To illustrate other uses of .NET methods, you create a folder and three files within the folder in *step 5*. Creating these files generates no output. In *step 6*, you use the `Get-ChildItem` cmdlet to retrieve details about these three files, including all file attributes, which looks like this:

```
PS C:\Foo> # 6. Viewing files in $Path folder
PS C:\Foo> $Files = Get-ChildItem -Path $Path
PS C:\Foo> $Files | Format-Table -Property Name, Attributes

Name           Attributes
----           ----------
SecureFile1.txt    Archive
SecureFile2.txt    Archive
SecureFile3.txt    Archive
```

Figure 5.20: Viewing the files in the $Path folder

In *step 7*, you use the encrypt method to encrypt the files, generating no output at the command line. In *step 8*, you view again the attributes of the files, which looks like this:

```
PS C:\Foo> # 8. Viewing file attributes
PS C:\Foo> Get-ChildItem -Path $Path |
             Format-Table -Property Name, Attributes

Name                   Attributes
----                   ----------
SecureFile1.txt Archive, Encrypted
SecureFile2.txt Archive, Encrypted
SecureFile3.txt Archive, Encrypted
```

Figure 5.21: Viewing the file attributes

Finally, with *step 9*, you decrypt the files using the `Decrypt()` method. You then use the `Get-ChildItem` cmdlet to view the file attributes, which allows you to determine that the files are not encrypted. The output of this step looks like this:

```
PS C:\Foo> # 9. Decrypting and viewing the files
PS C:\Foo> $Files| ForEach-Object {
             $_.Decrypt()
           }
PS C:\Foo> Get-ChildItem -Path $Path |
             Format-Table -Property Name, Attributes

Name           Attributes
----           ----------
SecureFile1.txt    Archive
SecureFile2.txt    Archive
SecureFile3.txt    Archive
```

Figure 5.22: Decrypting and viewing the file attributes again

There's more...

In *step 2*, you view the methods you can invoke on a process object. One of those methods is the `Kill()` method, as you can see in Figure 5.18. In *step 3*, you use that method to stop the Notepad process. The `Kill()` method is an instance method, meaning you invoke it to kill (stop) a specific process. You can read more about this .NET method at `https://docs.microsoft.com/dotnet/api/system.diagnostics.process.kill`.

The output in *step 4* illustrates an error occurring within VS Code. If you use the PowerShell 7 console, you may see slightly different output, although with the same actual error message.

In *steps 7* and *9*, you use the `FileInfo` objects (created by `Get-ChildItem`) and call the `Encrypt()` and `Decrypt()` methods. These steps demonstrate using .NET methods to achieve some objective in the absence of specific cmdlets. Best practice suggests always using cmdlets where you can. You should also note that the syntax in the two steps is different. In *step 7*, you use a more modern syntax which calls the `Encrypt` method for each file. In *step 9*, you use an older syntax that does the same thing, albeit with more characters. Both syntax methods work.

In *step 7*, while you get no console output, Windows may generate a popup (aka Toast) to tell you that you should back up your encryption key. This is a good idea because if you lose the key, you could lose all access to the file.

Creating a C# extension

For most day-to-day operations, the commands provided by PowerShell from Windows features, or third-party modules, give you all the functionality you need. In some cases, as you saw in the *Leveraging .NET methods* recipe, commands do not exist to achieve your goal. In those cases, you can make use of the methods provided by .NET.

There are also cases where you need to perform more complex operations without PowerShell cmdlet or direct .NET support. You may, for example, have a component of an ASP.NET web application, written in C# but which you now wish to repurpose for administrative scripting purposes.

PowerShell makes it easy to add a class, based on .NET language source code, into a PowerShell session. You supply the C# code, and PowerShell creates a .NET class that you can use in the same way you use .NET methods (and using virtually the same syntax). To do this, you use the `Add-Type` cmdlet and specify the C# code for your class/type(s). PowerShell compiles the code and loads the resultant class into your PowerShell session.

An essential aspect of .NET methods is that a method can have multiple definitions or calling sequences. Known as **method overloads**, these multiple definitions allow you to invoke a method using different sets of parameters. This is not dissimilar to PowerShell's use of parameter sets. For example, the `System.String` class, which PowerShell uses to hold strings, contains the `Trim()` method. You use that method to remove extra characters, usually space characters, from the start or end of a string (or both). The `Trim()` method has three different definitions, which you view in this recipe. Each overloaded definition trims characters from a string slightly differently. To view more details on this method, and the three overloaded definitions, see `https://docs.microsoft.com/dotnet/api/system.string.trim?view=net-5.0/`.

In this recipe, you create and use two simple classes, each with static methods.

Getting ready

You run this recipe on `SRV1`, a workgroup system running Windows Server Datacenter edition. You must have loaded PowerShell 7 and VS Code onto this host.

How to do it...

1. Examining the overloaded method definition

```
("a string").Trim
```

2. Creating a C# class definition in a here-string

```
$NewClass = @"
namespace Reskit {
   public class Hello {
     public static void World() {
         System.Console.WriteLine("Hello World!");
     }
   }
  }
"@
```

3. Adding the type into the current PowerShell session

```
Add-Type -TypeDefinition $NewClass
```

4. Examining the method definition

```
[Reskit.Hello]::World
```

5. Using the class's method

```
[Reskit.Hello]::World()
```

6. Extending the code with parameters

```
$NewClass2 = @"
using System;
using System.IO;
namespace Reskit {
  public class Hello2  {
    public static void World() {
      Console.WriteLine("Hello World!");
    }
    public static void World(string name) {
      Console.WriteLine("Hello " + name + "!");
    }
  }
}
"@
```

7. Adding the type into the current PowerShell session

```
Add-Type -TypeDefinition $NewClass2 -Verbose
```

8. Viewing method definitions

```
[Reskit.Hello2]::World
```

9. Calling new method with no parameters specified

```
[Reskit.Hello2]::World()
```

10. Calling new method with a parameter

```
[Reskit.Hello2]::World('Jerry')
```

How it works...

In *step 1*, you examine the different method definitions for the `Trim()` method. The output, showing the different overloaded definitions, looks like this:

```
PS C:\Foo> # 1. Examining overloaded method definition
PS C:\Foo> ("a string").Trim

OverloadDefinitions
-------------------
string Trim()
string Trim(char trimChar)
string Trim(Params char[] trimChars)
```

Figure 5.23: Examining different method definitions for Trim()

In *step 2*, you create a class definition and store the definition in a variable. In *step 3*, you add this class definition into your current PowerShell workspace. These two steps produce no output.

In *step 4*, you use the new method to observe the definitions available to you, which looks like this:

```
PS C:\Foo> # 4. Examining method definition
PS C:\Foo> [Reskit.Hello]::World

OverloadDefinitions
-------------------
static void World()
```

Figure 5.24: Examining method definitions

In *step 5*, you use the `World()` method, whose output is as follows:

```
PS C:\Foo> # 5. Using the class's method
PS C:\Foo> [Reskit.Hello]::World()
Hello World!
```

Figure 5.25: Using the World() method

In *steps 6* and *7*, you create another new class definition, this time with two overloaded definitions to the `World()` method. These two steps produce no output. In *step 8*, you view the definitions for the `World()` method within the `Reskit.Hello2` class, with output like this:

```
PS C:\Foo> # 8. Viewing method definitions
PS C:\Foo> [Reskit.Hello2]::World

OverloadDefinitions
-------------------
static void World()
static void World(string name)
```

Figure 5.26: Viewing the World() method definitions

In *step 9*, you invoke the `World()` method, specifying no parameters, which produces this output:

```
PS C:\Foo> # 9. Calling with no parameters specified
PS C:\Foo> [Reskit.Hello2]::World()
Hello World!
```

Figure 5.27: Calling World() with no parameters specified

In *step 10*, you invoke the `World()` method, specifying a single string parameter, which produces this output:

```
PS C:\Foo> # 10. Calling new method with a parameter
PS C:\Foo> [Reskit.Hello2]::World('Jerry')
Hello Jerry!
```

Figure 5.28: Calling World() while specifying a single string parameter

There's more...

In *step 1*, you examine the different definitions of a method – in this case, the `Trim()` method of the `System.String` class. There are several ways you discover method overloads. In this step, you create a pseudo-object containing a string and then look at the method definitions. PowerShell creates an unnamed object (in the .NET managed heap) which it immediately destroys after the statement completes. At some point later in time, .NET runs the GC process to reclaim the memory used by this temporary object and reorganizes the managed heap.

As you can see, there are three overloaded definitions – three different ways you can invoke the `Trim()` method. You use the first overloaded definition to remove both leading and trailing space characters from a string, probably the most common usage of `Trim()`. With the second definition, you specify a specific character and .NET removes any leading or trailing occurrences of that character. With the third definition, you specify an array of characters to trim from the start or end of the string. In most cases, the last two definitions are less useful, although it depends on your use cases. The extra flexibility is useful.

In *step 3*, you use the `Add-Type` cmdlet to add the class definition into your current workspace. The cmdlet compiles the C# code, creates, and then loads a DLL, which then enables you to use the classes. If you intend to use this add-in regularly, then you could add *steps 2* and *3* into your PowerShell profile file or a production script. Alternatively, if you use this method within a script, you could add the steps into the script.

As you can see from the C# code in *step 6*, you can add multiple definitions to a method. With the classes within .NET and its BCLs, or in your own code, overloaded methods can provide a great deal of value. The .NET method documentation, which you can find at `https://docs.microsoft.com`, is mainly oriented toward developers rather than IT professionals. If you have issues with a .NET method, feel free to fire off a query on one of the many PowerShell support forums, such as the PowerShell group at Spiceworks: `https://community.spiceworks.com/programming/powershell`.

Creating a PowerShell cmdlet

As noted previously, for most operations, the commands and cmdlets available to you natively provide all the functionality you need (in most cases). In the *Creating a C# extension* recipe, you saw how you could create a class definition and add it into PowerShell. In some cases, you may wish to expand on the class definition and create your own cmdlet.

Creating a compiled cmdlet requires you to either use a tool such as Visual Studio or use the free tools provided by Microsoft as part of the .NET Core **Software Development Kit** (**SDK**). Visual Studio, whether the free community edition or the commercial releases, is a rich and complex tool whose inner workings are well outside the scope of this book. The free tools in the SDK are more than adequate to help you create a cmdlet using C#.

As in the *Creating a C# extension* recipe, an important question to ask is: why and when should you create a cmdlet? Aside from the perennial "because you can" excuse, there are reasons why an extension or a cmdlet might be a good idea. You can create a cmdlet to improve performance – for some operations, a compiled cmdlet is just faster than doing operations using a PowerShell script. For some applications, using a cmdlet is to perform actions that are difficult or not possible directly in PowerShell, including asynchronous operations and the use of a **Language Independent Query** (**LINQ**). A developer could write the cmdlet in C#, allowing you to use it easily with PowerShell.

In order to create a cmdlet, you need to install the Windows SDK. The SDK contains all the tools you need to create a cmdlet. The SDK is a free download you get via the internet.

In this recipe, you load the Windows SDK and use it to create a new PowerShell cmdlet. The installation of the .NET SDK requires you to navigate to a web page, download, and then run the SDK installer.

Getting ready

You run this recipe on `SRV1`, a workgroup system running Windows Server Datacenter edition. You must have loaded PowerShell 7 and VS Code onto this host.

How to do it...

1. Installing the .NET 5.0 SDK

   Open your browser, navigate to `https://dotnet.microsoft.com/download/`, and follow the GUI to download and run the SDK installer. You can see this process in the *How it works...* section below.

2. Creating the `cmdlet` folder

```
New-Item -Path C:\Foo\Cmdlet -ItemType Directory -Force
Set-Location C:\Foo\Cmdlet
```

3. Creating a new class library project

```
dotnet new classlib --name SendGreeting
```

4. Viewing contents of the new folder

```
Set-Location -Path .\SendGreeting
Get-ChildItem
```

5. Creating and displaying the `global.json` file

```
dotnet new globaljson
Get-Content .\global.json
```

6. Adding the PowerShell package

```
dotnet add package PowerShellStandard.Library
```

7. Creating the cmdlet source file

```
$Cmdlet = @"
using System.Management.Automation;  // Windows PowerShell assembly.
namespace Reskit
{
  // Declare the class as a cmdlet
  // Specify verb and noun = Send-Greeting
  [Cmdlet(VerbsCommunications.Send, "Greeting")]
  public class SendGreetingCommand : PSCmdlet
  {
    // Declare the parameters for the cmdlet.
    [Parameter(Mandatory=true)]
    public string Name
    {
      get { return name; }
      set { name = value; }
    }
    private string name;
    // Override the ProcessRecord method to process the
    // supplied name and write a greeting to the user by
    // calling the WriteObject method.
    protected override void ProcessRecord()
    {
      WriteObject("Hello " + name + " - have a nice day!");
    }
  }
}
```

```
"@
$Cmdlet | Out-File .\SendGreetingCommand.cs
```

8. Removing the unused class file

```
Remove-Item -Path .\Class1.cs
```

9. Building the cmdlet

```
dotnet build
```

10. Importing the DLL holding the cmdlet

```
$DLLPath = '.\bin\Debug\net5.0\SendGreeting.dll'
Import-Module -Name $DLLPath
```

11. Examining the module's details

```
Get-Module SendGreeting
```

12. Using the cmdlet:

```
Send-Greeting -Name "Jerry Garcia"
```

How it works...

In *step 1*, you open your browser and navigate to the .NET Download page, which looks like this:

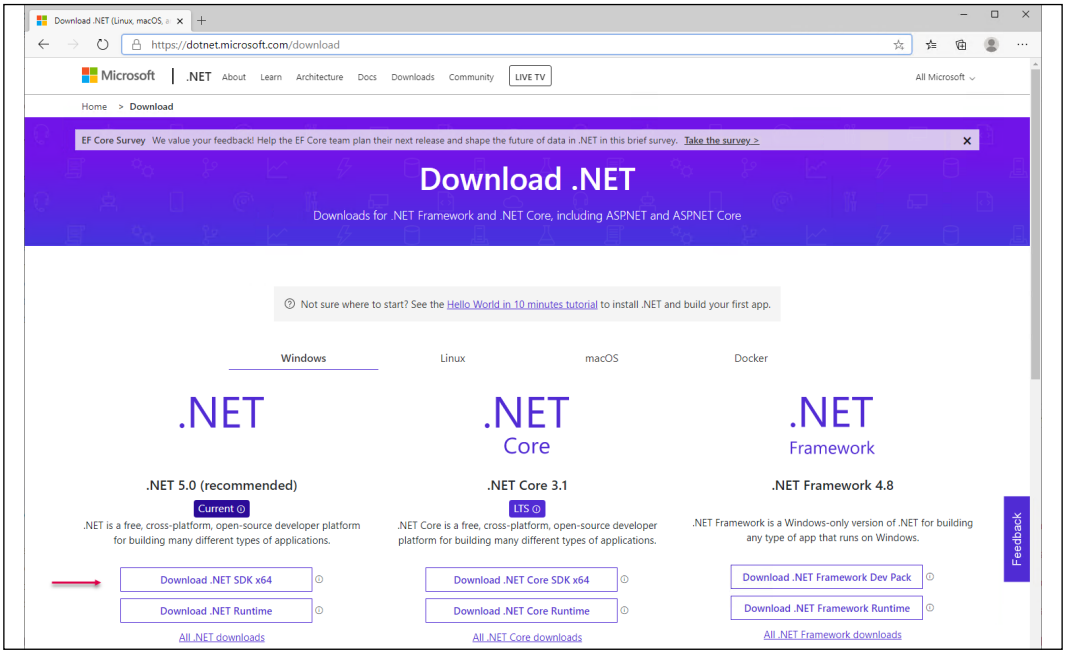


Figure 5.29: The .NET Download page

Click, as shown, on the link to **Download the .NET SDK X64** to download the .NET SDK installer program. Ensure you download the .NET 5.0 SDK, which looks like this:

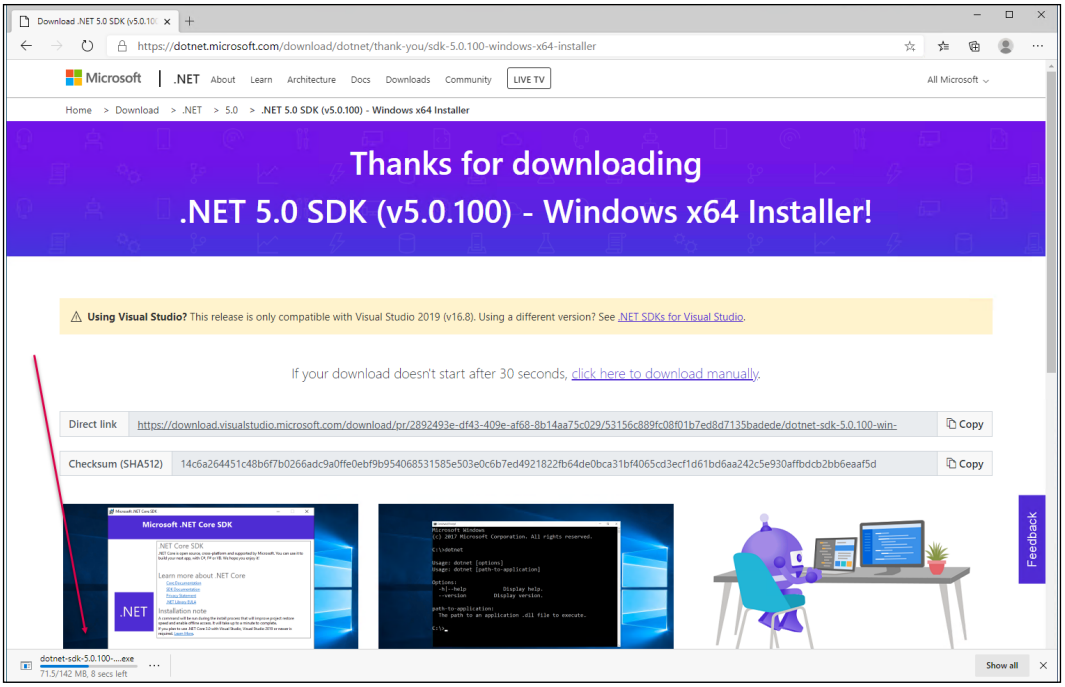


Figure 5.30: Downloading the .NET SDK installer

The installer program is around 150 MB, so it may take a while to download, depending on your internet connection speed. When the download is complete, you see the download indication look like this:

Figure 5.31: Installer download complete

If you click on the **Open file** link, you launch the installer and see the installer's opening page:

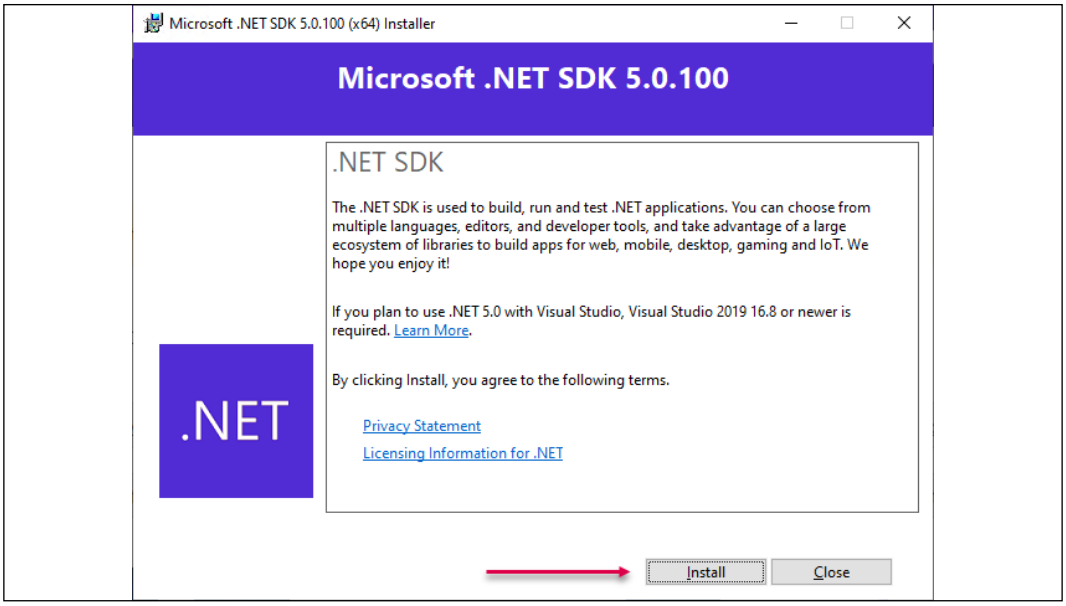


Figure 5.32: The .NET SDK installer

When you click on the **Install** button, as shown, the installer program completes the setup of the .NET SDK. When setup is complete, you see the final screen:

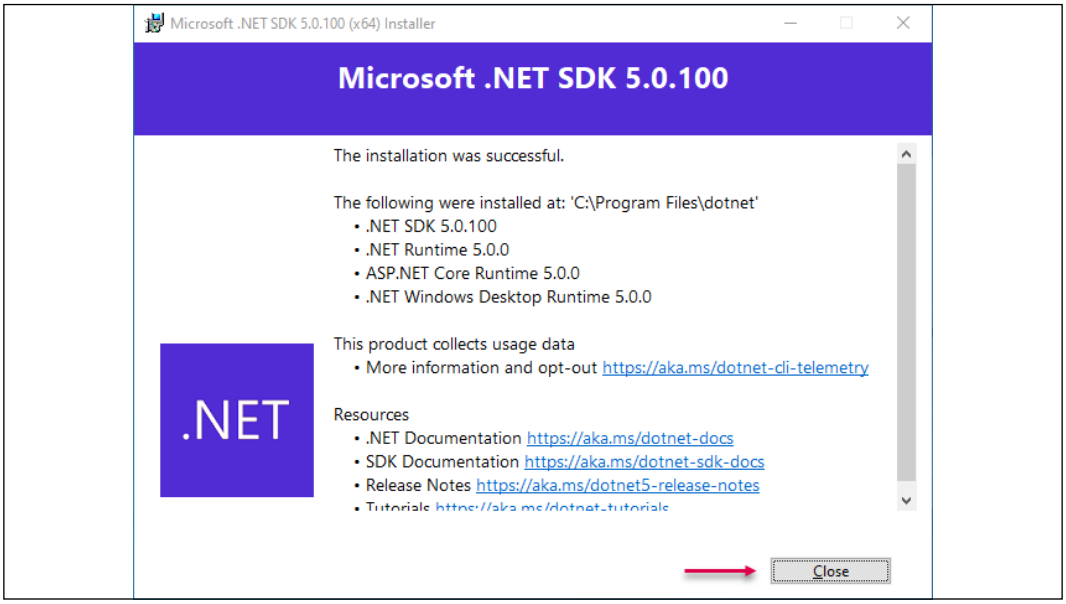


Figure 5.33: Installation successful screen

Click on **Close** to complete the setup. If you had a PowerShell console (or VS Code) open when you did the SDK installation, close then reopen it. The installer updates your system's `Path` variable so you can find the tools, which requires a restart of PowerShell to take effect.

In *step 2*, you create a new folder to hold your new cmdlet and use `Set-Location` to move into that folder. The output from this step looks like this:

```
PS C:\Foo> # 2. Creating the cmdlet folder
PS C:\Foo> New-Item -Path C:\Foo\Cmdlet -ItemType Directory -Force

    Directory: C:\Foo

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         01/12/2020    20:34                Cmdlet

PS C:\Foo> Set-Location C:\Foo\Cmdlet
PS C:\Foo\Cmdlet>
```

Figure 5.34: Creating the cmdlet folder and moving into it

In *step 3*, you create a new class library project using the `dotnet.exe` command, which looks like this:

```
PS C:\Foo\Cmdlet> # 3. Creating a new class library project
PS C:\Foo\Cmdlet> dotnet new classlib --name SendGreeting
The template "Class library" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on SendGreeting\SendGreeting.csproj...
  Determining projects to restore...
  Restored C:\Foo\Cmdlet\SendGreeting\SendGreeting.csproj (in 92 ms).
Restore succeeded.
```

Figure 5.35: Creating a new class library project

In *step 4*, you view the contents of the folder created by the previous step. The output looks like this:

```
PS C:\Foo\Cmdlet> # 4. Viewing contents of new folder
PS C:\Foo\Cmdlet> Set-Location -Path .\SendGreeting
PS C:\Foo\Cmdlet\SendGreeting> Get-ChildItem

    Directory: C:\Foo\Cmdlet\SendGreeting

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         01/12/2020    20:40                obj
-a----         01/12/2020    20:40             89 Class1.cs
-a----         01/12/2020    20:40            137 SendGreeting.csproj
```

Figure 5.36: Viewing the contents of the new folder

In *step 5*, you create and then view a new `global.json` file. This JSON file, among other things, specifies which version of .NET is to be used to build your new cmdlet. The output of this step looks like this:

```
PS C:\Foo\Cmdlet\SendGreeting> # 5. Creating and displaying global.json
PS C:\Foo\Cmdlet\SendGreeting> dotnet new globaljson
The template "global.json file" was created successfully.
PS C:\Foo\Cmdlet\SendGreeting> Get-Content -Path .\global.json

{
  "sdk": {
    "version": "5.0.100"
  }
}
```

Figure 5.37: Creating and displaying a new global.json file

In *step 6*, you add the PowerShell package to your project. This step enables you to use the classes in the package to build your cmdlet. The output looks like this:

```
PS C:\Foo\Cmdlet\SendGreeting> # 6. Adding PowerShell package
PS C:\Foo\Cmdlet\SendGreeting> dotnet add package PowerShellStandard.Library
  Determining projects to restore...
  Writing C:\Users\Administrator\AppData\Local\Temp\1\tmpDF50.tmp
info : Adding PackageReference for package 'PowerShellStandard.Library' into project 'C:\Foo\Cmdlet\SendGreeting\SendGreeting.csproj'.
info :   GET https://api.nuget.org/v3/registration5-gz-semver2/powershellstandard.library/index.json
info :   OK https://api.nuget.org/v3/registration5-gz-semver2/powershellstandard.library/index.json 413ms
info : Restoring packages for C:\Foo\Cmdlet\SendGreeting\SendGreeting.csproj...
info :   GET https://api.nuget.org/v3-flatcontainer/powershellstandard.library/index.json
info :   OK https://api.nuget.org/v3-flatcontainer/powershellstandard.library/index.json 400ms
info :   GET https://api.nuget.org/v3-flatcontainer/powershellstandard.library/5.1.0/powershellstandard.library.5.1.0.nupkg
info :   OK https://api.nuget.org/v3-flatcontainer/powershellstandard.library/5.1.0/powershellstandard.library.5.1.0.nupkg 14ms
info : Installing PowerShellStandard.Library 5.1.0.
info : Package 'PowerShellStandard.Library' is compatible with all the specified frameworks in project 'C:\Foo\Cmdlet\SendGreeting\SendGr
eeting.csproj'.
info : PackageReference for package 'PowerShellStandard.Library' version '5.1.0' added to file 'C:\Foo\Cmdlet\SendGreeting\SendGreeting.c
sproj'.
info : Committing restore...
info : Writing assets file to disk. Path: C:\Foo\Cmdlet\SendGreeting\obj\project.assets.json
log  : Restored C:\Foo\Cmdlet\SendGreeting\SendGreeting.csproj (in 1.04 sec).
```

Figure 5.38: Adding the PowerShell package

In *step 7*, you use PowerShell and a here-string to create the source code file for your new cmdlet. This step produces no output.

In *step 3*, you create your new project. This step also creates a file called `Class1.cs`, which is not needed for this project. In *step 8*, which generates no output, you remove this unneeded file.

In *step 9*, you build the cmdlet, which produces output like this:

```
PS C:\Foo\Cmdlet\SendGreeting> # 9. Building the cmdlet
PS C:\Foo\Cmdlet\SendGreeting> dotnet build
Microsoft (R) Build Engine version 16.8.0+126527ff1 for .NET
Copyright (C) Microsoft Corporation. All rights reserved.

  Determining projects to restore...
  All projects are up-to-date for restore.
  SendGreeting -> C:\Foo\Cmdlet\SendGreeting\bin\Debug\net5.0\SendGreeting.dll

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:03.55
```

Figure 5.39: Building the cmdlet

In *step 10*, you use `Import-Module` to import the DLL, which produces no output. With *step 11*, you use `Get-Module` to discover the imported module (containing just a single command), which looks like this:

```
PS C:\Foo\Cmdlet\SendGreeting> # 11. Examining the module's details
PS C:\Foo\Cmdlet\SendGreeting> Get-Module SendGreeting

ModuleType Version    PreRelease Name           ExportedCommands
---------- -------    ---------- ----           ----------------
Binary     1.0.0.0               SendGreeting   Send-Greeting
```

Figure 5.40: Examining the imported module

In the final step in the recipe, *step 12*, you execute the cmdlet, which produces the following output:

```
PS C:\Foo\Cmdlet\SendGreeting> # 12. Using the cmdlet
PS C:\Foo\Cmdlet\SendGreeting> Send-Greeting -Name "Jerry Garcia"
Hello Jerry Garcia - have a nice day!
```

Figure 5.41: Using the cmdlet

There's more...

In *step 1*, you download the .NET SDK, which you need in order to create a cmdlet. There appears to be no obvious way to automate the installation, so you need to run the installer and click through its GUI in order to install the .NET SDK.

The .NET SDK should be relatively straightforward to install, as shown in this recipe. You download and run the .NET installer and are then able to use the tools to build a cmdlet. But in a few, thankfully rare, cases the installation may not work correctly and/or the installer may get confused. You may need to uninstall any other SDKs you have loaded or perhaps inquire on a support forum such as `https://community.spiceworks.com/programming/powershell`. That being said, you should not have this problem on a newly created VM.

In *step 3*, you create a new class library project. This step also creates the `SendGreeting.csproj` file, as you can see. This file is a Visual Studio .NET C# project file that contains details about the files included in your `Send-Greeting` cmdlet project, assemblies referenced from your code, and more. For more details on the project file, see `https://docs.microsoft.com/aspnet/web-forms/overview/deployment/web-deployment-in-the-enterprise/understanding-the-project-file`.

In *step 5*, you create the `global.json` file. If you do not create this file, the `dotnet` command compiles your cmdlet with the latest version of .NET Core loaded on your system. Using this file tells the project build process to use a specific version of .NET Core (such as version 5.0). For an overview to this file, see `https://docs.microsoft.com/dotnet/core/tools/global-json`.

In *step 9*, you compile your source code file and create a DLL containing your cmdlet. This step compiles all of the source code files contained in the folder to create the DLL. This means you could have multiple cmdlets, each in a separate source code file, and build the entire set in one operation.

In *step 11*, you can see that the module, `SendGreeting`, is a binary module. A binary module is one just loaded directly from a DLL. This is fine for testing, but in production, you should create a manifest module by adding a manifest file. You can get more details on manifest files from `https://docs.microsoft.com/powershell/scripting/developer/module/how-to-write-a-powershell-module-manifest?view=powershell-7.1`. You should also move the module and manifest, and any other module content such as help files, to a supported module location. You might also consider publishing the completed module to either the PowerShell gallery or to an internal repository.

6

Managing Active Directory

In this chapter, we cover the following recipes:

- Installing an AD forest root domain
- Testing an AD installation
- Installing a replica domain controller
- Installing a child domain
- Creating and managing AD users and groups
- Managing AD computers
- Adding/removing users using CSV files
- Creating group policy objects
- Reporting on AD replication
- Reporting on AD computers
- Reporting on AD users

Introduction

A core component of almost all organizations' IT infrastructure is **Active Directory** (**AD**). AD provides access control, user and system customization, and a wealth of directory and other services. Microsoft first introduced AD with Windows 2000 and has improved and expanded the product with each successive release of Windows Server.

Over the years, Microsoft has made AD more of a brand than a single feature. At the core is **Active Directory Domain Services** (**AD DS**). There are four additional Windows Server features under the AD brand:

- **AD Certificate Services** (**AD-CS**): This allows you to issue X.509 certificates for your organization. For an overview of AD-CS, see `https://docs.microsoft.com/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/hh831740(v=ws.11)`.
- **AD Federation Services** (**AD-FS**): This feature enables you to federate identity with other organizations to facilitate interworking. You can find an overview of AD-FS at `https://docs.microsoft.com/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/hh831502(v=ws.11)`.
- **AD Lightweight Directory Services** (**AD-LDS**): This provides rich directory services for use by applications. You can find an overview of AD-LDS at `https://docs.microsoft.com/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/hh831593(v=ws.11)`.
- **AD Rights Management Services** (**AD-RMS**): RMS enables you to control the rights to document access to limit information leakage. For an overview of RMS, see `https://docs.microsoft.com/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/hh831364(v=ws.11)`.

Note that the overview documents referred to above are older documents based on Windows Server 2012. At the time of writing, the documentation teams have not updated them fully to reflect the latest Windows Server version. The overview and the essential operation of these features remain mostly unchanged.

Active Directory's domain service is complex, and there are a lot of moving parts. With AD, you have a logical structure consisting of forests, domains, domain trees, and **organizational units** (**OUs**). You also have the physical structure, including **domain controllers** (**DCs**) and **global catalogs** (**GCs**). There is also a replication mechanism to replicate objects across your domain. For a deeper look at the architecture of AD, see `https://activereach.net/support/knowledge-base/connectivity-networking/understanding-active-directory-its-architecture/`.

A **forest** is a top-level container that houses domains. A forest is a security boundary, although you can set up cross-forest trusts to enable interworking between multiple forests. AD bases forest (and domain) names on DNS. AD DCs and AD clients use DNS to find the IP address of DCs, which makes DNS a critical part of your AD infrastructure.

A **domain** is a collection of objects, including users, computers, policies, and much more. You create a forest by installing the forest's first domain controller. In AD domains, trees are collections of domains that you group in a hierarchical structure. Most organizations use a single domain (and domain tree) within a single forest. Multiple domains in one or more domain trees are also supported, but the best practice is to avoid them.

A **domain controller** is a Windows Server running AD and holding the objects for a given domain. ALL domains must have at least one DC, although best practice is always to have at least two. You install the AD DS service onto your server, then promote the server to be a DC.

The **global catalog** is a partial replica of objects from every domain in an object to enable searching. Exchange, for example, uses the GC heavily. You can have the GC service on some or all DCs in your forest. Generally, you install the GC facility while you promote a Windows Server to be a DC.

Using AD DS (or AD) and PowerShell, you can deploy your domain controllers throughout your organization. Use the *Installing an AD forest root domain* recipe to install a forest root domain controller and establish an AD forest.

Installing features and services using PowerShell in a domain environment often uses remoting, which in turn requires authentication. From one machine, you use PowerShell remoting to perform operations on other systems. You need the correct credentials for those operations.

In this book, you use a domain, `Reskit.Org`, for most of the recipes. In this chapter, you establish the forest and domain (and a child domain) and create users, groups, and organizational units which you rely on in later chapters.

Once you have a first DC in your Reskit.Org forest, you should add a replica DC to ensure reliable domain operations. In *Installing a replica domain controller,* you add a second DC to your domain. In *Installing a child domain*, you extend the forest and add a child domain to your forest.

Active Directory uses a database of objects that include users, computers, and groups. In the *Creating and managing AD users and groups* recipe, you create, move, and remove user and group objects and create and use organizational units. In *Managing AD computers*, you manage the computers in your AD, including joining workgroup systems to the domain. In the *Adding/removing users using CSV files* recipe, you add users to your AD using a **comma-separated values** (**CSV**) file containing users' details.

Group Policy is another important feature of AD. With group policy, you can define policies for users and computers that Windows applies automatically to the user and/or computer. In the *Creating group policy objects* recipe, you create a simple **group policy object** (**GPO**) and observe applying that policy.

Active Directory can make use of multiple DCs for both load balancing and fault tolerance. Such DCs must be synchronized whenever you make changes to your AD, and AD replication performs that function. In *Reporting on AD replication*, you examine tools to help you manage and troubleshoot replication.

In the recipes *Reporting on AD computers* and *Reporting on AD users*, you examine the AD to find details on computers that have not started up or logged onto the domain. You also look at user accounts for users who are members of special security groups (such as enterprise administrators). These two recipes help you to keep your AD free of stale objects or objects that could represent a security risk.

Systems used in this chapter

The recipes in this chapter, and the remainder of the book, are based on creating an AD domain, `Reskit.Org`, a child domain, `UK.Reskit.Org`, three domain controllers, and two additional servers. Here is a diagram showing the servers in use:

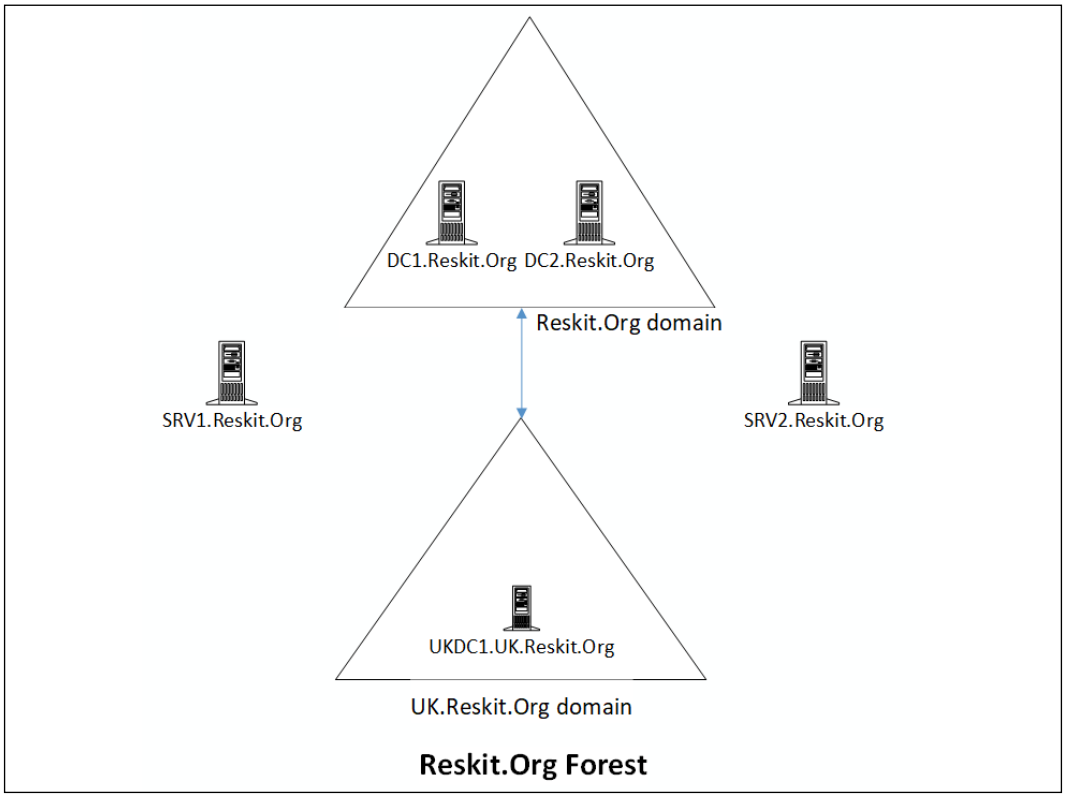


Figure 6.1: Reskit.Org forest diagram

You can use Hyper-V **virtual machines** (**VMs**) to implement this forest. To build these servers, you can get the build scripts from `https://github.com/doctordns/ReskitBuildScripts`. Each recipe in the chapter sets out the specific servers to use. You build new VMs as you need them.

Installing an AD forest root domain

You create an AD forest by creating your first domain controller. Installing Active Directory and DNS has always been reasonably straightforward. You can always use the Server Manager GUI, but using PowerShell is also straightforward.

To create a DC, you start with a system running Windows Server. You then add the AD DS Windows feature to the server. Finally, you create your first DC, for example, a single domain controller, `DC1.Reskit.Org`, for the `Reskit.Org` domain.

Getting ready

You run this recipe on `DC1`, a workgroup server on which you have installed PowerShell 7 and VS Code. You install this server as a workgroup server using the build scripts at `https://github.com/doctordns/ReskitBuildScripts`.

How to do it...

1. Installing the AD DS feature and management tools

```
Install-WindowsFeature -Name AD-Domain-Services -IncludeManagementTools
```

2. Importing the `ADDSDeployment` module

```
Import-Module -Name ADDSDeployment
```

3. Examining the commands in the `ADDSDeployment` module

```
Get-Command -Module ADDSDeployment
```

4. Creating a secure password for Administrator

```
$PSSHT = @{
  String      = 'Pa$$w0rd'
  AsPlainText = $true
  Force       = $true
}
$PSS = ConvertTo-SecureString @PSSHT
```

5. Testing DC forest installation starting on DC1

```
$FOTHT = @{
  DomainName           = 'Reskit.Org'
  InstallDNS           = $true
  NoRebootOnCompletion = $true
  SafeModeAdministratorPassword = $PSS
  ForestMode           = 'WinThreshold'
  DomainMOde           = 'WinThreshold'
}
Test-ADDSForestInstallation @FOTHT -WarningAction SilentlyContinue
```

6. Creating forest root DC on DC1

```
$ADHT = @{
  DomainName                    = 'Reskit.Org'
  SafeModeAdministratorPassword = $PSS
  InstallDNS                    = $true
  DomainMode                    = 'WinThreshold'
  ForestMode                    = 'WinThreshold'
  Force                         = $true
  NoRebootOnCompletion          = $true
  WarningAction                 = 'SilentlyContinue'
}
Install-ADDSForest @ADHT
```

7. Checking key AD and related services

```
Get-Service -Name DNS, Netlogon
```

8. Checking DNS zones

```
Get-DnsServerZone
```

9. Restarting DC1 to complete promotion

```
Restart-Computer -Force
```

How it works...

In *step 1*, you install the AD Domain Services feature. This feature enables you to deploy a server as a domain controller. The output of this command looks like this:

```
PS C:\Foo> # 1. Installing the AD Domain Services feature and management tools
PS C:\Foo> Install-WindowsFeature -Name AD-Domain-Services -IncludeManagementTools

Success Restart Needed Exit Code  Feature Result
------- -------------- ---------  --------------
True    No             Success    {Active Directory Domain Services, Group Pol…
```

Figure 6.2: Installing the AD DS feature

In *step 2*, you manually import the `ADDSDeployment` module. Since PowerShell does not support this module natively, this step loads the module using the Windows PowerShell Compatibility feature. The output of this command looks like this:

```
PS C:\Foo> # 2. Importing the ADDSDeployment module
PS C:\Foo> Import-Module -Name ADDSDeployment
WARNING: Module ADDSDeployment is loaded in Windows PowerShell using WinPSCompatSession
remoting session; please note that all input and output of commands from this module will
be deserialized objects. If you want to load this module into PowerShell please
use 'Import-Module -SkipEditionCheck' syntax.
```

Figure 6.3: Importing the ADDSDeployment module

In *step 3*, you use the `Get-Command` cmdlet to discover the commands contained in the `ADDSDeployment` module, which looks like this:

```
PS C:\Foo> # 3. Examining the commands in the ADDSDeployment module
PS C:\Foo> Get-Command -Module ADDSDeployment

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Function        Add-ADDSReadOnlyDomainControllerAccount            1.0        ADDSDeployment
Function        Install-ADDSDomain                                 1.0        ADDSDeployment
Function        Install-ADDSDomainController                       1.0        ADDSDeployment
Function        Install-ADDSForest                                 1.0        ADDSDeployment
Function        Test-ADDSDomainControllerInstallation              1.0        ADDSDeployment
Function        Test-ADDSDomainControllerUninstallation            1.0        ADDSDeployment
Function        Test-ADDSDomainInstallation                        1.0        ADDSDeployment
Function        Test-ADDSForestInstallation                        1.0        ADDSDeployment
Function        Test-ADDSReadOnlyDomainControllerAccountCreation   1.0        ADDSDeployment
Function        Uninstall-ADDSDomainController                     1.0        ADDSDeployment
```

Figure 6.4: Examining commands in the ADDSDeployment module

With *step 4*, you create a secure string password to use as the Administrator password in the domain you are creating. This step produces no output.

Before you promote a server to be a DC, it's useful to test to ensure that a promotion would be successful as far as possible. In *step 5*, you use the `Test-ADDSForestInstallation` command to check whether you can promote `DC1` as a DC in the `Reskit.Org` domain. The output of this command looks like this:

```
PS C:\Foo> # 5. Testing DC forest installation starting on DC1
PS C:\Foo> $FOTHT = @{
              DomainName           = 'Reskit.Org'
              InstallDNS           = $true
              NoRebootOnCompletion = $true
              SafeModeAdministratorPassword = $PSS
              ForestMode           = 'WinThreshold'
              DomainMOde           = 'WinThreshold'
           }
PS C:\Foo> Test-ADDSForestInstallation @DCTHT -WarningAction SilentlyContinue

RunspaceId     : 0da9f50f-16a7-4de2-8543-0f61b3e4adb7
Message        : Operation completed successfully
Context        : Test.VerifyDcPromoCore.DCPromo.General.3
RebootRequired : False
Status         : Success
```

Figure 6.5: Testing DC forest installation

In *step 6*, you promote `DC1` to be the first domain controller in a new domain, `Reskit.Org`. The output looks like this:

```
PS C:\Foo> # 6. Creating forest root DC on DC1
PS C:\Foo> $ADHT = @{
             DomainName                    = 'Reskit.Org'
             SafeModeAdministratorPassword = $PSS
             InstallDNS                    = $true
             DomainMode                    = 'WinThreshold'
             ForestMode                    = 'WinThreshold'
             Force                         = $true
             NoRebootOnCompletion          = $true
             WarningAction                 = 'SilentlyContinue'
           }
PS C:\Foo> Install-ADDSForest @ADHT

RunspaceId     : 0da9f50f-16a7-4de2-8543-0f61b3e4adb7
Message        : You must restart this computer to complete the operation.

Context        : DCPromo.General.4
RebootRequired : True
Status         : Success
```

Figure 6.6: Creating a forest root DC on DC1

After the promotion is complete, you can check the critical services required for Active Directory. Checking the Netlogon and DNS services, which you do in *step 7*, should look like this:

```
PS C:\Foo> # 7. Checking key AD and related services
PS C:\Foo> Get-Service -Name DNS, Netlogon

Status   Name               DisplayName
------   ----               -----------
Running  DNS                DNS Server
Stopped  Netlogon           Netlogon
```

Figure 6.7: Checking the Netlogon and DNS services

When you promoted `DC1`, you also requested the promotion to install DNS on `DC1`. In *step 8*, you check on the zones created by the DC promotion process, which looks like this:

```
PS C:\Foo> # 8. Checking DNS zones
PS C:\Foo> Get-DnsServerZone

ZoneName            ZoneType   IsAutoCreated   IsDsIntegrated   IsReverseLookupZone   IsSigned
--------            --------   -------------   --------------   -------------------   --------
_msdcs.Reskit.Org   Primary    False           False            False                 False
0.in-addr.arpa      Primary    True            False            True                  False
127.in-addr.arpa    Primary    True            False            True                  False
255.in-addr.arpa    Primary    True            False            True                  False
Reskit.Org          Primary    False           False            False                 False
TrustAnchors        Primary    False           False            False                 False
```

Figure 6.8: Checking DNS zones

You need to reboot DC1 to complete the promotion process, which you do in *step 9*, generating no actual output.

There's more...

The cmdlets that enable you to promote a server to be a DC are not installed on a server system by default. Adding the Active Directory Domain Services Windows feature, in *step 1*, adds the necessary cmdlets to the system.

In *step 6*, you install AD and direct that a DNS server should also be installed – and you check for its presence in *step 7*. In *step 8*, you view the DNS zones created automatically by the promotion process. You specify the DNS domain name, `Reskit.Org`, using the `DomainName` parameter to `Install-ADDSForest`. This step creates the DNS domain, but it is, at this point, still a *non*-AD-integrated zone. Once you reboot the service, this zone should become AD-integrated (and set for secure updates only).

Once you complete the verification of a successful AD installation, you reboot the server. After the restart, there are further tests that you should run, as we show in the next recipe, *Testing an AD installation*.

Testing an AD installation

In *Installing an AD forest root domain*, you installed AD on `DC1`. In that recipe, you installed AD (without rebooting) and tested certain services. After the required reboot (which you completed at the end of the previous recipe), it is useful to check to ensure that your domain is fully up and running and working correctly. In this recipe, you examine core aspects of the AD infrastructure on your first DC.

Getting ready

You run this recipe on `DC1`, the first domain controller in the `Reskit.Org` domain, after you have promoted it to be a DC. You created `DC1` as a domain controller in the Reskit.Org domain in *Installing an AD forest root domain*. Log on as `Reskit\Administrator`.

How to do it...

1. Examining AD root **directory service entry** (**DSE**)

   ```
   Get-ADRootDSE -Server DC1.Reskit.Org
   ```

2. Viewing AD forest details

   ```
   Get-ADForest
   ```

3. Viewing AD domain details

```
Get-ADDomain
```

4. Checking Netlogon, ADWS, and DNS services

```
Get-Service NetLogon, ADWS, DNS
```

5. Getting initial AD users

```
Get-ADUser -Filter * |
  Sort-Object -Property Name |
    Format-Table -Property Name, DistinguishedName
```

6. Getting initial AD groups

```
Get-ADGroup -Filter *  |
  Sort-Object -Property Groupscope, Name |
    Format-Table -Property Name, GroupScope
```

7. Examining Enterprise Admins group membership

```
Get-ADGroupMember -Identity 'Enterprise Admins'
```

8. Checking DNS zones on DC1

```
Get-DnsServerZone -ComputerName DC1
```

9. Testing domain name DNS resolution

```
Resolve-DnsName -Name Reskit.Org
```

How it works...

After you've completed the installation of AD and rebooted, in *step 1*, you examine the AD DSE, which looks like this:

```
PS C:\Foo> # 1. Examining AD root directory service entry (DSE)
PS C:\Foo> Get-ADRootDSE -Server DC1.Reskit.Org

configurationNamingContext    : CN=Configuration,DC=Reskit,DC=Org
currentTime                   : 27/11/2020 20:25:41
defaultNamingContext          : DC=Reskit,DC=Org
dnsHostName                   : DC1.Reskit.Org
domainControllerFunctionality : Windows2016
domainFunctionality           : Windows2016Domain
dsServiceName                 : CN=NTDS Settings,CN=DC1,CN=Servers,CN=Default-First-Site-Name,CN=Sites,
                                CN=Configuration,DC=Reskit,DC=Org
forestFunctionality           : Windows2016Forest
highestCommittedUSN           : 16488
isGlobalCatalogReady          : {TRUE}
isSynchronized                : {TRUE}
ldapServiceName               : Reskit.Org:dc1$@RESKIT.ORG
namingContexts                : {DC=Reskit,DC=Org, CN=Configuration,DC=Reskit,DC=Org,
                                CN=Schema,CN=Configuration,DC=Reskit,DC=Org,
                                DC=DomainDnsZones,DC=Reskit,DC=Org, DC=ForestDnsZones,DC=Reskit,DC=Org}
rootDomainNamingContext       : DC=Reskit,DC=Org
schemaNamingContext           : CN=Schema,CN=Configuration,DC=Reskit,DC=Org
serverName                    : CN=DC1,CN=Servers,CN=Default-First-Site-Name,CN=Sites,CN=Configuration,
                                DC=Reskit,DC=Org
subschemaSubentry             : CN=Aggregate,CN=Schema,CN=Configuration,DC=Reskit,DC=Org
supportedCapabilities         : {1.2.840.113556.1.4.800 (LDAP_CAP_ACTIVE_DIRECTORY_OID),
                                1.2.840.113556.1.4.1670 (LDAP_CAP_ACTIVE_DIRECTORY_V51_OID),
                                1.2.840.113556.1.4.1791 (LDAP_CAP_ACTIVE_DIRECTORY_LDAP_INTEG_OID),
                                1.2.840.113556.1.4.1935 (LDAP_CAP_ACTIVE_DIRECTORY_V61_OID),
                                1.2.840.113556.1.4.2080, 1.2.840.113556.1.4.2237}
supportedControl              : {1.2.840.113556.1.4.319 (LDAP_PAGED_RESULT_OID_STRING),
                                1.2.840.113556.1.4.801 (LDAP_SERVER_SD_FLAGS_OID),
                                1.2.840.113556.1.4.473 (LDAP_SERVER_SORT_OID), 1.2.840.113556.1.4.528
                                (LDAP_SERVER_NOTIFICATION_OID), 1.2.840.113556.1.4.417
                                (LDAP_SERVER_SHOW_DELETED_OID), 1.2.840.113556.1.4.619
                                (LDAP_SERVER_LAZY_COMMIT_OID), 1.2.840.113556.1.4.841
                                (LDAP_SERVER_DIRSYNC_OID), 1.2.840.113556.1.4.529
                                (LDAP_SERVER_EXTENDED_DN_OID), 1.2.840.113556.1.4.805
                                (LDAP_SERVER_TREE_DELETE_OID), 1.2.840.113556.1.4.521
                                (LDAP_SERVER_CROSSDOM_MOVE_TARGET_OID), 1.2.840.113556.1.4.970
                                (LDAP_SERVER_GET_STATS_OID), 1.2.840.113556.1.4.1338
                                (LDAP_SERVER_VERIFY_NAME_OID), 1.2.840.113556.1.4.474
                                (LDAP_SERVER_RESP_SORT_OID), 1.2.840.113556.1.4.1339
                                (LDAP_SERVER_DOMAIN_SCOPE_OID), 1.2.840.113556.1.4.1340
                                (LDAP_SERVER_SEARCH_OPTIONS_OID), 1.2.840.113556.1.4.1413
                                (LDAP_SERVER_PERMISSIVE_MODIFY_OID), 2.16.840.1.113730.3.4.9
                                (LDAP_CONTROL_VLVREQUEST), 2.16.840.1.113730.3.4.10
                                (LDAP_CONTROL_VLVRESPONSE), 1.2.840.113556.1.4.1504
                                (LDAP_SERVER_ASQ_OID), 1.2.840.113556.1.4.1852
                                (LDAP_SERVER_QUOTA_CONTROL_OID), 1.2.840.113556.1.4.802
                                (LDAP_SERVER_RANGE_OPTION_OID), 1.2.840.113556.1.4.1907
                                (LDAP_SERVER_SHUTDOWN_NOTIFY_OID), 1.2.840.113556.1.4.1948
                                (LDAP_SERVER_RANGE_RETRIEVAL_NOERR_OID), 1.2.840.113556.1.4.1974
                                (LDAP_SERVER_FORCE_UPDATE_OID), 1.2.840.113556.1.4.1341
                                (LDAP_SERVER_RODC_DCPROMO_OID), 1.2.840.113556.1.4.2026
                                (LDAP_SERVER_DN_INPUT_OID), 1.2.840.113556.1.4.2064,
                                1.2.840.113556.1.4.2065, 1.2.840.113556.1.4.2066,
                                1.2.840.113556.1.4.2090, 1.2.840.113556.1.4.2205,
                                1.2.840.113556.1.4.2204, 1.2.840.113556.1.4.2206,
                                1.2.840.113556.1.4.2211, 1.2.840.113556.1.4.2239,
                                1.2.840.113556.1.4.2255, 1.2.840.113556.1.4.2256,
                                1.2.840.113556.1.4.2309, 1.2.840.113556.1.4.2330,
                                1.2.840.113556.1.4.2354}
supportedLDAPPolicies         : {MaxPoolThreads, MaxPercentDirSyncRequests, MaxDatagramRecv,
                                MaxReceiveBuffer, InitRecvTimeout, MaxConnections, MaxConnIdleTime,
                                MaxPageSize, MaxBatchReturnMessages, MaxQueryDuration,
                                MaxDirSyncDuration, MaxTempTableSize, MaxResultSetSize, MinResultSets,
                                MaxResultSetsPerConn, MaxNotificationPerConn, MaxValRange,
                                MaxValRangeTransitive, ThreadMemoryLimit, SystemMemoryLimitPercent}
supportedLDAPVersion          : {3, 2}
supportedSASLMechanisms       : {GSSAPI, GSS-SPNEGO, EXTERNAL, DIGEST-MD5}
```

Figure 6.9: Examining the AD DSE

In *step 2*, you use the `Get-ADForest` command to review further information on the newly created forest, which looks like this:

```
PS C:\Foo> # 2. Viewing AD forest details
PS C:\Foo> Get-ADForest

ApplicationPartitions : {DC=ForestDnsZones,DC=Reskit,DC=Org, DC=DomainDnsZones,DC=Reskit,DC=Org}
CrossForestReferences : {}
DomainNamingMaster    : DC1.Reskit.Org
Domains               : {Reskit.Org}
ForestMode            : Windows2016Forest
GlobalCatalogs        : {DC1.Reskit.Org}
Name                  : Reskit.Org
PartitionsContainer   : CN=Partitions,CN=Configuration,DC=Reskit,DC=Org
RootDomain            : Reskit.Org
SchemaMaster          : DC1.Reskit.Org
Sites                 : {Default-First-Site-Name}
SPNSuffixes           : {}
UPNSuffixes           : {}
```

Figure 6.10: Viewing the AD forest details

In *step 3*, you use `Get-ADDomain` to return more information about the `Reskit.Org` domain, which looks like this:

```
PS C:\Foo> # 3. Viewing AD Domain details
PS C:\Foo> Get-ADDomain

AllowedDNSSuffixes                : {}
ChildDomains                      : {}
ComputersContainer                : CN=Computers,DC=Reskit,DC=Org
DeletedObjectsContainer           : CN=Deleted Objects,DC=Reskit,DC=Org
DistinguishedName                 : DC=Reskit,DC=Org
DNSRoot                           : Reskit.Org
DomainControllersContainer        : OU=Domain Controllers,DC=Reskit,DC=Org
DomainMode                        : Windows2016Domain
DomainSID                         : S-1-5-21-1051839064-3594903887-4180521017
ForeignSecurityPrincipalsContainer : CN=ForeignSecurityPrincipals,DC=Reskit,DC=Org
Forest                            : Reskit.Org
InfrastructureMaster              : DC1.Reskit.Org
LastLogonReplicationInterval      :
LinkedGroupPolicyObjects          : {CN={31B2F340-016D-11D2-945F-00C04FB984F9},CN=Policies,CN=System,DC=Reskit
                                    ,DC=Org}
LostAndFoundContainer             : CN=LostAndFound,DC=Reskit,DC=Org
ManagedBy                         :
Name                              : Reskit
NetBIOSName                       : RESKIT
ObjectClass                       : domainDNS
ObjectGUID                        : a12396d8-3e8a-431e-8d9b-1b2de80209a5
ParentDomain                      :
PDCEmulator                       : DC1.Reskit.Org
PublicKeyRequiredPasswordRolling  : True
QuotasContainer                   : CN=NTDS Quotas,DC=Reskit,DC=Org
ReadOnlyReplicaDirectoryServers   : {}
ReplicaDirectoryServers           : {DC1.Reskit.Org}
RIDMaster                         : DC1.Reskit.Org
SubordinateReferences             : {DC=ForestDnsZones,DC=Reskit,DC=Org, DC=DomainDnsZones,DC=Reskit,DC=Org,
                                    CN=Configuration,DC=Reskit,DC=Org}
SystemsContainer                  : CN=System,DC=Reskit,DC=Org
UsersContainer                    : CN=Users,DC=Reskit,DC=Org
```

Figure 6.11: Viewing AD domain details

The AD services run within the Netlogon Windows service. The ADWS service is a web service used by the PowerShell AD cmdlets to communicate with the AD. AD relies on DNS as a locator service, enabling AD clients and DCs to find DCs. All three must be up and running for you to manage AD using PowerShell. In *step 4*, you check all three services, which looks like this:

```
PS C:\Foo> # 4. Checking Netlogon. ADWS, and DNS services
PS C:\Foo> Get-Service NetLogon, ADWS, DNS

Status   Name               DisplayName
------   ----               -----------
Running  ADWS               Active Directory Web Services
Running  DNS                DNS Server
Running  Netlogon           NetLogon
```

Figure 6.12: Checking the Netlogon, ADWS, and DNS services

In *Installing an AD forest root domain*, you promoted `DC1`, a Windows server, to be a DC. In doing so, the promotion process creates three domain users: Administrator, Guest, and the krbtgt (Kerberos Ticket-Granting Ticket) user. You should never touch the krbtgt user and should usually leave the Guest user disabled. For added security, you can rename the Administrator user to something less easy to guess and create a new, very low-privilege user with the name Administrator.

In *step 5*, you examine the users in your newly created AD forest, which looks like this:

```
PS C:\Foo> # 5. Getting initial AD users
PS C:\Foo> Get-ADUser -Filter * |
             Sort-Object -Property Name |
               Format-Table -Property Name, DistinguishedName

Name          DistinguishedName
----          -----------------
Administrator CN=Administrator,CN=Users,DC=Reskit,DC=Org
Guest         CN=Guest,CN=Users,DC=Reskit,DC=Org
krbtgt        CN=krbtgt,CN=Users,DC=Reskit,DC=Org
```

Figure 6.13: Getting initial AD users in the new AD forest

The DC promotion process also creates several groups that can be useful. In *step 6*, you examine the different groups created (and their scope), which looks like this:

```
PS C:\Foo> # 6. Getting initial AD groups
PS C:\Foo> Get-ADGroup -Filter *  |
             Sort-Object -Property Groupscope,Name |
               Format-Table -Property Name, GroupScope

Name                                        GroupScope
----                                        ----------
Access Control Assistance Operators     DomainLocal
Account Operators                       DomainLocal
Administrators                          DomainLocal
Allowed RODC Password Replication Group DomainLocal
Backup Operators                        DomainLocal
Cert Publishers                         DomainLocal
Certificate Service DCOM Access         DomainLocal
Cryptographic Operators                 DomainLocal
Denied RODC Password Replication Group  DomainLocal
Distributed COM Users                   DomainLocal
DnsAdmins                               DomainLocal
Event Log Readers                       DomainLocal
Guests                                  DomainLocal
Hyper-V Administrators                  DomainLocal
IIS_IUSRS                               DomainLocal
Incoming Forest Trust Builders          DomainLocal
Network Configuration Operators         DomainLocal
Performance Log Users                   DomainLocal
Performance Monitor Users               DomainLocal
Pre-Windows 2000 Compatible Access      DomainLocal
Print Operators                         DomainLocal
RAS and IAS Servers                     DomainLocal
RDS Endpoint Servers                    DomainLocal
RDS Management Servers                  DomainLocal
RDS Remote Access Servers               DomainLocal
Remote Desktop Users                    DomainLocal
Remote Management Users                 DomainLocal
Replicator                              DomainLocal
Server Operators                        DomainLocal
Storage Replica Administrators          DomainLocal
Terminal Server License Servers         DomainLocal
Users                                   DomainLocal
Windows Authorization Access Group      DomainLocal
Cloneable Domain Controllers                 Global
DnsUpdateProxy                               Global
Domain Admins                                Global
Domain Computers                             Global
Domain Controllers                           Global
Domain Guests                                Global
Domain Users                                 Global
Group Policy Creator Owners                  Global
Key Admins                                   Global
Protected Users                              Global
Read-only Domain Controllers                 Global
Enterprise Admins                         Universal
Enterprise Key Admins                     Universal
Enterprise Read-only Domain Controllers   Universal
Schema Admins                             Universal
```

Figure 6.14: Examining initial AD groups and their scope

The Enterprise Admins group is one with very high privilege. Members of this group can perform just about any operation across the domain – stop/start services, modify the AD, and access any file or folder on the domain or any domain-joined system. In *step 7*, you examine the initial members of this high-privilege group, which looks like this:

```
PS C:\Foo> # 7. Examining Enterprise Admins group membership
PS C:\Foo> Get-ADGroupMember -Identity 'Enterprise Admins'

distinguishedName : CN=Administrator,CN=Users,DC=Reskit,DC=Org
name              : Administrator
objectClass       : user
objectGUID        : 5ee8d68c-6210-44b9-8308-3b120e4efaa6
SamAccountName    : Administrator
SID               : S-1-5-21-1051839064-3594903887-4180521017-500
```

Figure 6.15: Examining Enterprise Admins group membership

AD relies on DNS to enable an AD client or AD DC to find DCs. When you install a DC, you can also install the DNS service at the same time. When you install a DC with DNS, the AD promotion process creates several DNS zones in your newly created DNS service. In *step 8*, you examine the DNS zones created on `DC1`, which looks like this:

```
PS C:\Foo> # 8. Checking DNS zones on DC1
PS C:\Foo> Get-DnsServerZone -ComputerName DC1

ZoneName              ZoneType   IsAutoCreated  IsDsIntegrated  IsReverseLookupZone  IsSigned
--------              --------   -------------  --------------  -------------------  --------
_msdcs.Reskit.Org     Primary    False          True            False                False
0.in-addr.arpa        Primary    True           False           True                 False
127.in-addr.arpa      Primary    True           False           True                 False
255.in-addr.arpa      Primary    True           False           True                 False
Reskit.Org            Primary    False          True            False                False
TrustAnchors          Primary    False          False           False                False
```

Figure 6.16: Checking the DNS zones created on DC1

Another good test of your newly promoted DC is to ensure you can resolve the DNS name of your new domain (`Reskit.Org`). In *step 9*, you use the `Resolve-DnsName` to check, which looks like this:

```
PS C:\Foo> # 9. Testing domain name DNS resolution
PS C:\Foo> Resolve-DnsName -Name Reskit.Org

Name         Type   TTL   Section    IPAddress
----         ----   ---   -------    ---------
Reskit.Org   AAAA   600   Answer     2a02:8010:6386:0:55f:51d1:c96a:bafe
Reskit.Org   A      600   Answer     10.10.10.10
```

Figure 6.17: Testing domain name DNS resolution

There's more...

In *step 1*, you viewed the DSE for your domain. The AD implements a **Lightweight Directory Access Protocol** (**LDAP**) directory service for domain activities. The DSE is a component of LDAP directories and contains information about your directory structure. The DSE is available without requiring authentication and contains much information about your AD forest. That information could help an attacker; thus, best practice is never to expose an AD DC to the Internet if you can help it. For a more detailed look at the root DSE, see `https://docs.microsoft.com/windows/win32/adschema/rootdse`.

The ADWS service, which you investigate in *step 4*, implements a web service. The AD commands use this web service to get information from and make changes to your AD. If this service is not running, AD commands do not work. You should always check to ensure the service has started before proceeding to use the AD cmdlets.

In *step 6*, you saw the groups created by the promotion process. These groups have permissions associated and thus are useful. Before adding users to these groups, consider reviewing the group and determining (and possibly changing) the permissions and rights assigned to these groups.

Installing a replica domain controller

In *Installing an AD forest root domain*, you installed AD on `DC1`. If you have just one DC, then that DC is a single point of failure. With a single DC, when `DC1` goes down, you cannot manage AD using the PowerShell cmdlets, and users cannot log on. It is always best practice to install at least two DCs. If you are using VMs for your DCs, you should also ensure that each DC VM is on a separate virtualization host.

To add a second DC to your domain, you run `Install-ADDSDomainController` on another host, for example, `DC2`. This cmdlet is similar to `Install-ADDSForest` in terms of parameters. As with creating your first DC, it is useful to carry out some tests to ensure the second DC's promotion can succeed.

In this recipe, you promote a host, `DC2`, to be the second DC in the `Reskit.Org` domain. Like when creating your first DC, after you promote `DC2` to be a DC, you need to reboot the server before processing. And after the reboot, it is useful to ensure the promotion process was successful.

Getting ready

You run this recipe on `DC2`, a domain-joined server on which you have installed PowerShell 7 and VS Code. You should log into `DC2` as `Reskit\Administrator`, a member of the Enterprise Admins group.

Using the `Reskit.Org` build scripts, you would build this VM after creating `DC1` to be a DC.

How to do it...

1. Importing the `ServerManager` module

```
Import-Module -Name ServerManager -WarningAction SilentlyContinue
```

2. Checking `DC1` can be resolved

```
Resolve-DnsName -Name DC1.Reskit.Org -Type A
```

3. Testing the network connection to `DC1`

```
Test-NetConnection -ComputerName DC1.Reskit.Org -Port 445
Test-NetConnection -ComputerName DC1.Reskit.Org -Port 389
```

4. Adding the AD DS features on `DC2`

```
Install-WindowsFeature -Name AD-Domain-Services -IncludeManagementTools
```

5. Promoting `DC2` to be a DC

```
Import-Module -Name ADDSDeployment -WarningAction SilentlyContinue
$URK    = 'Administrator@Reskit.Org'
$PW     = 'Pa$$w0rd'
$PSS    = ConvertTo-SecureString -String $PW -AsPlainText -Force
$CredRK = [PSCredential]::New($URK,$PSS)
$INSTALLHT = @{
  DomainName                    = 'Reskit.Org'
  SafeModeAdministratorPassword = $PSS
  SiteName                      = 'Default-First-Site-Name'
  NoRebootOnCompletion          = $true
  InstallDNS                    = $false
  Credential                    = $CredRK
  Force                         = $true
  }
Install-ADDSDomainController @INSTALLHT | Out-Null
```

6. Checking the computer objects in AD

```
Get-ADComputer -Filter * |
  Format-Table DNSHostName, DistinguishedName
```

7. Rebooting `DC2` manually

```
Restart-Computer -Force
```

8. Checking DCs in `Reskit.Org`

```
$SB = 'OU=Domain Controllers,DC=Reskit,DC=Org'
Get-ADComputer -Filter * -SearchBase $SB |
  Format-Table -Property DNSHostName, Enabled
```

9. Viewing `Reskit.Org` domain DCs

```
Get-ADDomain |
  Format-Table -Property Forest, Name, Replica*
```

How it works...

In *step 1*, you import the `ServerManager` module, which creates no output. With *step 2*, you ensure that you can resolve your DC's address, which is, at this point, the only DC in the `Reskit.Org` domain. The output looks like this:

```
PS C:\Foo> # 2. Checking DC1 can be resolved
PS C:\Foo> Resolve-DnsName -Name DC1.Reskit.Org -Type A

Name                 Type   TTL   Section    IPAddress
----                 ----   ---   -------    ---------
DC1.Reskit.Org       A      3600  Answer     10.10.10.10
```

Figure 6.18: Checking DC1 can be resolved

After confirming that you can resolve the IP address of DC1, in *step 3*, you check that you can connect to two key ports on `DC1` (`445` and `389`). The output looks like this:

```
PS C:\Foo> # 3. Testing the network connection to DC1
PS C:\Foo> Test-NetConnection -ComputerName DC1.Reskit.Org -Port 445

ComputerName     : DC1.Reskit.Org
RemoteAddress    : 2a02:8010:6386:0:55f:51d1:c96a:bafe
RemotePort       : 445
InterfaceAlias   : Ethernet 2
SourceAddress    : 2a02:8010:6386:0:1a:9c97:279a:d742
TcpTestSucceeded : True


PS C:\Foo> Test-NetConnection -ComputerName DC1.Reskit.Org -Port 389

ComputerName     : DC1.Reskit.Org
RemoteAddress    : 2a02:8010:6386:0:55f:51d1:c96a:bafe
RemotePort       : 389
InterfaceAlias   : Ethernet 2
SourceAddress    : 2a02:8010:6386:0:1a:9c97:279a:d742
TcpTestSucceeded : True
```

Figure 6.19: Testing the network connection to DC1

As you saw, when promoting `DC1` to be your first DC, you need to add the `ADDSDeployment` feature to `DC2` before you can promote the DC. The output from *step 4* looks like this:

```
PS C:\Foo> # 4. Adding the AD DS features on DC2
PS C:\Foo> Install-WindowsFeature -Name AD-Domain-Services -IncludeManagementTools

Success Restart Needed Exit Code      Feature Result
------- -------------- ---------      --------------
True    No             Success        {Active Directory Domain Services, Group Pol…
```

Figure 6.20: Adding the AD DS features on DC2

With all the prerequisites in place, in *step 5*, you promote `DC2` to be a DC and ensure that the DC does not reboot after this step completes. There is no output from this step.

In *step 6*, before rebooting `DC2` (necessary to complete the promotion process), you check to see the computer objects now in AD. This step helps you to ensure that `DC2` is now in the correct place in AD. The output of this step is like this:

```
PS C:\Foo> # 6. Checking the computer objects in AD
PS C:\Foo> Get-ADComputer -Filter * |
             Format-Table DNSHostName, DistinguishedName

DNSHostName     DistinguishedName
-----------     -----------------
DC1.Reskit.Org  CN=DC1,OU=Domain Controllers,DC=Reskit,DC=Org
DC2.Reskit.Org  CN=DC2,OU=Domain Controllers,DC=Reskit,DC=Org
```

Figure 6.21: Checking the computer objects in the AD

In *step 7*, you finalize the promoting process and reboot `DC2`. Once rebooted, you need to log on as the domain administrator (`Reskit\Administrator`). This step produces no output as such.

With *step 8*, you examine the hosts in the Domain Controllers OU, which looks like this:

```
PS C:\Foo> # 8. Checking DCs in Reskit.Org
PS C:\Foo> $SB = 'OU=Domain Controllers,DC=Reskit,DC=Org'
PS C:\Foo> Get-ADComputer -Filter * -SearchBase $SB |
             Format-Table -Property DNSHostName, Enabled

DNSHostName    Enabled
-----------    -------
DC2.Reskit.Org    True
DC1.Reskit.Org    True
```

Figure 6.22: Checking DCs in Reskit.Org

In the final step in this recipe, *step 9*, you use the `Get-ADDomain` cmdlet to check that `DC1` and `DC2` are DCs for this domain. The output of this step looks like this:

```
PS C:\Foo> # 9. Viewing Reskit.Org domain DCs
PS C:\Foo> Get-ADDomain |
             Format-Table -Property Forest, Name, Replica*

Forest      Name   ReplicaDirectoryServers
------      ----   -----------------------
Reskit.Org Reskit {DC1.Reskit.Org, DC2.Reskit.Org}
```

Figure 6.23: Viewing Reskit.Org domain DCs

There's more...

In *step 1*, you import the `ServerManager` module manually. As you have seen in earlier recipes, this module is not supported natively with PowerShell 7. This step loads the module using the Windows PowerShell Compatibility solution described earlier in the book.

In *step 3*, you check to ensure `DC2` can connect to `DC1` over ports `445` and `389`. Windows uses TCP port `445` for SMB file sharing. The group policy agent on each domain-joined host uses this port to retrieve group policy details from the SYSVOL share on `DC1`. LDAP uses port `389` to perform actions against a DC, such as adding a new user or checking group membership. Both ports need to be open and contactable if the promotion of `DC2` is to be successful.

In *step 6*, you retrieve all computer accounts currently in your AD. By default, the AD DC promotion process ensures that this host's computer account is now in the Domain Controllers OU. The location of a DC in an OU is essential as it enables your new DC to apply the group policy settings on this OU. If `DC2` were a workgroup computer and not joined to the domain, the promotion process would create this account in the Domain Controllers OU. If `DC2` were a domain member, then the promotion process would move the computer account into the Domain Controllers OU.

In this recipe, before promoting `DC2` to be a DC, you check to ensure that the prerequisites are in place before the promotion. Then you ensure that you have configured your forest, domain, and DCs correctly. Mistakes are easy to make, but also (usually) easy to correct. In practice, this is redundant checking since most AD promotions "just work." The additional checks ensure you discover and resolve any issues that might affect the promotion or cause the AD to not work correctly after a not-fully-successful promotion.

Installing a child domain

As you saw in *Installing a replica domain controller*, adding a DC to an existing domain is straightforward. So long as prerequisites like DNS are in place, the promotion process is simple.

An AD forest can contain more than one domain. This architecture has some value in terms of delegated control and some reduction in replication traffic. And like creating a replica DC, creating a child domain is simple, as you can see in this recipe.

Best practice calls for a contiguous namespace of domains, where the additional domain is a child of another existing domain. In this recipe, you create a child domain to the `Reskit.Org` domain you created earlier. You promote the server `UKDC1` to be the first DC in a new child domain, `UK.Reskit.Org`. The steps in this recipe are very similar to those in *Installing a replica domain controller*.

Getting ready

You run this recipe on `UKDC1`, a domain-joined server in the `Reskit.Org` domain on which you have installed PowerShell 7 and VS Code. You create this server after you have installed `DC1` as a DC.

How to do it...

1. Importing the `ServerManager` module

```
Import-Module -Name ServerManager -WarningAction SilentlyContinue
```

2. Checking `DC1` can be resolved

```
Resolve-DnsName -Name DC1.Reskit.Org -Type A
```

3. Checking network connection to `DC1`

```
Test-NetConnection -ComputerName DC1.Reskit.Org -Port 445
Test-NetConnection -ComputerName DC1.Reskit.Org -Port 389
```

4. Adding the AD DS features on `UKDC1`

```
$Features = 'AD-Domain-Services'
Install-WindowsFeature -Name $Features -IncludeManagementTools
```

5. Creating a credential and installation hash table

```
Import-Module -Name ADDSDeployment -WarningAction SilentlyContinue
$URK    = 'Administrator@Reskit.Org'
```

```
$PW     = 'Pa$$w0rd'
$PSS    = ConvertTo-SecureString -String $PW -AsPlainText -Force
$CredRK = [PSCredential]::New($URK,$PSS)
$INSTALLHT    = @{
  NewDomainName                 = 'UK'
  ParentDomainName              = 'Reskit.Org'
  DomainType                    = 'ChildDomain'
  SafeModeAdministratorPassword = $PSS
  ReplicationSourceDC           = 'DC1.Reskit.Org'
  Credential                    = $CredRK
  SiteName                      = 'Default-First-Site-Name'
  InstallDNS                    = $false
  Force                         = $true
}
```

6. Installing child domain

```
Install-ADDSDomain @INSTALLHT
```

7. Looking at the AD forest

```
Get-ADForest -Server UKDC1.UK.Reskit.Org
```

8. Looking at the UK domain

```
Get-ADDomain -Server UKDC1.UK.Reskit.Org
```

How it works...

In *step 1*, you load the `ServerManager` module. This module is not natively supported by PowerShell 7. This step, which produces no output, loads the module using the Windows PowerShell Compatibility solution.

When you create a new domain and new DC (using `UKDC1`, for example), the server needs to contact the holder of the domain naming FSMO role, the DC responsible for the forest's changes. In *step 2,* you check to ensure you can reach the host, `DC1`, which looks like this:

```
PS C:\Foo> # 2. Checking DC1 can be resolved
PS C:\Foo> Resolve-DnsName -Name DC1.Reskit.Org -Type A

Name             Type  TTL   Section   IPAddress
----             ----  ---   -------   ---------
DC1.Reskit.Org   A     3600  Answer    10.10.10.10
```

Figure 6.24: Checking DC1 can be resolved

In *step 3*, you check that you can connect to `DC1` on ports `445` and `389`, both required for proper domain functioning. The output of this step looks like this:

```
PS C:\Foo> # 3. Checking network connection to DC1
PS C:\Foo> Test-NetConnection -ComputerName DC1.Reskit.Org -Port 389

ComputerName     : DC1.Reskit.Org
RemoteAddress    : 2a02:8010:6386:0:55f:51d1:c96a:bafe
RemotePort       : 389
InterfaceAlias   : Ethernet 2
SourceAddress    : 2a02:8010:6386:0:709f:f504:1dec:5a48
TcpTestSucceeded : True

PS C:\Foo> Test-NetConnection -ComputerName DC1.Reskit.Org -Port 445

ComputerName     : DC1.Reskit.Org
RemoteAddress    : 2a02:8010:6386:0:55f:51d1:c96a:bafe
RemotePort       : 445
InterfaceAlias   : Ethernet 2
SourceAddress    : 2a02:8010:6386:0:709f:f504:1dec:5a48
TcpTestSucceeded : True
```

Figure 6.25: Checking network connection to DC1

In *step 4*, you add the AD DS feature to `UKDC1`. This feature is required to install the necessary tools you use to create the child domain. The output of this step looks like this:

```
PS C:\Foo> # 4. Adding the AD DS features on UKDC1
PS C:\Foo> $Features = 'AD-Domain-Services'
PS C:\Foo> Install-WindowsFeature -Name $Features -IncludeManagementTools

Success Restart Needed Exit Code   Feature Result
------- -------------- ---------   --------------
True    No             Success     {Active Directory Domain Services, Group Pol…
```

Figure 6.26: Adding the AD DS features on UKDC1

In *step 5*, you build a hash table of parameters that you use in a later step to perform the promotion. In *step 6*, you use this hash table when calling `Install-ADDSDomain` to create a child domain. These two steps create no output. After the promotion has completed, your host reboots; log on as `Reskit\Administrator`.

Once you have logged in, you can check the AD forest details from your newly promoted child domain DC. In *step 7*, you use `Get-ADForest`, with output like this:

```
PS C:\Foo> # 7. Looking at the AD forest
PS C:\Foo> Get-ADForest -Server UKDC1.UK.Reskit.Org

ApplicationPartitions : {DC=DomainDnsZones,DC=Reskit,DC=Org, DC=ForestDnsZones,DC=Reskit,DC=Org}
CrossForestReferences : {}
DomainNamingMaster    : DC1.Reskit.Org
Domains               : {Reskit.Org, UK.Reskit.Org}
ForestMode            : Windows2016Forest
GlobalCatalogs        : {DC1.Reskit.Org, DC2.Reskit.Org, UKDC1.UK.Reskit.Org}
Name                  : Reskit.Org
PartitionsContainer   : CN=Partitions,CN=Configuration,DC=Reskit,DC=Org
RootDomain            : Reskit.Org
SchemaMaster          : DC1.Reskit.Org
Sites                 : {Default-First-Site-Name}
SPNSuffixes           : {}
UPNSuffixes           : {}
```

Figure 6.27: Looking at the AD forest details

In *step 8*, you examine the domain details of the newly created child domain. By using `Get-ADDomain` and targeting the new DC, you see output that looks like this:

```
PS C:\Foo> # 8. Looking at the UK domain
PS C:\Foo> Get-ADDomain -Server UKDC1.UK.Reskit.Org

AllowedDNSSuffixes                : {}
ChildDomains                      : {}
ComputersContainer                : CN=Computers,DC=UK,DC=Reskit,DC=Org
DeletedObjectsContainer           : CN=Deleted Objects,DC=UK,DC=Reskit,DC=Org
DistinguishedName                 : DC=UK,DC=Reskit,DC=Org
DNSRoot                           : UK.Reskit.Org
DomainControllersContainer        : OU=Domain Controllers,DC=UK,DC=Reskit,DC=Org
DomainMode                        : Windows2016Domain
DomainSID                         : S-1-5-21-1221044622-3182058179-1497057381
ForeignSecurityPrincipalsContainer : CN=ForeignSecurityPrincipals,DC=UK,DC=Reskit,DC=Org
Forest                            : Reskit.Org
InfrastructureMaster              : UKDC1.UK.Reskit.Org
LastLogonReplicationInterval      :
LinkedGroupPolicyObjects          : {CN={31B2F340-016D-11D2-945F-00C04FB984F9},CN=Policies,CN=System,DC=UK,DC=Reskit,DC=Org}
LostAndFoundContainer             : CN=LostAndFound,DC=UK,DC=Reskit,DC=Org
ManagedBy                         :
Name                              : UK
NetBIOSName                       : UK
ObjectClass                       : domainDNS
ObjectGUID                        : dda27a2f-73d0-41c5-a624-742f89b6986f
ParentDomain                      : Reskit.Org
PDCEmulator                       : UKDC1.UK.Reskit.Org
PublicKeyRequiredPasswordRolling  : True
QuotasContainer                   : CN=NTDS Quotas,DC=UK,DC=Reskit,DC=Org
ReadOnlyReplicaDirectoryServers   : {}
ReplicaDirectoryServers           : {UKDC1.UK.Reskit.Org}
RIDMaster                         : UKDC1.UK.Reskit.Org
SubordinateReferences             : {}
SystemsContainer                  : CN=System,DC=UK,DC=Reskit,DC=Org
UsersContainer                    : CN=Users,DC=UK,DC=Reskit,DC=Org
```

Figure 6.28: Looking at the UK domain details

There's more...

In *step 2*, you check that `DC1` is the domain naming FSMO role holder. Assuming you have installed the `Reskit.Org` domain and forest as shown in this chapter, that role holder is `DC1`. You could call `Get-ADForest` and obtain the hostname from the `DomainNamingMaster` property.

In *steps 5* and *6*, you promote `UKDC1` to be the first DC in a new child domain, `UK.Reskit.Org`. Unlike in the two previous DC promotions (that is, promoting `DC1` and `DC2`), in this step, you allow Windows to reboot immediately after the promotion has completed. This step can take quite a while – potentially 10 minutes or more – so be patient.

In *step 7*, you use `Get-ADForest` to examine the details of the forest as stored on `UKDC1`. As you can see in *Figure 6.27*, these details now show your new domain (`UK.Reskit.Org`) in the `Domains` property. Also, by default, you can see that `UKDC1.UK.Reskit.Org` is also a global catalog server.

Creating and managing AD users and groups

After creating your forest/domain and your DCs, you can begin to manage the core objects in AD, namely, users, groups, computers, and organizational units. User and computer accounts identify a specific user or computer. Windows uses these objects to enable the computer and the user to log on securely using passwords held in the AD.

AD groups enable you to collect users and computers into a single (group) account that simplifies setting access controls on resources such as files or file shares. As you saw in *Testing an AD installation*, when you create a new forest, the AD promotion process creates many potentially useful groups.

Organizational units enable you to partition users, computers, and groups into separate container OUs. OUs provide you with essential roles in your AD. The first is role delegation. You can delegate the management of any OU (and child OUs) to be carried out by different groups. For example, you could create a top-level OU called `UK` in the `Reskit.Org` domain. You could then delegate permissions to the objects in this OU to a group, such as `UKAdmins`, enabling any group member to manage AD objects in, and below, the UK OU.

The second role played by OUs is to act as a target for group policy objects. You could create a GPO for the IT team and apply it to the IT OU. You could create a separate OU and create GPOs that apply to only the computer and user objects in that OU. Thus, each user and computer in a given OU is configured based on the GPO.

In this recipe, you examine AD user and group objects. In a later recipe, *Reporting on AD computers*, you explore managing AD computers. And in *Creating group policy objects*, you assign a group policy to an OU you create in this recipe.

Getting ready

You run this recipe on DC1, a DC in the Reskit.Org domain. You have a workgroup server on which you have installed PowerShell 7 and VS Code on this host.

How to do it...

1. Creating a hash table for general user attributes

```
$PW  = 'Pa$$w0rd'
$PSS = ConvertTo-SecureString -String $PW -AsPlainText -Force
$NewUserHT = @{}
$NewUserHT.AccountPassword       = $PSS
$NewUserHT.Enabled               = $true
$NewUserHT.PasswordNeverExpires  = $true
$NewUserHT.ChangePasswordAtLogon = $false
```

2. Creating two new users

```
# First user
$NewUserHT.SamAccountName    = 'ThomasL'
$NewUserHT.UserPrincipalName = 'thomasL@reskit.org'
$NewUserHT.Name              = 'ThomasL'
$NewUserHT.DisplayName       = 'Thomas Lee (IT)'
New-ADUser @NewUserHT
# Second user
$NewUserHT.SamAccountName    = 'RLT'
$NewUserHT.UserPrincipalName = 'rlt@reskit.org'
$NewUserHT.Name              = 'Rebecca Tanner'
$NewUserHT.DisplayName       = 'Rebecca Tanner (IT)'
New-ADUser @NewUserHT
```

3. Creating an OU for IT

```
$OUHT = @{
    Name        = 'IT'
    DisplayName = 'Reskit IT Team'
    Path        = 'DC=Reskit,DC=Org'
}
New-ADOrganizationalUnit @OUHT
```

4. Moving users into the OU

```
$MHT1 = @{
    Identity   = 'CN=ThomasL,CN=Users,DC=Reskit,DC=ORG'
    TargetPath = 'OU=IT,DC=Reskit,DC=Org'
}
```

```
Move-ADObject @MHT1
$MHT2 = @{
    Identity = 'CN=Rebecca Tanner, CN=Users, DC=Reskit, DC=ORG'
    TargetPath = 'OU=IT,DC=Reskit,DC=Org'
}
Move-ADObject @MHT2
```

5. Creating a third user directly in the IT OU

```
$NewUserHT.SamAccountName    = 'JerryG'
$NewUserHT.UserPrincipalName = 'jerryg@reskit.org'
$NewUserHT.Description       = 'Virtualization Team'
$NewUserHT.Name              = 'Jerry Garcia'
$NewUserHT.DisplayName       = 'Jerry Garcia (IT)'
$NewUserHT.Path              = 'OU=IT,DC=Reskit,DC=Org'
New-ADUser @NewUserHT
```

6. Adding two users who get removed later

```
# First user to be removed
$NewUserHT.SamAccountName    = 'TBR1'
$NewUserHT.UserPrincipalName = 'tbr@reskit.org'
$NewUserHT.Name              = 'TBR1'
$NewUserHT.DisplayName       = 'User to be removed'
$NewUserHT.Path              = 'OU=IT,DC=Reskit,DC=Org'
New-ADUser @NewUserHT
# Second user to be removed
$NewUserHT.SamAccountName     = 'TBR2'
$NewUserHT.UserPrincipalName  = 'tbr2@reskit.org'
$NewUserHT.Name               = 'TBR2'
New-ADUser @NewUserHT
```

7. Viewing existing AD users

```
Get-ADUser -Filter *  -Property *|
  Format-Table -Property Name, Displayname, SamAccountName
```

8. Removing a user via a `Get | Remove` pattern

```
Get-ADUser -Identity 'CN=TBR1,OU=IT,DC=Reskit,DC=Org' |
    Remove-ADUser -Confirm:$false
```

9. Removing a user directly

```
$RUHT = @{
  Identity = 'CN=TBR2,OU=IT,DC=Reskit,DC=Org'
  Confirm  = $false}
Remove-ADUser @RUHT
```

10. Updating a user object

```
$TLHT =@{
  Identity     = 'ThomasL'
  OfficePhone  = '4416835420'
  Office       = 'Cookham HQ'
  EmailAddress = 'ThomasL@Reskit.Org'
  GivenName    = 'Thomas'
  Surname      = 'Lee'
  HomePage     = 'Https://tfl09.blogspot.com'
}
Set-ADUser @TLHT
```

11. Viewing updated user

```
Get-ADUser -Identity ThomasL -Properties * |
  Format-Table -Property DisplayName,Name,Office,
             OfficePhone,EmailAddress
```

12. Creating a new domain local group

```
$NGHT = @{
 Name        = 'IT Team'
 Path        = 'OU=IT,DC=Reskit,DC=org'
 Description = 'All members of the IT Team'
 GroupScope  = 'DomainLocal'
}
New-ADGroup @NGHT
```

13. Adding all the users in the IT OU into the IT Team group

```
$SB = 'OU=IT,DC=Reskit,DC=Org'
$ItUsers = Get-ADUser -Filter * -SearchBase $SB
Add-ADGroupMember -Identity 'IT Team' -Members $ItUsers
```

14. Displaying members of the IT Team group

```
Get-ADGroupMember -Identity 'IT Team' |
  Format-Table SamAccountName, DistinguishedName
```

How it works...

You use the `New-ADUser` cmdlet to create a new AD user. Due to the amount of information you wish to hold for any user you create, the number of parameters you might need to pass to `New-ADUser` can make for very long code lines. To get around that, you create a hash table of parameters and parameter values and then use it to create the user. In *step 1*, you create such a hash table, which creates no output.

In *step 2*, you add to the hash table you created in the previous step and create two new AD users. This step creates no output.

With *step 3*, you create a new OU, `IT`, which creates no output. You use this OU to collect user and computer objects for the IT department of Reskit.Org. Next, in *step 4*, which also creates no output, you move the two users you created previously into the IT OU.

Rather than creating a user (which, by default, places the new user object in the Users container in your AD) and later moving it to an OU, you can create a new user object directly in an OU. In *step 5*, you create a third new user directly in the IT OU, which creates no output.

In *step 6*, which generates no output, you create two additional users, which you use later in the recipe. With *step 7*, you use `Get-ADUser` to retrieve all the user objects in the `Reskit.Org` domain, which looks like this:

```
PS C:\Foo> # 7. Viewing existing AD users
PS C:\Foo> Get-ADUser -Filter *  -Property *|
             Format-Table -Property Name, Displayname, SamAccountName

Name           Displayname         SamAccountName
----           -----------         --------------
Administrator                      Administrator
Guest                              Guest
krbtgt                             krbtgt
UK$                                UK$
ThomasL        Thomas Lee (IT)     ThomasL
Rebecca Tanner Rebecca Tanner (IT) RLT
Jerry Garcia   Jerry Garcia (IT)   JerryG
TBR1           User to be removed  TBR1
TBR2           User to be removed  TBR2
```

Figure 6.29: Viewing existing AD users

You can remove any AD user by invoking `Remove-ADUser`. There are two broad patterns for removing a user. The first pattern involves using `Get-ADUser` to get the user you want to remove and then piping that to `Remove-ADUser`. This pattern is appropriate for the command line, where you should always verify you have the right user before deleting that user. The second, possibly more appropriate in scripts, is to remove the user in one operation (and deal with any risks of removing the wrong user). In *step 8*, you use the `Get | Remove` pattern to get the user, then remove it. In *step 9*, you remove the user directly by specifying the user's identity when calling `Remove-ADUser`. Neither *step 8* nor *step 9* generate output.

In *step 10*, you update a user object with new/updated properties. This step generates no output. To see the changes, in *step 11*, you can use `Get-ADUser` and view the properties of the updated user, which looks like this:

```
PS C:\Foo> # 11. Viewing updated user
PS C:\Foo> Get-ADUser -Identity ThomasL -Properties * |
             Format-Table -Property DisplayName,Name,Office,
                                    OfficePhone,EmailAddress

DisplayName     Name    Office     OfficePhone EmailAddress
-----------     ----    ------     ----------- ------------
Thomas Lee (IT) ThomasL Cookham HQ 4416835420  ThomasL@Reskit.Org
```

Figure 6.30: Viewing the updated user

AD enables two types of group account, both of which can contain users and groups: security and distribution. You use distribution groups typically for email systems. Exchange, for example, uses them to implement distribution lists. You use security groups to assign permissions and rights. For example, if you allow a group to have access to a file or folder, then by default, all members of that group have that access permission. Using groups is a best practice when setting permissions and rights.

Security groups in AD have three scopes, which offer different features in terms of members and how you use them to assign permissions and rights. For more details on the different scopes and AD security groups in general, see `https://docs.microsoft.com/windows/security/identity-protection/access-control/active-directory-security-groups`.

In *step 12*, you create a new domain local group, `IT Team`. In *step 13*, you add all the users in the IT OU to this IT Team group. Neither step produces output. In *step 14*, you get and display the members of the IT Team group, which looks like this:

```
PS C:\Foo> # 14. Displaying members of the IT Team group
PS C:\Foo> Get-ADGroupMember -Identity 'IT Team' |
             Format-Table SamAccountName, DistinguishedName

SamAccountName DistinguishedName
-------------- -----------------
JerryG         CN=Jerry Garcia,OU=IT,DC=Reskit,DC=Org
RLT            CN=Rebecca Tanner,OU=IT,DC=Reskit,DC=Org
ThomasL        CN=ThomasL,OU=IT,DC=Reskit,DC=Org
```

Figure 6.31: Displaying members of the IT Team group

There's more...

In *step 7*, you see the users in the Reskit.Org domain. Note the user `UK$`. This user relates to the child domain (`UK.Reskit.Org`). It is not, as such, an actual user. The `$` character at the end of the username indicates it's a hidden user account but fundamental to supporting the child domain. Don't be tempted to tidy up and remove this user account – that would break the child domain structure.

As you see in this recipe, the cmdlets you use to add or modify a user, group, or OU create no output, which cuts down on the output you have to wade through. Some cmdlets and cmdlet sets would output details of the objects created, updated, or possibly deleted. Consider the lack of output for AD cmdlets as a feature.

Managing AD computers

AD computer objects represent domain-joined computers that can use the domain to authenticate user login. Before you can log in as a domain user, such as `Reskit\JerryG`, your computer must be a domain member. When a domain-joined computer starts up, it contacts a DC to authenticate itself. In effect, the computer logs into the domain and creates a secure channel to the DC. Once Windows establishes this secure channel, Windows can log on a user. Under the covers, Windows uses the secure channel to negotiate the user logon.

In this recipe, you work with AD computers and add `SRV1` to the `Reskit.Org` domain.

Getting ready

You run this recipe on `DC1`, a DC in the Reskit.Org domain. This recipe also uses `SRV1`. This host starts as a non-domain-joined Windows Server 2022 host (which you used in earlier chapters in this book). You also use `UKDC1` (the DC in the `UK.Reskit.Org` domain). You should have PowerShell 7 and VS Code installed on each of these hosts.

How to do it...

1. Getting computers in the Reskit domain

```
Get-ADComputer -Filter * |
  Format-Table -Property Name, DistinguishedName
```

2. Getting computers in the UK domain

```
Get-ADComputer -Filter * -Server UKDC1.UK.Reskit.Org |
  Format-Table -Property Name, DistinguishedName
```

3. Creating a new computer in the `Reskit.Org` domain

```
$NCHT = @{
    Name                  = 'Wolf'
    DNSHostName           = 'Wolf.Reskit.Org'
    Description           = 'One for Jerry'
    Path                  = 'OU=IT,DC=Reskit,DC=Org'
}
New-ADComputer @NCHT
```

4. Creating a credential object for `SRV1`

```
$ASRV1    = 'SRV1\Administrator'
$PSRV1    = 'Pa$$w0rd'
$PSSRV1   = ConvertTo-SecureString -String $PSRV1 -AsPlainText -Force
$CredSRV1 = [pscredential]::New($ASRV1, $PSSRV1)
```

5. Creating a script block to join `SRV1`

```
$SB = {
  $ARK    = 'Reskit\Administrator'
  $PRK    = 'Pa$$w0rd'
  $PSRK   = ConvertTo-SecureString -String $PRK -AsPlainText -Force
  $CredRK = [pscredential]::New($ARK, $PSRK)
  $DJHT = @{
    DomainName  = 'Reskit.Org'
    OUPath      = 'OU=IT,DC=Reskit,DC=Org'
    Credential  = $CredRK
    Restart     = $false
 }
    Add-Computer @DJHT
}
```

6. Joining the computer to the domain

```
Set-Item -Path WSMan:\localhost\Client\TrustedHosts -Value '*'
Invoke-Command -ComputerName SRV1 -Credential $CredSRV1 -ScriptBlock $SB
```

7. Restarting `SRV1`

```
Restart-Computer -ComputerName SRV1 -Credential $CredSRV1 -Force
```

8. Viewing the resulting computer accounts for `Reskit.Org`

```
Get-ADComputer -Filter * -Properties DNSHostName,LastLogonDate |
  Format-Table -Property Name, DNSHostName, Enabled
```

How it works...

In *step 1*, you use the `Get-ADComputer` cmdlet to obtain the computer objects defined thus far in the AD, which looks like this:

```
PS C:\Foo> # 1. Getting computers in the Reskit domain
PS C:\Foo> Get-ADComputer -Filter * |
             Format-Table -Property Name, DistinguishedName

Name  DistinguishedName
----  -----------------
DC1   CN=DC1,OU=Domain Controllers,DC=Reskit,DC=Org
DC2   CN=DC2,OU=Domain Controllers,DC=Reskit,DC=Org
UKDC1 CN=UKDC1,CN=Computers,DC=Reskit,DC=Org
```

Figure 6.32: Getting computers in the Reskit domain

In *step 2*, you obtain the computers in the `UK.Reskit.Org` domain from that domain's DC, `UKDC1`, which looks like this:

```
PS C:\Foo> # 2. Getting computers in the UK domain
PS C:\Foo> Get-ADComputer -Filter * -Server UKDC1.UK.Reskit.Org |
             Format-Table -Property Name, DistinguishedName

Name  DistinguishedName
----  -----------------
UKDC1 CN=UKDC1,OU=Domain Controllers,DC=UK,DC=Reskit,DC=Org
```

Figure 6.33: Getting computers in the UK domain

In *step 3,* you create a new computer in the Reskit domain, `Wolf`, or `Wolf.Reskit.Org`. This step creates no output.

To prepare to add `SRV1` to the domain in a single operation, in *step 4*, you create a credential object for `SRV1`'s administrator user. For this credential, you use the generic password used throughout this book. Feel free to use your own password scheme. You need this credential object because `SRV1` is currently a workgroup computer. In *step 5*, you create a script block that, when you run it, adds a computer to the `Reskit.Org` domain. Neither step produces any output.

In *step 6,* you invoke the script block on `SRV1`, which adds `SRV1` to the `Reskit.Org` domain. The output of this step looks like this:

```
PS C:\Foo> # 6. Joining the computer to the domain
PS C:\Foo> Set-Item -Path WSMan:\localhost\Client\TrustedHosts -Value '*'
PS C:\Foo> Invoke-Command -ComputerName SRV1 -Credential $CredSRV1 -ScriptBlock $SB
WARNING: The changes will take effect after you restart the computer SRV1.
```

Figure 6.34: Joining the computer to the domain

To complete the process of joining SRV1 to the Reskit.Org domain, in *step 7*, you reboot SRV1, which creates no output.

After SRV1 has completed its reboot, in *step 8,* you view all the accounts now in the Reskit.Org domain, which looks like this:

```
PS C:\Foo> # 8. Viewing the resulting computer accounts for Reskit.Org
PS C:\Foo> Get-ADComputer -Filter * -Properties DNSHostName,LastLogonDate |
             Format-Table -Property Name, DNSHostName, Enabled

Name  DNSHostName      Enabled
----  -----------      -------
DC1   DC1.Reskit.Org      True
DC2   DC2.Reskit.Org      True
UKDC1 UKDC1.Reskit.Org   False
Wolf  Wolf.Reskit.Org     True
SRV1  SRV1.Reskit.Org     True
```

Figure 6.35: Viewing resulting computer accounts for Reskit.Org

There's more...

There are two broad ways of adding a computer to a domain. The first is to log on to the computer to be added and join the domain. To achieve this, you must have credentials for a user with permissions needed to add a computer to a domain (that is, the domain administrator). You also need the credentials that enable you to log on to the system itself. Alternatively, you can create a computer object in advance, which is known as **pre-staging**. You need administrator credentials for this operation, but once pre-staged any user can join the computer to the domain.

In *step 3*, you pre-stage the `Wolf` computer. A user able to log on to `Wolf` could then use `Add-Computer` (or the GUI) to add the host to the domain. In *step 4*, you add a computer to a domain using domain administrator credentials.

Adding users to AD using a CSV file

Spiceworks (`https://www.spiceworks.com/`) is an excellent site for IT professionals to learn more and get their problems solved. Spiceworks has a busy PowerShell support forum, which you can access at `https://community.spiceworks.com/programming/powershell`.

A frequently asked (and answered) question is: How do I add multiple users using an input file? This recipe does just that. You start with a CSV file containing details of the users you are going to add. Then you run this recipe to add the users.

This recipe uses a CSV file of users to add to AD, with a limited set of properties and values. In production, you would most likely extend the information contained in the CSV, based on your business needs and the information you want to store in AD.

Getting ready

You run this recipe on `SRV1`, a domain-joined server on which you have installed PowerShell 7 and VS Code. Log in as `Reskit\Administrator`. You should also have `DC1` and `DC2` up and running.

This recipe creates and uses a CSV file. As an alternative to using *step 1* in the recipe, you can also download the CSV from GitHub at `https://github.com/PacktPublishing/Windows-Server-Automation-with-PowerShell-7.1-Cookbook-Fourth-Edition/blob/main/Chapter06/goodies/users.csv`. If you download it from GitHub, make sure you store it in `C:\Foo\Users.CSV`.

How to do it...

1. Creating a CSV file

```
$CSVDATA = @'
Firstname, Initials,Lastname,UserPrincipalName,Alias,Description,
Password
J, K,Smith, JKS, James, Data Team, Christmas42
Clair, B, Smith, CBS, Claire, Receptionist, Christmas42
Billy, Bob, JoeBob, BBJB, BillyBob, A Bob, Christmas42
Malcolm, Dudley, Duewrong, Malcolm, Malcolm, Mr Danger, Christmas42
'@
$CSVDATA | Out-File -FilePath C:\Foo\Users.Csv
```

2. Importing and displaying the CSV

```
$Users = Import-CSV -Path C:\Foo\Users.Csv |
  Sort-Object  -Property Alias
$Users | Format-Table
```

3. Adding the users using the CSV

```
$Users |
  ForEach-Object -Parallel {
    $User = $_
    #  Create a hash table of properties to set on created user
    $Prop = @{}
    #  Fill in values
    $Prop.GivenName         = $User.Firstname
```

```
    $Prop.Initials          = $User.Initials
    $Prop.Surname           = $User.Lastname
    $Prop.UserPrincipalName = $User.UserPrincipalName + "@Reskit.Org"
    $Prop.Displayname       = $User.FirstName.Trim() + " " +
                              $User.LastName.Trim()
    $Prop.Description       = $User.Description
    $Prop.Name              = $User.Alias
    $PW = ConvertTo-SecureString -AsPlainText $User.Password -Force
    $Prop.AccountPassword   = $PW
    $Prop.ChangePasswordAtLogon = $true
    $Prop.Path                  = 'OU=IT,DC=Reskit,DC=ORG'
    $Prop.Enabled               = $true
    #  Now Create the User
    New-ADUser @Prop
    # Finally, Display User Created
    "Created $($Prop.Name)"
}
```

4. Showing all users in AD (`Reskit.Org`)

```
Get-ADUser -Filter * |
  Format-Table -Property Name, UserPrincipalName
```

How it works...

In *step 1*, which produces no output, you create a simple CSV file which you save to `C:\Foo\Users.CSV`.

In *step 2*, you import this newly created CSV file and display the information it contains, which looks like this:

```
PS C:\Foo> # 2. Importing and displaying the CSV
PS C:\Foo> $Users = Import-CSV -Path C:\Foo\Users.Csv |
             Sort-Object  -Property Alias
PS C:\Foo> $Users | Format-Table

Firstname Initials Lastname UserPrincipalName Alias    Description  Password
--------- -------- -------- ----------------- -----    -----------  --------
Billy     Bob      JoeBob   BBJB              BillyBob A Bob        Christmas42
Clair     B        Smith    CBS               Claire   Receptionist Christmas42
J         K        Smith    JKS               James    Data Team    Christmas42
Malcolm   Dudley   Duewrong Malcolm           Malcolm  Mr Danger    Christmas42
```

Figure 6.36: Importing and displaying the CSV file

In *step 3*, you add each user contained in the CSV into AD. You add the users using `New-ADUser`, which itself produces no output. This step adds some output to show what users you added, which looks like this:

```
PS C:\Foo> # 3. Adding the users using the CSV
PS C:\Foo> $Users |
             ForEach-Object -Parallel {
               $User = $_
               #  Create a hash table of properties to set on created user
               $Prop = @{}
               #  Fill in values
               $Prop.GivenName         = $User.Firstname
               $Prop.Initials          = $User.Initials
               $Prop.Surname           = $User.Lastname
               $Prop.UserPrincipalName = $User.UserPrincipalName + "@Reskit.Org"
               $Prop.Displayname       = $User.FirstName.Trim() + " " +
                                         $User.LastName.Trim()
               $Prop.Description       = $User.Description
               $Prop.Name              = $User.Alias
               $PW = ConvertTo-SecureString -AsPlainText $User.Password -Force
               $Prop.AccountPassword   = $PW
               $Prop.ChangePasswordAtLogon = $true
               $Prop.Path                  = 'OU=IT,DC=Reskit,DC=ORG'
               $Prop.Enabled               = $true
               #  Now Create the User
               New-ADUser @Prop
               # Finally, Display User Created
               "Created $($Prop.Name)"
             }
Created BillyBob
Created Malcolm
Created Claire
Created James
```

Figure 6.37: Adding users into AD using the CSV file

In the final step in this recipe, *step 4*, you use `Get-ADUser` to view all the users in the `Reskit.Org` domain. This step's output looks like this:

```
PS C:\Foo> # 4. Showing all users in AD (Reskit.Org)
PS C:\Foo> Get-ADUser -Filter * |
             Format-Table -Property Name, UserPrincipalName

Name           UserPrincipalName
----           -----------------
Administrator
Guest
krbtgt
UK$
ThomasL        thomasL@reskit.org
Rebecca Tanner rlt@reskit.org
Jerry Garcia   jerryg@reskit.org
BillyBob       BBJB@Reskit.Org
Malcolm        Malcolm@Reskit.Org
Claire         CBS@Reskit.Org
James          JKS@Reskit.Org
```

Figure 6.38: Viewing all users in the Reskit.Org domain

There's more...

In *step 3*, you add users based on the CSV. You add these users explicitly to the IT OU, using the parameter `-Path` (as specified in the `$Prop` hash table). In production, when adding users that could reside in different OUs, you should extend the CSV to include the distinguished name of the OU into which you wish to add each user.

Creating Group Policy objects

A group policy allows you to define computer and user configuration settings that ensure a system remains configured per policy. Each time a domain-joined computer starts up and each time a domain user logs on, the local group policy agent on your computer obtains the group policy settings from AD and ensures they are applied.

In this recipe, you begin by first creating a GPO within the AD. You then configure the GPO, for example, enabling computers in the IT OU to use PowerShell scripts on those systems or set a specific screensaver. There are thousands of settings you can configure for a user or computer through a group policy. Microsoft has created a spreadsheet that lists the policy settings, which you can download from `https://www.microsoft.com/en-us/download/101451`. At the time of writing, the spreadsheet covers the Group Policy template files delivered with the Windows 10 May 2020 update (that is, Windows 10 2004).

Once you configure your GPO, you link the policy object to the OU you want to configure. You can also apply a GPO to the domain as a whole, to a specific AD site, or to an OU. You can also assign any GPO to multiple OUs, which can simplify your OU design.

The configuration of a GPO typically results in Windows generating information that a host's group policy agent (the code that applies the GPO objects) can access. This information tells the agent how to work. Settings made through administrative templates use registry settings inside `Registry.POL` files. The group policy agent obtains the policy details from the SYSVOL share on a DC and applies them whenever a user logs on or off or when a computer starts up or shuts down. The group policy module also provides the ability to create nice-looking reports describing the GPO.

Getting ready

You run this recipe on `DC1`, a DC in the `Reskit.Org` domain. You created this DC in *Installing an AD forest root domain* and after creating the IT OU. Also, ensure you have created the `C:\Foo` folder on `DC1` before continuing.

How to do it...

1. Creating a group policy object

```
$Pol = New-GPO -Name ITPolicy -Comment 'IT GPO' -Domain Reskit.Org
```

2. Ensuring just computer settings are enabled

```
$Pol.GpoStatus = 'UserSettingsDisabled'
```

3. Configuring the policy with two registry-based settings

```
$EPHT1= @{
  Name      = 'ITPolicy'
  Key       = 'HKLM\Software\Policies\Microsoft\Windows\PowerShell'
  ValueName = 'ExecutionPolicy'
  Value     = 'Unrestricted'
  Type      = 'String'
}
Set-GPRegistryValue @EPHT1 | Out-Null
$EPHT2= @{
  Name   = 'ITPolicy'
  Key    = 'HKLM\Software\Policies\Microsoft\Windows\PowerShell'
  ValueName = 'EnableScripts'
  Type   = 'DWord'
  Value  = 1
}
Set-GPRegistryValue @EPHT2 | Out-Null
```

4. Creating a screensaver GPO

```
$Pol2 = New-GPO -Name 'Screen Saver Time Out'
$Pol2.GpoStatus   = 'ComputerSettingsDisabled'
$Pol2.Description = '15 minute timeout'
```

5. Setting a Group Policy enforced registry value

```
$EPHT3= @{
  Name   = 'Screen Saver Time Out'
  Key    = 'HKCU\Software\Policies\Microsoft\Windows\' +
           'Control Panel\Desktop'
  ValueName = 'ScreenSaveTimeOut'
  Value  = 900
  Type   = 'DWord'
}
Set-GPRegistryValue @EPHT3 | Out-Null
```

6. Linking both GPOs to the IT OU

```
$GPLHT1 = @{
  Name     = 'ITPolicy'
  Target   = 'OU=IT,DC=Reskit,DC=org'
}
New-GPLink @GPLHT1 | Out-Null
$GPLHT2 = @{
  Name     = 'Screen Saver Time Out'
  Target   = 'OU=IT,DC=Reskit,DC=org'
}
New-GPLink @GPLHT2 | Out-Null
```

7. Displaying the GPOs in the domain

```
Get-GPO -All -Domain Reskit.Org |
  Sort-Object -Property DisplayName |
    Format-Table -Property Displayname, Description, GpoStatus
```

8. Creating and viewing a GPO report

```
$RPath = 'C:\Foo\GPOReport1.HTML'
Get-GPOReport -All -ReportType Html -Path $RPath
Invoke-Item -Path $RPath
```

9. Getting report in XML format

```
$RPath2 = 'C:\Foo\GPOReport2.XML'
Get-GPOReport -All -ReportType XML -Path $RPath2
$XML = [xml] (Get-Content -Path $RPath2)
```

10. Creating a simple GPO report

```
$FMTS = "{0,-33}  {1,-30} {2,-10} {3}"
$FMTS -f 'Name', 'Linked To', 'Enabled', 'No Override'
$FMTS -f '----', '---------', '-------', '-----------'
$XML.report.GPO |
  Sort-Object -Property Name |
    ForEach-Object {
     $Gname = $_.Name
     $SOM = $_.linksto.SomPath
     $ENA = $_.linksto.enabled
     $NOO = $_.linksto.nooverride
     $FMTS -f $Gname, $SOM, $ENA, $NOO
   }
```

How it works...

Note that, like many AD-related cmdlets, the cmdlets you use to manage GPOs do not produce much output.

In *step 1*, you create a new GPO in the Reskit.Org domain. This step creates an empty GPO. This GPO is not yet linked to any OU and thus does not get applied.

In *step 2*, you disable user settings, which allows the GPO client to ignore any user settings. Doing so can make the client GPO processing a bit faster.

In *step 3*, you set this GPO to have two specific registry-based values. When a computer starts up, the GPO processing on that client computer ensures that these two registry values are set on the client. During Group Policy refresh (which happens approximately every 2 hours) the group policy agent enforces the value in the policy.

In *step 4* and *step 5*, you create a new GPO and set a screen saver timeout of 900 seconds.

In *step 6*, you link the two GPOs to the IT OU. Until you link the GPOs to an OU (or to the domain or a domain site), GPO processing ignores the GPO.

In this recipe, *step 1* through *step 6* produce no output.

In *step 7*, you use `Get-GPO` to return information about all the GPOs in the domain, which looks like this:

```
PS C:\Foo> # 7. Displaying the GPOs in the domain
PS C:\Foo> Get-GPO -All -Domain Reskit.Org |
             Sort-Object -Property DisplayName |
               Format-Table -Property Displayname, Description, GpoStatus

DisplayName                       Description GpoStatus
-----------                       ----------- ---------
Default Domain Controllers Policy             AllSettingsEnabled
Default Domain Policy                         AllSettingsEnabled
ITPolicy                          IT GPO      AllSettingsEnabled
Screen Saver Time Out                         AllSettingsEnabled
```

Figure 6.39: Displaying all the GPOs in the domain

In *step 8*, you generate and display a GPO report by using the `Get-GPOReport` command. The command produces no output, but by using `Invoke-Command`, you view the report in your default browser, which looks like this:

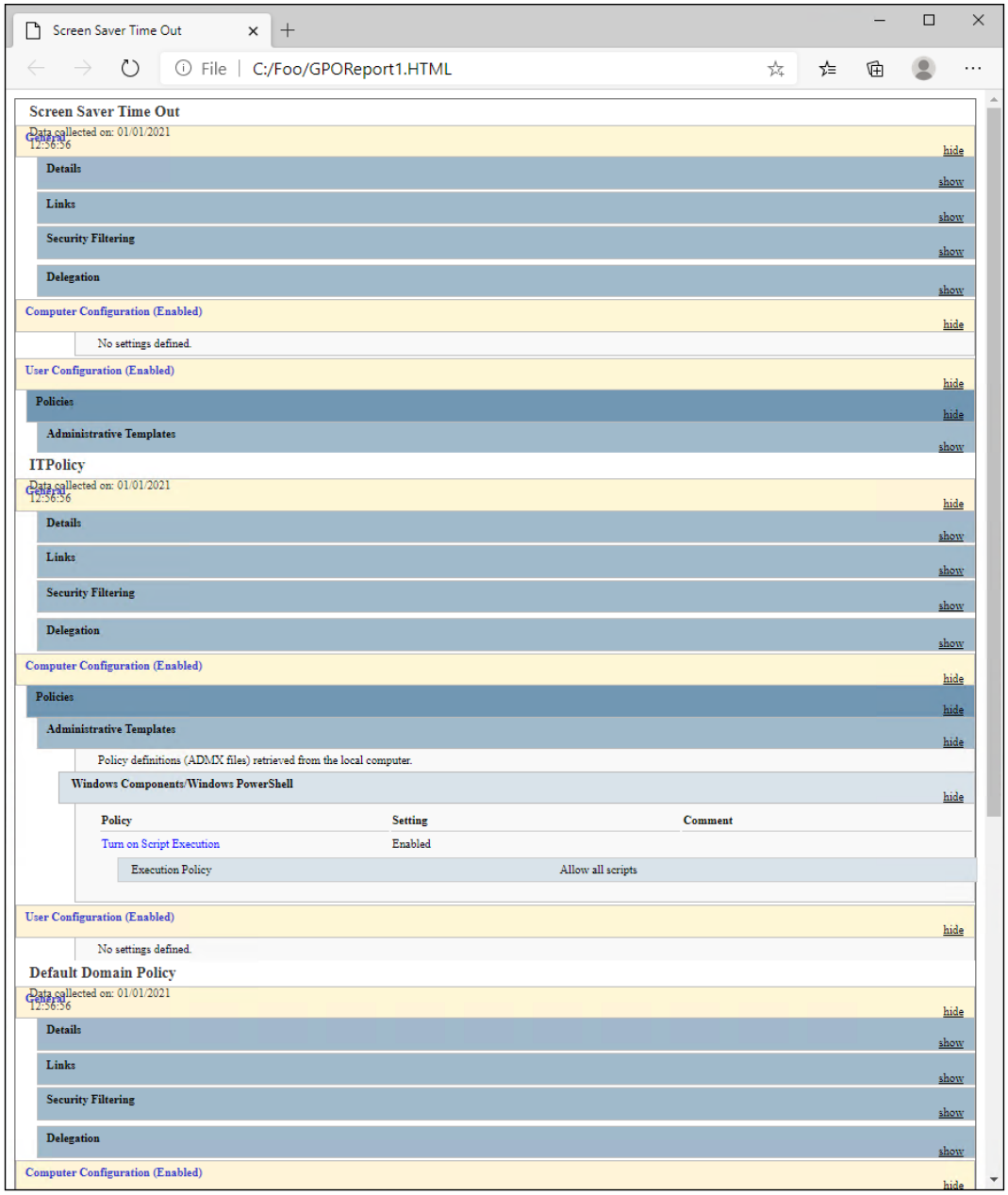


Figure 6.40: Viewing the GPO report in browser

In *step 9*, you use `Get-GPOReport` to return a report of all the GPOs in the domain in an XML format, which produces no output.

In *step 10*, you iterate through the returned XML and produce a simple report on GPOs and where they are linked, which looks like this:

```
PS C:\Foo> # 10. Creating simple GPO report
PS C:\Foo> $RPath2 = 'C:\Foo\GPOReport2.XML'
PS C:\Foo> $FMTS = "{0,-33}  {1,-30} {2,-10} {3}"
PS C:\Foo> $FMTS -f 'Name','Linked To', 'Enabled', 'No Override'
PS C:\Foo> $FMTS -f '----','---------', '-------', '-----------'
PS C:\Foo> $XML.report.GPO |
             Sort-Object -Property Name |
               ForEach-Object {
                $Gname = $_.Name
                $SOM = $_.linksto.SomPath
                $ENA = $_.linksto.enabled
                $NOO = $_.linksto.nooverride
                $FMTS -f $Gname, $SOM, $ENA, $NOO
               }

Name                              Linked To                      Enabled    No Override
----                              ---------                      -------    -----------
Default Domain Controllers Policy Reskit.Org/Domain Controllers  true       false
Default Domain Policy             Reskit.Org                     true       false
ITPolicy                          Reskit.Org/IT                  true       false
Screen Saver Time Out             Reskit.Org/IT                  true       false
```

Figure 6.41: Creating a simple report on GPOs

There's more...

In *step 8*, you see the output from the `Get-GPOReport` cmdlet. At the time of writing, the Edge browser does not render the output as nicely as you might want. What may look like a bug in the graphic is just how Edge (again, at the time of writing) renders this HTML document.

In *step 9* and *step 10*, you create your mini-report. In *step 9*, you use the `Get-GPOReport` command to obtain a report of all GPOs in the domain returned as XML. In *step 10*, you report on the GPOs in the Reskit domain using .NET composite string formatting with the `-f` operator.

Using .NET composite formatting enables you to create nice-looking output when the objects returned by a cmdlet are not in a form to be used directly with `Format-Table`. In *step 9*, for example, the XML returned contains details of the GPO links as a property which is actually an object with sub-properties. To create nice-looking output, you create a format string you use to display each row of your report, including report header lines. You first use this format string to create the two report header lines for the report. Then, for each GPO in the returned list, you obtain the GPO name, which OU the GPO is linked to, whether it is enabled, and whether the GPO is non-overridable. Finally, you use the format string to output a single report line. In most cases, these custom reports are easier to read and contain only the information you deem useful.

As an alternative to creating a report as shown in *step 9*, you could have created a custom hash table and used it to create a customized object for your report.

The `Format-Table` and `Format-List` cmdlets are of most use when each object has simple properties. When an object has a property that is an object with properties, the format commands do not surface these (sub)properties. In this case, you must obtain the details manually of the GPO links for each GPO and generate your report lines based on those results. While the report layout works well for this specific set of GPOs, should you have GPOs with longer names, or linked to deeper OUs, you may need to adjust the format string you set in *step 10*.

Reporting on AD replication

AD uses a special database to support its operations. The database is a distributed, multi-master database with convergence – every DC in every domain stores this database in the file `C:\Windows\NTDS\ntds.dit`.

Every DC in any domain holds a complete copy of this database. If you add a new user or change a user's office, that change occurs on just one DC (initially). AD replication makes the change in all database copies. In this way, the database remains consistent over time and across all DCs.

AD replication is based on partitions – a slice of the overall database. AD can replicate each partition separately. There are several partitions in AD:

- **Schema partition**: This holds the AD schema that defines each object that AD stores in the database. The schema also defines the properties of all these objects.
- **Configuration partition**: This holds the details of the structure of the domain.
- **Domain partition**: This partition, also known as the domain naming context, contains the objects relating to a domain (users, groups, OUs, and so on). The objects in this partition are defined based on the schema.
- **Application partition**: Some applications, such as DNS, store objects in your AD and rely on AD replication to replicate the values.

There are two types of replication: intra-site replication and inter-site replication. Intra-site replication happens between DCs in a given AD site, while inter-site replication occurs between different sites.

You can create different topologies for replication, including:

- **Ring**: Each DC in a site has at least two inbound replication partners. When any change is made to any DC, that DC notifies its replication partners that it has a change. Those DCs can then replicate that change (if they have not seen the change before). AD by default ensures there are no more than three hops within the replication topology. If you have a large number of DCs (more than seven), AD automatically creates additional replication links to keep the hop count below three.
- **Full mesh**: With this topology, all DCs replicate to all others. Full mesh replication keeps the database in sync with a minimum of replication delay, but can be more expensive in terms of bandwidth (and DC utilization). It is not scalable.
- **Hub and spoke**: You might use this approach in enormous organizations where a "spoke" DC replicates with a central hub DC. The hub DC then replicates the change to all other spoke DCs in your organization. Hub and spoke can reduce replication for widely dispersed implementations.
- **Hybrid**: Here you can combine any of the above, based on business needs.

By default, AD replication uses a ring topology. You can adopt different topologies if your business needs dictate, but this requires configuration.

In most smaller organizations (such as `Reskit.Org`), replication is set up and operates automagically. But for large and distributed organizations, replication can be quite complicated as you attempt to balance being totally up to date against the cost of geographical bandwidth. If you have DCs in several continents, you want to collect those changes and do them all at once, say every 4 hours. But that means the remote DC would have out-of-date information for a period. As a general rule of thumb, you should design replication so that it happens faster than a person can fly between your AD sites served by a DC. If you change your password, say in London, then as long as the changes occur within 12 hours, when you fly to, say Brisbane, Australia, the Australian DCs contain the replicated password. Thus, you can log in with the new password immediately upon landing in Brisbane.

For more information on AD replication concepts, see `https://docs.microsoft.com/windows-server/identity/ad-ds/get-started/replication/active-directory-replication-concepts`.

For any sizeable organization, the design and planning of AD is vital – see `https://docs.microsoft.com/windows-server/identity/ad-ds/plan/ad-ds-design-and-planning` for more information on the planning and design work necessary.

Traditionally, you use many Win32 console applications to manage and troubleshoot replication, including `repadmin.exe`, which broadly replaces an earlier command `replmon.exe`. For some detail on the `repadmin.exe` command (and replication) see `https://techcommunity.microsoft.com/t5/ask-the-directory-services-team/getting-over-replmon/ba-p/396687`.

With the advent of PowerShell and the PowerShell AD module, you can now perform many of the functions of `repadmin.exe` using PowerShell cmdlets. In this recipe, you examine some of the details of AD replication using PowerShell.

Getting ready

You run this recipe on `DC1`, a DC in the Reskit.Org domain. You should also have `DC2` and `UKDC1` available and online at the start of testing this recipe. You must have installed PowerShell 7 and VS Code on all these hosts.

How to do it...

1. Checking replication partners for `DC1`

```
Get-ADReplicationPartnerMetadata -Target DC1.Reskit.Org   |
  Format-List -Property Server, PartnerType, Partner,
              Partition, LastRep*
```

2. Checking AD replication partner metadata in the domain

```
Get-ADReplicationPartnerMetadata -Target Reskit.Org -Scope Domain |
  Format-Table -Property Server, P*Type,Last*
```

3. Investigating group membership metadata

```
$REPLHT = @{
  Object              = (Get-ADGroup -Identity 'IT Team')
  Attribute           = 'Member'
  ShowAllLinkedValues = $true
  Server              = (Get-ADDomainController)
}
Get-ADReplicationAttributeMetadata @REPLHT |
  Format-Table -Property A*NAME, A*VALUE, *TIME
```

4. Adding two users to the group and removing one

```
Add-ADGroupMember -Identity 'IT Team' -members Malcolm
Add-ADGroupMember -Identity 'IT Team' -members Claire
Remove-ADGroupMember -Identity 'IT Team' -members Claire -Confirm:$False
```

5. Checking updated metadata

```
Get-ADReplicationAttributeMetadata @REPLHT |
  Format-Table -Property A*NAME, A*VALUE, *TIME
```

6. Creating an initial replication failure report

```
$DomainController = 'DC1'
$Report = [ordered] @{}
## Replication Partners ##
$ReplMeta =
    Get-ADReplicationPartnerMetadata -Target $DomainController
$Report.ReplicationPartners = $ReplMeta.Partner
$Report.LastReplication     = $ReplMeta.LastReplicationSuccess
## Replication Failures ##
$REPLF = Get-ADReplicationFailure -Target $DomainController
$Report.FailureCount  = $REPLF.FailureCount
$Report.FailureType   = $REPLF.FailureType
$Report.FirstFailure  = $REPLF.FirstFailureTime
$Report.LastFailure   = $REPLF.LastFailure
$Report
```

7. Simulating a connection issue

```
Stop-Computer DC2  -Force
Start-Sleep -Seconds 30
```

8. Making a change to this AD

```
Get-AdUser -identity BillyBob  |
  Set-AdUser -Office 'Cookham Office' -Server DC1
```

9. Using `repadmin.exe` to generate a status report

```
repadmin /replsummary
```

How it works...

In *step 1*, you check to discover replication partners for `DC1`. With two DCs in the Reskit domain (`DC1` and `DC2`), the output of this step looks like this:

```
PS C:\Foo> # 1. Checking replication partners for DC1
PS C:\Foo> Get-ADReplicationPartnerMetadata -Target DC1.Reskit.Org   |
             Format-List -Property Server, PartnerType, Partner,
                                   Partition, LastRep*

Server                : DC1.Reskit.Org
PartnerType           : Inbound
Partner               : CN=NTDS Settings,CN=DC2,CN=Servers,CN=Default-First-Site-Name,CN=Sites,CN=Configuration,DC=Reskit,DC=Org
Partition             : DC=Reskit,DC=Org
LastReplicationAttempt : 02/01/2021 15:45:10
LastReplicationResult  : 0
LastReplicationSuccess : 02/01/2021 15:45:10
```

Figure 6.42: Checking replication partners for DC1

In *step 2*, you check on the replication partner metadata for the entire domain, which looks like this:

```
PS C:\Foo> # 2. Checking AD replication partner metadata in the domain
PS C:\Foo> Get-ADReplicationPartnerMetadata -Target Reskit.Org -Scope Domain |
             Format-Table -Property Server, P*Type, Last*

Server          PartnerType LastChangeUsn LastReplicationAttempt LastReplicationResult LastReplicationSuccess
------          ----------- ------------- ---------------------- --------------------- ----------------------
DC1.Reskit.Org      Inbound        172044 02/01/2021 16:10:13                         0 02/01/2021 16:10:13
DC2.Reskit.Org      Inbound        107760 02/01/2021 16:14:13                         0 02/01/2021 16:14:13
```

Figure 6.43: Checking replication partner metadata in the domain

In *step 3*, you examine the metadata for group membership. You take this step after creating the `IT Team` security group and populating it, at which point the output looks like this:

```
PS C:\Foo> # 3. Investigating group membership metadata
PS C:\Foo> $REPLHT = @{
             Object              = (Get-ADGroup -Identity 'IT Team')
             Attribute           = 'Member'
             ShowAllLinkedValues = $true
             Server              = (Get-ADDomainController)
           }
PS C:\Foo> Get-ADReplicationAttributeMetadata @REPLHT |
             Format-Table -Property A*NAME,A*VALUE, *TIME

AttributeName AttributeValue                          FirstOriginatingCreateTime LastOriginatingChangeTime LastOriginatingDeleteTime
------------- --------------                          -------------------------- ------------------------- -------------------------
member        CN=Jerry Garcia,OU=IT,DC=Reskit,DC=Org  06/12/2020 16:33:54        06/12/2020 16:33:54       01/01/1601 00:00:00
member        CN=Rebecca Tanner,OU=IT,DC=Reskit,DC=Org 06/12/2020 16:33:54       06/12/2020 16:33:54       01/01/1601 00:00:00
member        CN=ThomasL,OU=IT,DC=Reskit,DC=Org       06/12/2020 16:33:54        06/12/2020 16:33:54       01/01/1601 00:00:00
```

Figure 6.44: Examining group membership metadata

In *step 4*, which creates no output, you change this group by adding two members. Then, you remove one of them from the group. Under the covers, this step updates the group membership (three times), which generates replication traffic to replicate the group membership changes from `DC1` to `DC2`.

With *step 5*, you re-examine the group membership metadata to see the effects of *step 4*, which looks like this:

```
PS C:\Foo> # 5. Checking the updated metadata
PS C:\Foo> Get-ADReplicationAttributeMetadata @REPLHT |
             Format-Table -Property A*NAME,A*VALUE, *TIME

AttributeName AttributeValue                          FirstOriginatingCreateTime LastOriginatingChangeTime LastOriginatingDeleteTime
------------- --------------                          -------------------------- ------------------------- -------------------------
member        CN=Claire,OU=IT,DC=Reskit,DC=Org        02/01/2021 15:57:45        02/01/2021 15:58:17       02/01/2021 15:58:17
member        CN=Malcolm,OU=IT,DC=Reskit,DC=Org       02/01/2021 15:57:41        02/01/2021 15:57:41       01/01/1601 00:00:00
member        CN=Jerry Garcia,OU=IT,DC=Reskit,DC=Org  06/12/2020 16:33:54        06/12/2020 16:33:54       01/01/1601 00:00:00
member        CN=Rebecca Tanner,OU=IT,DC=Reskit,DC=Org 06/12/2020 16:33:54       06/12/2020 16:33:54       01/01/1601 00:00:00
member        CN=ThomasL,OU=IT,DC=Reskit,DC=Org       06/12/2020 16:33:54        06/12/2020 16:33:54       01/01/1601 00:00:00
```

Figure 6.45: Checking the updated metadata

In *step 6*, you create a report of replication failures, with output like this:

```
PS C:\Foo> # 6. Creating an initial replication failure report
PS C:\Foo> $DomainController = 'DC1'
PS C:\Foo> $Report = [ordered] @{}
PS C:\Foo> ## Replication Partners ##
PS C:\Foo> $ReplMeta =
             Get-ADReplicationPartnerMetadata -Target $DomainController
PS C:\Foo> $Report.ReplicationPartners = $ReplMeta.Partner
PS C:\Foo> $Report.LastReplication     = $ReplMeta.LastReplicationSuccess
PS C:\Foo> ## Replication Failures ##
PS C:\Foo> $REPLF = Get-ADReplicationFailure -Target $DomainController
PS C:\Foo> $Report.FailureCount  = $REPLF.FailureCount
PS C:\Foo> $Report.FailureType   = $REPLF.FailureType
PS C:\Foo> $Report.FirstFailure  = $REPLF.FirstFailureTime
PS C:\Foo> $Report

Name                           Value
----                           -----
FailureCount                   {0, 0}
LastReplication                02/01/2021 16:10:13
FirstFailure                   {21/12/2020 23:56:06, 21/12/2020 23:56:06}
ReplicationPartners            CN=NTDS Settings,CN=DC2,CN=Servers,CN=Default-First-Site-Name,
                               CN=Sites,CN=Configuration,DC=Reskit,DC=Org
FailureType                    {Link, Link}
```

Figure 6.46: Creating a report of replication failures

In *step 7*, you simulate a connection issue by stopping `DC2` and waiting until the system has completed its shutdown. In *step 8*, you make a change to a user (on `DC1`). These steps create no output, although *step 7* results in AD attempting to replicate the changed user account to `DC2`.

In *step 9*, you use the `repadmin.exe` command to generate a replication summary report, which looks like this:

```
PS C:\Foo> # 9. Using Repadmin to generate a status report
PS C:\Foo> repadmin /replsummary
Replication Summary Start Time: 2021-01-02 16:46:26

Beginning data collection for replication summary, this may take awhile:
  ......

Source DSA          largest delta    fails/total %%   error
 DC1                       46m:33s    0 /   3    0
 DC2                    01h:47m:36s    4 /   7   57  (1722) The RPC server is unavailable.
 UKDC1                  01h:01m:16s    0 /   3    0

Experienced the following operational errors trying to retrieve replication information:
          58 - DC2.Reskit.Org
```

Figure 6.47: Generating a replication status report using repadmin

There's more...

In *step 2*, you discover the replication partners within the domain. In a domain with just two DCs, you would expect each DC to be an inbound replication partner to the other DC, and that is what the figure shows.

In *step 3* through *step 5*, you examine the change in replication data before and after a group membership change. Using the `Get-ADReplicationAttributeMetadata` cmdlet is an excellent way to discover the specific changes to a security group's membership, including removing a member. The information provided does not tell you who made the change or the system where the update originated. But knowing that someone made a change and when are essential first steps.

In *step 7*, you create a potential replication issue by stopping `DC2`. In *step 8*, you generate a change to the AD on `DC1`, which would typically be replicated very quickly to `DC2`. But since `DC2` is down, that replication cannot happen.

In *step 9*, you use the `repadmin.exe` console application to generate a replication summary showing that `DC1` cannot contact `DC2`. Sometimes, older tools, like `repadmin.exe`, produce better output than the cmdlets. For console usage and when you want a simple summary of replication, `repadmin.exe` is still a great tool.

Reporting on AD computers

Monitoring the AD is a necessary albeit time-consuming task. With larger numbers of users and computers to manage, you need all the help you can get, and PowerShell makes it easy to keep track of things.

A computer that has not logged on for an extended period could represent a security risk or could be a lost/stolen computer. It could also be a system that you have not rebooted after having applied patches and updates.

This recipe creates a report of computers that have not logged on or that you have not rebooted for a while.

One challenge in developing scripts like this is creating meaningful test data. If you wish to generate a test report showing a system that has not logged in for over 6 months, you might have to wait for 6 months to get the necessary data. This recipe shows a way around that for testing purposes.

Getting ready

You run this recipe on `DC1`, a DC in the Reskit domain on which you have installed PowerShell 7 and VS Code. Also, you should have completed earlier recipes that created some computer objects inside the AD, including *Creating and managing AD users and groups*.

How to do it...

1. Creating example computer accounts in the AD

```
$NCHT1 = @{
    Name        = 'NLIComputer1_1week'
    Description = 'Computer last logged in 1 week ago'
}
New-ADComputer @NCHT1
$NCHT2 = @{
  Name        = 'NLIComputer2_1month'
  Description = 'Computer last logged in 1 week ago'
}
New-ADComputer @NCHT2
$NCHT3 = @{
  Name        = 'NLIComputer3_6month'
  Description = 'Computer last logged in 1 week ago'
}
New-ADComputer @NCHT3
```

2. Creating some constants for later comparison

```
$OneWeekAgo   = (Get-Date).AddDays(-7)
$OneMonthAgo  = (Get-Date).AddMonths(-1)
$SixMonthsAgo = (Get-Date).AddMonths(-6)
```

3. Defining a function to create sample data

```
Function Get-RKComputers {
$ADComputers = Get-ADComputer -Filter * -Properties LastLogonDate
$Computers = @()
foreach ($ADComputer in $ADComputers) {
  $Name = $ADComputer.Name
  # Real computers and last logon date
  if ($adComputer.name -NotMatch '^NLI') {
    $LLD = $ADComputer.LastLogonDate
  }
  Elseif ($ADComputer.Name -eq 'NLIComputer1_1week')  {
    $LLD = $OneWeekAgo.AddMinutes(-30)
  }
  Elseif ($ADComputer.Name -eq 'NLIComputer2_1month')  {
    $LLD = $OneMonthAgo.AddMinutes(-30)
  }
  Elseif ($ADComputer.Name -eq 'NLIComputer3_6month')  {
    $LLD = $SixMonthsAgo.AddMinutes(-30)
  }
```

```
  $Computers += [pscustomobject] @{
    Name = $Name
    LastLogonDate = $LLD
  }
 }
 $Computers
}
```

4. Building the report header

```
$RKReport = ''           # Start of report
$RKReport += "*** Reskit.Org AD Daily AD Computer Report`n"
$RKReport += "*** Generated [$(Get-Date)]`n"
$RKReport += "*********************************`n`n"
```

5. Getting computers in RK AD using `Get-RKComputers`

```
$Computers = Get-RKComputers
```

6. Getting computers that have never logged on

```
$RKReport += "Computers that have never logged on`n"
$RkReport += "Name                    LastLogonDate`n"
$RkReport += "----                    -------------`n"
$RKReport += Foreach($Computer in $Computers) {
  If ($null -eq $Computer.LastLogonDate) {
   "{0,-22}  {1}  `n" -f $Computer.Name, "Never"
  }
}
```

7. Reporting on computers that have not logged on in over 6 months

```
foreach($Computer in $Computers) {
  If (($Computer.LastLogonDate -lt $SixMonthsAgo) -and
      ($null -ne $Computer.LastLogonDate)) {
("`n{0,-23}  {1}  `n" -f $Computer.Name, $Computer.LastLogonDate)
  }
}
```

8. Reporting on computer accounts that have not logged in for 1–6 months

```
$RKReport += "`n`nComputers that havent logged in 1-6 months`n"
$RkReport += "Name                    LastLogonDate`n"
$RkReport += "----                    -------------"
$RKReport +=
foreach($Computer in $Computers) {
  If (($Computer.LastLogonDate -ge $SixMonthsAgo) -and
     ($Computer.LastLogonDate -lt $OneMonthAgo) -and
       ($null -ne $Computer.LastLogonDate)) {
```

```
    "`n{0,-22}  {1}  " -f $Computer.Name, $Computer.LastLogonDate
  }
}
```

9. Reporting on computer accounts that have not logged in for the past 1 week to 1 month

```
$RKReport += "`n`nComputers that have between one week "
$RKReport += "and one month ago`n"
$RkReport += "Name                    LastLogonDate`n"
$RkReport += "----                    -------------"
$RKReport +=
foreach($Computer in $Computers) {
  If (($Computer.LastLogonDate -ge $OneMonthAgo) -and
     ($Computer.LastLogonDate -lt $OneWeekAgo) -and
       ($null -ne $Computer.LastLogonDate)) {
   "`n{0,-22}  {1}  " -f $Computer.Name, $Computer.LastLogonDate
  }
}
```

10. Displaying the report

```
$RKReport
```

How it works...

In this recipe, all but the final step produce no output. Some of the steps exist to enable you to test the report that this recipe generates. Some of these steps might not be necessary for real life, if you already have enough real-life data to create a complete report.

In *step 1*, you create three AD computer accounts. You use these accounts to simulate computers that have not logged on for a while, thus enabling you to view a complete report. If you re-run this recipe or this step, adding these accounts produces errors since the accounts already exist. You could modify this step to check to see if the accounts exist before creating them.

In *step 2*, you create three time constants, representing the time 7 days ago, 1 month ago, and 6 months ago. This step enables you to test if a given user account has not logged on in that period.

In *step 3*, you create a new function, `Get-RKComputers`. This function returns a list of all the computer accounts in AD along with their last logon time.

In *step 4*, you begin the report by creating a report header.

In *step 5*, you call the `Get-RKComputers` and populate the `$Computers` array (all the computers available).

In *step 6* through *step 9*, you add details to the report of computers that have never logged on, have not logged on in over 6 months, have not logged on in 1–6 months, and computers that have not logged on in 1 week to 1 month.

In the final step, *step 10*, you display the report created by the earlier steps. The output of this step looks like this:

```
PS C:\Foo> # 10. Displaying the report
PS C:\Foo> $RKReport

*** Reskit.Org AD Daily AD Computer Report
*** Generated [01/08/2021 21:42:01]
**********************************

Computers that have never logged on
Name                    LastLogonDate
----                    -------------
Wolf                    Never

Computers that havent logged in over 6 months
Name                    LastLogonDate
----                    -------------
NLIComputer3_6month      08/07/2020 21:12:21

Computers that havent logged in 1-6 months
Name                    LastLogonDate
----                    -------------
UKDC1                   03/12/2020 14:43:01
NLIComputer2_1month     08/12/2020 21:12:21

Computers that have between one week and one month ago
Name                    LastLogonDate
----                    -------------
DC1                     30/12/2020 23:57:33
DC2                     30/12/2020 18:28:33
SRV1                    01/01/2021 14:41:18
NLIComputer1_1week      01/01/2021 21:12:21
```

Figure 6.48: Displaying the report

There's more...

In *step 3*, you create a function to get the computer accounts in AD. This function returns an array of computer names and the last logon date. In production, you might amend this function to return computer accounts from just certain OUs. You could also extend this function to test whether each computer is online by testing a network connection to the computer, or checking to see if there is a DNS A record for the computer to detect stale computer accounts.

In *step 5* through *step 9*, you create a report by adding text lines to the `$RKReport` variable. In doing so, you need to add CRLF characters before or after each line of text when you add each line to the report. Ensuring each line of the report begins in the right place can be challenging in creating reports using the technique shown by this recipe.

Reporting on AD users

In the previous recipe, you created a report on computer accounts that may be of interest. User and group accounts are also worth tracking. If a user has not logged on for a reasonable period, the account could be a security risk. Likewise, a user with membership in a privileged account (for example, Enterprise Admins) could be used by an attacker. IT professionals know how much easier it is just to put someone in a high-privilege group than to set up more fine-grained permissions using something like Just Enough Administration (see the *Implementing Just Enough Administration (JEA)* recipe in *Chapter 8*, *Implementing Enterprise Security*).

Regular reporting can help focus on accounts that could be deactivated, removed from a security group, or removed altogether.

In this recipe, you obtain all the accounts in the AD and examine potential security risks.

Getting ready

You run this recipe on `DC1`, a DC in the `Reskit.Org` domain, after running the recipes in this chapter. You should also have installed PowerShell 7 and VS Code on this host.

How to do it...

1. Defining a function `Get-ReskitUser` to return objects related to users in Reskit.Org domain

```
Function Get-ReskitUser {
# Get PDC Emulator DC
$PrimaryDC = Get-ADDomainController -Discover -Service PrimaryDC
# Get Users
$ADUsers = Get-ADUser -Filter * -Properties * -Server $PrimaryDC
# Iterate through them and create $Userinfo hash table:
Foreach ($ADUser in $ADUsers) {
    # Create a userinfo HT
    $UserInfo = [Ordered] @{}
    $UserInfo.SamAccountname = $ADUser.SamAccountName
    $Userinfo.DisplayName    = $ADUser.DisplayName
    $UserInfo.Office         = $ADUser.Office
    $Userinfo.Enabled        = $ADUser.Enabled
    $userinfo.LastLogonDate  = $ADUser.LastLogonDate
    $UserInfo.ProfilePath    = $ADUser.ProfilePath
    $Userinfo.ScriptPath     = $ADUser.ScriptPath
    $UserInfo.BadPWDCount    = $ADUser.badPwdCount
    New-Object -TypeName PSObject -Property $UserInfo
    }
} # end of function
```

2. Getting the users

```
$RKUsers = Get-ReskitUser
```

3. Building the report header

```
$RKReport =  ''  # first line of the report
$RkReport += "*** Reskit.Org AD Report`n"
$RKReport += "*** Generated [$(Get-Date)]`n"
$RKReport += "*******************************`n`n"
```

4. Reporting on disabled users

```
$RkReport += "*** Disabled Users`n"
$RKReport += $RKUsers |
    Where-Object {$_.Enabled -NE $true} |
        Format-Table -Property SamAccountName, Displayname |
            Out-String
```

5. Reporting on users who have not recently logged on

```
$OneWeekAgo = (Get-Date).AddDays(-7)
$RKReport += "`n*** Users Not logged in since $OneWeekAgo`n"
$RkReport += $RKUsers |
  Where-Object {$_.Enabled -and $_.LastLogonDate -le $OneWeekAgo} |
        Sort-Object -Property LastlogonDate |
            Format-Table -Property SamAccountName,lastlogondate |
                Out-String
```

6. Discovering users with a high number of invalid password attempts

```
$RKReport += "`n*** High Number of Bad Password Attempts`n"
$RKReport += $RKUsers | Where-Object BadPwdCount -ge 5 |
  Format-Table -Property SamAccountName, BadPwdCount |
    Out-String
```

7. Adding another report header line for this part of the report and creating an empty array of privileged users

```
$RKReport += "`n*** Privileged  User Report`n"
$PUsers = @()
```

8. Querying the Enterprise Admins/Domain Admins/Scheme Admins groups for members and adding them to the `$PUsers` array

```
# Get Enterprise Admins group members
$Members = Get-ADGroupMember -Identity 'Enterprise Admins' -Recursive |
    Sort-Object -Property Name
$PUsers += foreach ($Member in $Members) {
    Get-ADUser -Identity $Member.SID -Properties * |
        Select-Object -Property Name,
               @{Name='Group';expression={'Enterprise Admins'}},
               whenCreated,LastLogonDate
}
# Get Domain Admins group members
$Members =
  Get-ADGroupMember -Identity 'Domain Admins' -Recursive |
    Sort-Object -Property Name
$PUsers += Foreach ($Member in $Members)
    {Get-ADUser -Identity $member.SID -Properties * |
        Select-Object -Property Name,
                @{Name='Group';expression={'Domain Admins'}},
                WhenCreated, LastLogonDate,SamAccountName
}
```

```
# Get Schema Admins members
$Members =
  Get-ADGroupMember -Identity 'Schema Admins' -Recursive |
    Sort-Object Name
$PUsers += Foreach ($Member in $Members) {
    Get-ADUser -Identity $member.SID -Properties * |
        Select-Object -Property Name,
            @{Name='Group';expression={'Schema Admins'}}, `
            WhenCreated, Lastlogondate,SamAccountName
}
```

9. Adding the special users to the report

```
$RKReport += $PUsers | Out-String
```

10. Displaying the final report

```
$RKReport
```

How it works...

In this recipe, all the steps except the last one produce no output. The steps create a report which you view in the final step.

In *step 1*, you create a function, `Get-ReskitUser`, which creates a set of user objects related to each of the users in your AD. In *step 2*, you use the function to populate an array, `$RKUsers`, containing users and necessary details needed for your report.

With *step 3*, you build the header for the report, and then in *step 4*, you build a report section on disabled users. You use *step 4* to add details of disabled user accounts. In *step 5*, you add details of users that have not logged on within the last 7 days. In *step 6*, you add details of users who have had more than five unsuccessful login attempts.

The final section of the report lists users that are members of crucial AD groups. With *step 7*, you create a header for this section. Then with *step 8* and *step 9*, you add details of members of these groups.

Once these steps are complete, in *step 10*, you can view the output of the report, which looks like this:

```
C:\Foo> # 10. Displaying the final report
C:\Foo> $RKReport
*** Reskit.Org AD Report
*** Generated [01/09/2021 14:28:58]
******************************

*** Disabled Users

SamAccountname DisplayName
-------------- -----------
Guest
krbtgt


*** Users Not logged in since 01/02/2021 14:28:58

SamAccountname LastLogonDate
-------------- -------------
UK$
RLT
BillyBob
Claire
James


*** High Number of Bad Password Attempts

SamAccountname BadPWDCount
-------------- -----------
Malcolm                  6


*** Privileged  User Report

Name          Group             whenCreated         LastlogonDate
----          -----             -----------         -------------
Administrator Enterprise Admins 03/12/2020 12:04:13 03/01/2021 17:09:59
Malcolm       Enterprise Admins 06/12/2020 20:33:28 09/01/2021 14:24:06
Administrator Domain Admins     03/12/2020 12:04:13 03/01/2021 17:09:59
Jerry Garcia  Domain Admins     06/12/2020 16:25:45 09/01/2021 14:18:16
ThomasL       Domain Admins     06/12/2020 16:23:29 09/01/2021 14:16:37
Administrator Schema Admins     03/12/2020 12:04:13 03/01/2021 17:09:59
```

Figure 6.49: Displaying the final report

There's more...

In *step 1*, you create a function to retrieve users. This function allows you to reformat the properties as needed, improving the output in your report.

In *step 4* through *step 8*, you build the report contents. To create the output you see in the above output, you must log in to a host in the domain. Otherwise, these sections would be blank and contain no users.

In *step 9*, you see the final report. Note that the user Malcolm both had a high number of failed login attempts and has logged in at some point successfully. If you log in using a domain account and an incorrect password, AD rejects the login and increases the bad attempt count. Once you log in successfully, however, the bad logon count is zeroed.

7

Managing Networking in the Enterprise

In this chapter, we cover the following recipes:

- Configuring IP addressing
- Testing network connectivity
- Installing DHCP
- Configuring DHCP scopes and options
- Using DHCP
- Implementing DHCP failover and load balancing
- Deploying DNS in the Enterprise
- Configuring DNS forwarding
- Managing DNS zones and resource records

Introduction

Every organization's heart is its network – the infrastructure that enables your client and server systems to interoperate. Windows has included networking features since the early days of Windows for Workgroups 3.1 (and earlier with Microsoft LAN Manager).

One thing worth noting is that even in the cloud age, "the network" isn't going anywhere; its role of enabling a client to connect to a server is just changing, with some servers (and clients) now in the cloud. The cloud is really just resources in someone else's network, and you still need the network to communicate.

Every server or workstation in your environment needs to have a correct IP configuration. While IPv6 is gaining in popularity, most organizations rely on IPV4. In the *Configuring IP addressing* recipe, we look at setting a network interface's IPv4 configuration, including DNS settings.

Many organizations assign a static IPv4 address to most server systems. The servers used throughout this book, for example, make use of static IP addresses. For client hosts and some servers, an alternative to assigning a static IP address is to use the **Dynamic Host Control Protocol** (**DHCP**). DHCP is a network management protocol that enables a workstation to lease an IP address (and release it when the lease expires). You set up a DHCP server to issue IP address configuration to clients using the *Installing DHCP* recipe.

Once you have installed your DHCP server, you can use the *Configuring DHCP scopes and options* recipe to set up the details that your DHCP server is to hand out to clients. In the *Configuring DHCP failover and load balancing* recipe, we deploy a second DHCP server and configure it to act as a failover/load balancing DHCP service.

In this chapter's final recipe, *Configuring DNS zones and resource records*, you configure the DNS server on `DC1` with zones and additional resource records. Before you can administer your Windows Server 2022 infrastructure, you need to create an environment to use PowerShell to carry out the administration.

Configuring IP addressing

By default, Windows uses DHCP to configure all NICs that the Windows installation process discovers when installing Windows. Once you complete the Windows installation, you can use the control panel applet (`ncpa.cpl`), the network shell console application (`netsh.exe`), or, of course, PowerShell to set IP configuration manually. In this recipe, you set the IP address details for `SRV2` and ensure the host registers DNS names in the Reskit.Org DNS domain (on the DNS service running on `DC1`).

Setting up any host requires setting an IP address, a subnet mask, and a default gateway, which you do in the first part of this recipe. Then, you configure `SRV2` (a workgroup host) to register with the DNS server on `DC1.Reskit.Org`. This approach raises some challenges. By default, when you create `DC1.Reskit.Org` as a DC, the domain promotion process sets the domain's DNS zone to require secure updates. That means a workgroup host cannot register. You can overcome this by setting the zone to allow all updates. But this could be dangerous as it allows ANY host to, potentially, register their address. A second challenge is that since `SRV2` is not a domain member, remoting to `DC1` fails. A solution to that issue is to set the WinRM service to trust all hosts. Configuring WinRM to disregard server authentication has security implications you should consider before using this approach in production.

Getting ready

This recipe uses SRV2, a recently added workgroup host. As with all hosts used in this book, you install SRV2 using the Reskit installation scripts you can find on GitHub at `https://github.com/doctordns/ReskitBuildScripts`. You should also have loaded PowerShell 7 and VS Code, as you did in *Chapter 1*, *Installing and Configuring PowerShell 7.1*. To simplify this setup, you can run the two setup scripts in `https://github.com/PacktPublishing/Windows-Server-Automation-with-PowerShell-7.1-Cookbook-Fourth-Edition/tree/main/goodies`. Run the first script in the Windows PowerShell ISE and the second in an elevated VS Code instance.

Note that using Azure for these VMs is not supported.

By default, this host is a DHCP client.

How to do it...

1. Discovering the network adapter, adapter interface, and adapter interface index

```
$IPType    = 'IPv4'
$Adapter   = Get-NetAdapter |  Where-Object Status -eq 'Up'
$Interface = $Adapter | Get-NetIPInterface -AddressFamily $IPType
$Index     = $Interface.IfIndex
Get-NetIPAddress -InterfaceIndex $Index -AddressFamily $IPType |
  Format-Table -Property Interface*, IPAddress, PrefixLength
```

2. Setting a new IP address for the NIC

```
$IPHT = @{
  InterfaceIndex = $Index
  PrefixLength   = 24
  IPAddress      = '10.10.10.51'
  DefaultGateway = '10.10.10.254'
  AddressFamily  = $IPType
}
New-NetIPAddress @IPHT
```

3. Verifying the new IP address

```
Get-NetIPAddress -InterfaceIndex $Index -AddressFamily $IPType |
  Format-Table IPAddress, InterfaceIndex, PrefixLength
```

4. Setting DNS server IP address

```
$CAHT = @{
  InterfaceIndex  = $Index
  ServerAddresses = '10.10.10.10'
}
Set-DnsClientServerAddress @CAHT
```

5. Verifying the new IP configuration

```
Get-NetIPAddress -InterfaceIndex $Index -AddressFamily $IPType |
  Format-Table
```

6. Testing that SRV2 can see the domain controller

```
Test-NetConnection -ComputerName DC1.Reskit.Org |
  Format-Table
```

7. Creating a credential for DC1

```
$U    = 'Reskit\Administrator'
$PPT  = 'Pa$$w0rd'
$PSC  = ConvertTo-SecureString -String $ppt -AsPlainText -Force
$Cred = [pscredential]::new($U,$PSC)
```

8. Setting WinRM on SRV2 to trust DC1

```
$TPPATH = 'WSMan:\localhost\Client\TrustedHosts'
Set-Item -Path $TPPATH -Value 'DC1' -Force
Restart-Service -Name WinRM -Force
```

9. Enabling non-secure updates to Reskit.Org DNS domain

```
$DNSSSB = {
  $SBHT = @{
    Name          = 'Reskit.Org'
    DynamicUpdate = 'NonsecureAndSecure'
  }
  Set-DnsServerPrimaryZone @SBHT
}
Invoke-Command -ComputerName DC1 -ScriptBlock $DNSSSB -Credential $Cred
```

10. Ensuring SRV2 registers within the Reskit.Org DNS zone

```
$DNSCHT = @{
  InterfaceIndex                = $Index
  ConnectionSpecificSuffix      = 'Reskit.Org'
  RegisterThisConnectionsAddress = $true
  UseSuffixWhenRegistering      = $true
}
Set-DnsClient  @DNSCHT
```

11. Registering host IP address at DC1

```
Register-DnsClient
```

12. Pre-staging SRV2 in AD

```
$SB = {New-ADComputer -Name SRV2}
Invoke-Command -ComputerName DC1 -ScriptBlock $SB
```

13. Testing the DNS server on `DC1.Reskit.Org` resolves `SRV2`

```
Resolve-DnsName -Name SRV2.Reskit.Org -Type 'A' -Server DC1.Reskit.Org
```

How it works...

In *step 1*, you use `Get-NetAdapter` and `Get-NetIPAddress` to determine the IP address of this server. Then, you display the resultant address, which looks like this:

```
PS C:\Foo> # 1. Discovering the adapter, adapter interface and adapter interface index
PS C:\Foo> $IPType    = 'IPv4'
PS C:\Foo> $Adapter   = Get-NetAdapter |  Where-Object Status -eq 'Up'
PS C:\Foo> $Interface = $Adapter | Get-NetIPInterface -AddressFamily $IPType
PS C:\Foo> $Index     = $Interface.IfIndex
PS C:\Foo> Get-NetIPAddress -InterfaceIndex $Index -AddressFamily $IPType |
             Format-Table -Property Interface*, IPAddress, PrefixLength

InterfaceAlias InterfaceIndex IPAddress       PrefixLength
-------------- -------------- ---------       ------------
Ethernet                   6 169.254.140.135           16
```

Figure 7.1: Discovering the adapter, adapter interface, and adapter interface index

In *step 2*, you use the `New-NetIPAddress` cmdlet to set a static IP address on `SRV2`. The output looks like this:

```
PS C:\Foo> # 2. Setting a new IP address for the NIC
PS C:\Foo> $IPHT = @{
             InterfaceIndex = $Index
             PrefixLength   = 24
             IPAddress      = '10.10.10.51'
             DefaultGateway = '10.10.10.254'
             AddressFamily  = $IPType
           }
PS C:\Foo> New-NetIPAddress @IPHT

IPAddress         : 10.10.10.51
InterfaceIndex    : 6
InterfaceAlias    : Ethernet
AddressFamily     : IPv4
Type              : Unicast
PrefixLength      : 24
PrefixOrigin      : Manual
SuffixOrigin      : Manual
AddressState      : Tentative
ValidLifetime     : Infinite ([TimeSpan]::MaxValue)
PreferredLifetime : Infinite ([TimeSpan]::MaxValue)
SkipAsSource      : False
PolicyStore       : ActiveStore
```

Figure 7.2: Setting a new IP address for the NIC

To double check that you have configured SRV2 with the correct IP address configuration, you can use the `Get-NetIPaddress` cmdlet. The output looks like this:

```
PS C:\Foo> # 3. Verifying the new IP address
PS C:\Foo> Get-NetIPAddress -InterfaceIndex $Index -AddressFamily $IPType |
             Format-Table IPAddress, InterfaceIndex, PrefixLength

IPAddress    InterfaceIndex PrefixLength
---------    -------------- ------------
10.10.10.51               6           24
```

Figure 7.3: Verifying the new IP address

Besides setting an IP address, subnet mask, and default gateway, you also need to configure SRV2 with a DNS server address. In *step 4*, you use the `Set-DnsClientServerAddress` cmdlet, which creates no output.

To check the updated IP configuration on SRV2, in *step 5*, you verify the configuration by (re) using the `Get-Get-NetIPAddress` cmdlet, with output like this:

```
PS C:\Foo> # 5. Verifying the new IP configuration
PS C:\Foo> Get-NetIPAddress -InterfaceIndex $Index -AddressFamily $IPType |
             Format-Table

ifIndex IPAddress    PrefixLength PrefixOrigin SuffixOrigin AddressState PolicyStore
------- ---------    ------------ ------------ ------------ ------------ -----------
6       10.10.10.52            24 Manual       Manual       Preferred    ActiveStore
```

Figure 7.4: Verifying the new IP configuration

In *step 6*, you use `Test-interconnection` to ensure SRV2 can connect to DC1, the domain controller in the Reskit.Org domain, with this as the output:

```
PS C:\Foo> # 6. Testing that SRV2 can see the domain controller
PS C:\Foo> Test-NetConnection -ComputerName DC1.Reskit.Org |
             Format-Table

ComputerName   RemotePort RemoteAddress PingSucceeded PingReplyDetails (RTT) TcpTestSucceeded
------------   ---------- ------------- ------------- ---------------------- ----------------
DC1.Reskit.Org 0          10.10.10.10   True          0 ms                   False
```

Figure 7.5: Testing that SRV2 can see the domain controller

To enable SRV2, a workgroup computer to run commands on DC1, you need the correct Windows credentials. In *step 7*, which creates no output, you create credentials for the Administrator user in `Reskit.Org`.

With *step 8*, you configure the WinRM service to trust DC1 explicitly. This step creates no output.

In *step 9*, you reconfigure the DNS service on `DC1` to enable non-secure updates to the Reskit.Org domain. In *step 10*, you configure `SRV2` to register itself within the `Reskit.Org` zone on `DC1`. And then, in *step 11*, you register SRV2's IP address within the DNS service on `DC1`. These three steps also produce no output.

In the next step, *step 12*, you pre-stage the `SRV2` computer into the Reskit.Org AD. This step, which creates no output at the console, creates the `SRV2` computer account within the AD. This means a low privilege user can now add `SRV2` to the domain. Also, note that you run this command remotely on `DC1` – this is because you have not yet added the AD tools to `SRV2`.

In the final step, *step 13*, you query the DNS service to resolve the domain name, `SRV2.Reskit.Org`. This step produces the following output:

```
PS C:\Foo> # 13. Testing the DNS server on DC1.Reskit.Org correctly resolves SRV2
PS C:\Foo> Resolve-DnsName -Name SRV2.Reskit.Org -Type 'A' -Server DC1.Reskit.Org

Name              Type  TTL   Section   IPAddress
----              ----  ---   -------   ---------
SRV2.Reskit.Org   A     1200  Answer    10.10.10.51
```

Figure 7.6: Testing the DNS server on DC1.Reskit.Org resolves SRV2

In this recipe, you configured a workgroup server to have a static IP address. You also configured the DNS service to enable SRV2 to register a DNS record within the `Reskit.Org` domain. In most production scenarios, you would join `SRV2` to the domain, in which case DNS registration just works without needing *step 7* through *step 11*.

There's more...

In *step 5*, you verify SRV2's IP address. This test does not check SRV2's DNS configuration. To check the DNS address as well, you could use `Get-NetIPConfiguration`.

In *step 7*, you create a credential to enable you to run commands on `DC1`. In this step, you use the Enterprise/Domain Administrator account. In production, a better approach would be to create another user with a subset of the Enterprise Admin's group's permissions, then use that user to perform *step 9*.

In *step 8*, you configure WinRM to trust the DNS server, `DC1`. This configuration is necessary for a workgroup environment because, by default, workgroup computers do not trust other servers when using PowerShell remoting. PowerShell remoting, by default, performs mutual authentication. Kerberos provides mutual authentication in a domain environment, while in a workgroup environment, you could use SSL to connect to `DC1`. By configuring `SRV2` to trust `DC1`, you are disabling the authentication of `DC1` by `SRV2`. In a protected environment, like where you have your set of Reskit.Org servers, this is acceptable. In production and especially in larger environments, a better approach is to enable SSL for remoting to hosts in separate security realms.

Testing network connectivity

In today's connected world, network connectivity is vital. When you add a new server to your infrastructure, it is useful to ensure that the server can connect to and use the network.

In this recipe, you perform necessary network connectivity tests on the newly installed SRV2 host. You should ensure that full connectivity exists before adding a server to the domain.

Getting ready

This recipe uses SRV2, a workgroup host. You gave this host a static IP address in the *Configuring IP addressing* recipe.

How to do it...

1. Verifying SRV2 itself is up and that Loopback is working

```
Test-Connection -ComputerName SRV2 -Count 1 -IPv4
```

2. Testing connection to local host's WinRM port

```
Test-NetConnection -ComputerName SRV2 -CommonTCPPort WinRM
```

3. Testing basic connectivity to DC1

```
Test-Connection -ComputerName DC1.Reskit.Org -Count 1
```

4. Checking connectivity to SMB port on DC1

```
Test-NetConnection -ComputerName DC1.Reskit.Org -CommonTCPPort SMB
```

5. Checking connectivity to the LDAP port on DC1

```
Test-NetConnection -ComputerName DC1.Reskit.Org -Port 389
```

6. Examining the path to a remote server on the Internet

```
$NCHT = @{
  ComputerName     = 'WWW.Packt.Com'
  TraceRoute       = $true
  InformationLevel = 'Detailed'
}
Test-NetConnection @NCHT    # Check our wonderful publisher
```

How it works...

In *step 1*, you verify that SRV2's Loopback adapter works and that the basic TCP/IP stack is up and working. The output looks like this:

```
PS C:\Foo> # 1. Verifying SRV2 itself is up, and that loopback is working
PS C:\Foo> Test-Connection -ComputerName SRV2 -Count 1 -IPv4

   Destination: SRV2

Ping Source   Address      Latency BufferSize Status
                              (ms)        (B)
---- ------   -------      ------- ---------- ------
   1 SRV2     10.10.10.51        0         32 Success
```

Figure 7.7: Verifying SRV2 itself is up and that Loopback is working

In *step 2*, you check that the WinRM port is open and working, with output like this:

```
PS C:\Foo> # 2. Testing connection to local host's WinRM port
PS C:\Foo> Test-NetConnection -ComputerName SRV2 -CommonTCPPort WinRM

ComputerName     : SRV2
RemoteAddress    : fe80::3814:81da:aecf:8c87%6
RemotePort       : 5985
InterfaceAlias   : Ethernet
SourceAddress    : fe80::3814:81da:aecf:8c87%6
TcpTestSucceeded : True
```

Figure 7.8: Testing the connection to the local host's WinRM port

In the `Reskit.Org` network, `DC1` is a domain controller and a DNS server. In *step 3*, you test the connectivity to this critical enterprise server, with output like this:

```
PS C:\Foo> # 3. Testing basic connectivity to DC1
PS C:\Foo> Test-Connection -ComputerName DC1.Reskit.Org -Count 1

   Destination: DC1.Reskit.Org

Ping Source    Address       Latency BufferSize Status
                                (ms)        (B)
---- ------    -------       ------- ---------- ------
   1 SRV2      10.10.10.10         0         32 Success
```

Figure 7.9: Testing basic connectivity to DC1

In any domain environment, hosts need to access the SYSVOL share on a DC to download group policy `.POL` files. In *step 4*, you test that `SRV2` can access the SMB port, port `445`, on the DC, with output like this:

```
PS C:\Foo> # 4. Checking connectivity to SMB port on DC1
PS C:\Foo> Test-NetConnection -ComputerName DC1.Reskit.Org -CommonTCPPort SMB

ComputerName     : DC1.Reskit.Org
RemoteAddress    : 10.10.10.10
RemotePort       : 445
InterfaceAlias   : Ethernet
SourceAddress    : 10.10.10.51
TcpTestSucceeded : True
```

Figure 7.10: Checking connectivity to the SMB port on DC1

In *step 5*, you test that `SRV2` can access `DC1` on the LDAP port, port `389`, with the following output:

```
PS C:\Foo> # 5. Checking connectivity to the LDAP port on DC1
PS C:\Foo> Test-NetConnection -ComputerName DC1.Reskit.Org -Port 389

ComputerName     : DC1.Reskit.Org
RemoteAddress    : 10.10.10.10
RemotePort       : 389
InterfaceAlias   : Ethernet
SourceAddress    : 10.10.10.51
TcpTestSucceeded : True
```

Figure 7.11: Checking connectivity to the LDAP port on DC1

In *step 6*, you check connectivity to the Internet and test the network path to the publisher's website at `www.packt.com`. The output is:

```
PS C:\Foo> # 6. Examining path to a remote server on the Internet
PS C:\Foo> $NCHT = @{
               ComputerName     = 'WWW.Packt.Com'
               TraceRoute       = $true
               InformationLevel = 'Detailed'
           }
PS C:\Foo> Test-NetConnection @NCHT    # Check our wonderful publisher

ComputerName            : WWW.Packt.Com
RemoteAddress           : 172.67.10.110
NameResolutionResults   : 172.67.10.110
                          104.22.66.180
                          104.22.67.180
InterfaceAlias          : Ethernet 2
SourceAddress           : 10.10.10.51
NetRoute (NextHop)      : 10.10.10.254
PingSucceeded           : True
PingReplyDetails (RTT)  : 7 ms
TraceRoute              : 10.10.10.254
                          51.148.42.43
                          51.148.42.206
                          51.148.42.195
                          195.66.225.179
                          172.67.10.110
```

Figure 7.12: Examining the path to a remote server on the Internet

There's more...

This recipe's steps confirm that the host can accept connections over WinRM and can contact the DC for core activities. You could add several additional tests, such as testing you can access the DNS server and resolve DNS queries.

In *step 6*, you test the Internet connectivity to our publisher's website (`www.packt.com`). Since we are just testing the connectivity to this site, you only need to specify the actual computer name and do not need the HTTP or HTTPS prefix.

Installing DHCP

In previous recipes, you configured `SRV2` with a static IP address and tested its connectivity. Each server needs a unique IP address and other configuration options to configure on a server-by-server basis. You can also configure client computers running Windows 10 or other OSes manually, although this can be a huge and challenging task in large organizations.

Dynamic Host Configuration Protocol (**DHCP**) enables a DHCP client to get its IP configuration and other networking details automagically from a DHCP server. DHCP automates IP configuration and avoids the work and the inevitable issues involved with manual IP configuration.

Windows and most other client operating systems, including Linux and Apple Macs, have a built-in DHCP client. Windows Server also includes a DHCP server service you can install to provide DHCP services to the clients. You can install DHCP using Server Manager and configure the service using the DHCP GUI application. Alternatively, you can automate the installation of DHCP, as you can see in this recipe. In the next recipe, *Configuring DHCP scopes and options*, you configure the DHCP service to issue IP addresses in a specific range. You also configure DHCP to provide DHCP clients with other IP address configuration options, such as the subnet mask, default gateway, and the DNS server IP address or addresses.

Getting ready

This recipe uses `DC1`, a domain controller in the Reskit.Org domain. You should have installed AD on this host and configured it as per earlier recipes in *Chapter 5*, *Exploring .NET* and *Chapter 6*, *Managing Active Directory*.

How to do it...

1. Installing the DHCP feature on `DC1` and adding the management tools

```
Import-Module -Name ServerManager -WarningAction SilentlyContinue
Install-WindowsFeature -Name DHCP -IncludeManagementTools
```

2. Adding DC1 to trusted DHCP servers and adding the DHCP security group

```
Import-Module -Name DHCPServer -WarningAction SilentlyContinue
Add-DhcpServerInDC
Add-DHCPServerSecurityGroup
```

3. Letting DHCP know it's all configured

```
$DHCPHT = @{
  Path  = 'HKLM:\SOFTWARE\Microsoft\ServerManager\Roles\12'
  Name  = 'ConfigurationState'
  Value = 2
}
Set-ItemProperty @DHCPHT
```

4. Restarting the DHCP server

```
Restart-Service -Name DHCPServer -Force
```

5. Testing service availability

```
Get-Service -Name DHCPServer |
  Format-List -Property *
```

How it works...

In *step 1*, you import the `ServerManager` module and use `Install-WindowsFeature` to add the DHCP server service to `DC1`. The output from this step looks like this:

```
PS C:\Foo> # 1. Installing the DHCP feature on DC1 and adding the management tools
PS C:\Foo> Import-Module -Name ServerManager -WarningAction SilentlyContinue
PS C:\Foo> Install-WindowsFeature -Name DHCP -IncludeManagementTools

Success Restart Needed Exit Code   Feature Result
------- -------------- ---------   --------------
True    No             Success     {DHCP Server, DHCP Server Tools}
```

Figure 7.13: Installing the DHCP feature on DC1 and adding the management tools

In *step 2*, you add `DC1` to the set of authorized DHCP servers in the domain and add the DHCP security groups to the DHCP server, which produces no output to the console. The groups that this command adds are the `DHCP Users` and `DHCP Administrators` security groups. For more details on these groups, see `https://secureidentity.se/delegate-dhcp-admins-in-the-domain/`.

In *step 3*, you set a registry entry to tell Windows that all post-deployment DHCP configuration activities are complete. The GUI installation process takes you through this automatically. When installing via PowerShell, you need to set the registry entry to complete the configuration.

When you have completed the configuration activities, you restart the DHCP service. Once restarted, the DHCP service can issue IP configuration to DHCP clients. For this to happen, you must also have specified the configuration information you specify in the *Configuring DHCP scopes and options* recipe. *Step 2*, *step 3*, and *step 4* produce no output.

In *step 5*, you complete this recipe by ensuring that the DHCP service has started. The output of this step looks like this:

```
PS C:\Foo> # 5. Testing service availability
PS C:\Foo> Get-Service -Name DHCPServer |
             Format-List -Property *

UserName             : NT AUTHORITY\NetworkService
Description          : Performs TCP/IP configuration for DHCP clients, including dynamic assignments of IP
                       addresses, specification of the WINS and DNS servers, and connection-specific DNS names.
                       If this service is stopped, the DHCP server will not perform TCP/IP configuration for
                       clients. If this service is disabled, any services that explicitly depend on it will fail
                       to start.
DelayedAutoStart     : False
BinaryPathName       : C:\Windows\system32\svchost.exe -k DHCPServer -p
StartupType          : Automatic
Name                 : DHCPServer
RequiredServices     : {RpcSs, Tcpip, SamSs, EventLog, EventSystem}
CanPauseAndContinue  : True
CanShutdown          : True
CanStop              : True
DisplayName          : DHCP Server
DependentServices    : {}
MachineName          : .
ServiceName          : DHCPServer
ServicesDependedOn   : {RpcSs, Tcpip, SamSs, EventLog, EventSystem}
StartType            : Automatic
ServiceHandle        : Microsoft.Win32.SafeHandles.SafeServiceHandle
Status               : Running
ServiceType          : Win32OwnProcess, Win32ShareProcess
Site                 :
Container            :
```

Figure 7.14: Testing service availability

There's more...

When the Windows DHCP service starts, it checks to ensure the server is on the DHCP server list authorized in the domain. The DHCP service does not start on any non-authorized DHCP server. Adding `DC1` to the list of authorized servers can help to guard against rogue DHCP servers.

In *step 5*, you check the DHCP service. The output from `Get-Service` includes a description of the service and the path name to the service executable. The DHCP service does not run in its own process. Instead, it runs inside `svchost.exe`. It is for this reason that you do not see the service explicitly when you use `Get-Process`.

Configuring DHCP scopes and options

Installing DHCP is simple, as you see in the *Installing DHCP* recipe. You add the Windows feature and then carry out two small configuration steps. In most cases, you probably do not need to take these extra steps. The extra steps enable you to use the relevant security groups and avoid the Server Manager GUI message stating that there are configuration steps that have not been performed yet.

Before your DHCP server can provide IP address configuration information to DHCP clients, you need to create a DHCP scope and DHCP options. A DHCP scope is a range of DHCP addresses that your DHCP server can give out for a given IP subnet. DHCP options are specific configuration options your DHCP server provides, such as the DNS server's IP address and the IPv4 default gateway.

You can set DHCP options at a scope level or a server level, depending on your organization's needs. For example, you would most likely specify a default gateway in the scope options, with DNS server address(es) set at the server level.

In this recipe, you create a new scope for the `10.10.10.0/24` subnet and specify both scope- and server-level options.

Getting ready

You run this recipe on `DC1`, a domain controller in the Reskit.Org domain, after installing the DHCP server service. You must have installed PowerShell 7 and VS Code on this host.

How to do it...

1. Importing the DHCP server module

```
Import-Module DHCPServer -WarningAction SilentlyContinue
```

2. Creating an IPv4 scope

```
$SCOPEHT = @{
  Name         = 'ReskitOrg'
  StartRange   = '10.10.10.150'
  EndRange     = '10.10.10.199'
  SubnetMask   = '255.255.255.0'
  ComputerName = 'DC1.Reskit.Org'
}
Add-DhcpServerV4Scope @SCOPEHT
```

3. Getting IPV4 scopes from the server

```
Get-DhcpServerv4Scope -ComputerName DC1.Reskit.Org
```

4. Setting server-wide option values

```
$OPTION1HT = @{
  ComputerName = 'DC1.Reskit.Org' # DHCP Server to Configure
  DnsDomain    = 'Reskit.Org'     # Client DNS Domain
  DnsServer    = '10.10.10.10'    # Client DNS Server
}
Set-DhcpServerV4OptionValue @OPTION1HT
```

5. Setting a scope-specific option

```
$OPTION2HT = @{
  ComputerName = 'DC1.Reskit.Org' # DHCP Server to Configure
  Router       = '10.10.10.254'
  ScopeID      = '10.10.10.0'
}
Set-DhcpServerV4OptionValue @OPTION2HT
```

6. Viewing DHCP server options

```
Get-DhcpServerv4OptionValue | Format-Table -AutoSize
```

7. Viewing scope-specific options

```
Get-DhcpServerv4OptionValue -ScopeId '10.10.10.10' |
  Format-Table -AutoSize
```

8. Viewing DHCPv4 option definitions

```
Get-DhcpServerv4OptionDefinition | Format-Table -AutoSize
```

How it works...

In *step 1*, you import the `DHCPServer` module. When you installed DHCP (in the *Installing DHCP* recipe), you added the management tools, including this module. However, the DHCP team has not yet made this module compatible with PowerShell 7. This step, which produces no output, loads the module using the Windows PowerShell Compatibility solution. You read about the Windows PowerShell Compatibility solution in *Chapter 3*, *Exploring Compatibility with Windows PowerShell*.

In *step 2*, you create a new DHCP scope for IPV4 addresses. The scope enables the DHCP server to issue IP addresses in the `10.10.10.150` - `10.10.10.199` range. This step produces no output.

In *step 3*, you use `Get-DHCPServerIPV4Scope` to retrieve details of all the DHCP scopes you have defined on `DC1`. The output of this step looks like this:

```
PS C:\Foo> # 3. Getting IPV4 scopes from the server
PS C:\Foo> Get-DhcpServerv4Scope -ComputerName DC1.Reskit.Org

ScopeId     SubnetMask     Name       State   StartRange     EndRange       LeaseDuration
-------     ----------     ----       -----   ----------     --------       -------------
10.10.10.0  255.255.255.0  ReskitOrg  Active  10.10.10.150   10.10.10.199
```

Figure 7.15: Getting IPV4 scopes from the server

In *step 4*, you set two server-wide DHCP options, creating no output. These are options and values offered to all clients of any DHCP scope defined on this server. In *step 5*, you specify a scope option. These two steps produce no output.

In *step 6*, you view the DHCP server-wide options, with output that looks like this:

```
PS C:\Foo> # 6. Viewing server options
PS C:\Foo> Get-DhcpServerv4OptionValue | Format-Table -AutoSize
OptionId Name            Type        Value         VendorClass UserClass PolicyName
-------- ----            ----        -----         ----------- --------- ----------
15       DNS Domain Name String      {Reskit.Org}
6        DNS Servers     IPv4Address {10.10.10.10}
```

Figure 7.16: Viewing server options

With *step 7*, you view the options you have set on the `10.10.10.10` scope, which looks like this:

```
PS C:\Foo> # 7. Viewing scope-specific options
PS C:\Foo> Get-DhcpServerv4OptionValue -ScopeId '10.10.10.10' |
             Format-Table -AutoSize

OptionId Name   Type        Value          VendorClass UserClass PolicyName
-------- ----   ----        -----          ----------- --------- ----------
51       Lease  DWord       {691200}
3        Router IPv4Address {10.10.10.254}
```

Figure 7.17: Viewing scope-specific options

There are 66 DHCP options you can use to provide option values to DHCP clients. Most of these options are of little use in most cases but provide support for niche and uncommon scenarios. You view the set of options defined by default in *step 8*, which looks like this:

```
PS C:\Foo> # 8. Viewing DHCPv4 option definitions
PS C:\Foo> Get-DhcpServerv4OptionDefinition | Format-Table -AutoSize

Name                                                OptionId Type       VendorClass MultiValued
----                                                -------- ----       ----------- -----------
Classless Static Routes                             121      BinaryData             False
Subnet Mask                                         1        IPv4Address            False
Time Offset                                         2        DWord                  False
Router                                              3        IPv4Address            True
Time Server                                         4        IPv4Address            True
Name Servers                                        5        IPv4Address            True
DNS Servers                                         6        IPv4Address            True
Log Servers                                         7        IPv4Address            True
Cookie Servers                                      8        IPv4Address            True
LPR Servers                                         9        IPv4Address            True
Impress Servers                                     10       IPv4Address            True
Resource Location Servers                           11       IPv4Address            True
Host Name                                           12       String                 False
Boot File Size                                      13       Word                   False
Merit Dump File                                     14       String                 False
DNS Domain Name                                     15       String                 False
Swap Server                                         16       IPv4Address            False
Root Path                                           17       String                 False
Extensions Path                                     18       String                 False
IP Layer Forwarding                                 19       Byte                   False
IP Layer Forwarding                                 19       Byte                   False
Nonlocal Source Routing                             20       Byte                   False
Policy Filter Masks                                 21       IPv4Address            True
Max DG Reassembly Size                              22       Word                   False
Default IP Time-to-live                             23       Byte                   False
Path MTU Aging Timeout                              24       DWord                  False
Path MTU Plateau Table                              25       Word                   True
MTU Option                                          26       Word                   False
All subnets are local                               27       Byte                   False
Broadcast Address                                   28       IPv4Address            False
Perform Mask Discovery                              29       Byte                   False
Mask Supplier Option                                30       Byte                   False
Perform Router Discovery                            31       Byte                   False
Router Solicitation Address                         32       IPv4Address            False
Static Route Option                                 33       IPv4Address            True
Trailer Encapsulation                               34       Byte                   False
ARP Cache Timeout                                   35       DWord                  False
Ethernet Encapsulation                              36       Byte                   False
TCP Default Time-to-live                            37       Byte                   False
Keepalive Interval                                  38       DWord                  False
Keepalive Garbage                                   39       Byte                   False
NIS Domain Name                                     40       String                 False
NIS Servers                                         41       IPv4Address            True
NTP Servers                                         42       IPv4Address            True
Vendor Specific Info                                43       BinaryData             False
WINS/NBNS Servers                                   44       IPv4Address            True
NetBIOS over TCP/IP NBDD                            45       IPv4Address            True
WINS/NBT Node Type                                  46       Byte                   False
NetBIOS Scope ID                                    47       String                 False
X Window System Font                                48       IPv4Address            True
X Window System Display                             49       IPv4Address            True
Lease                                               51       DWord                  False
Renewal (T1) Time Value                             58       DWord                  False
Rebinding (T2) Time Value                           59       DWord                  False
NIS+ Domain Name                                    64       String                 False
NIS+ Servers                                        65       IPv4Address            True
Boot Server Host Name                               66       String                 False
Bootfile Name                                       67       String                 False
Mobile IP Home Agents                               68       IPv4Address            True
Simple Mail Transport Protocol (SMTP) Servers       69       IPv4Address            True
Post Office Protocol (POP3) Servers                 70       IPv4Address            True
Network News Transport Protocol (NNTP) Servers      71       IPv4Address            True
World Wide Web (WWW) Servers                        72       IPv4Address            True
Finger Servers                                      73       IPv4Address            True
Internet Relay Chat (IRC) Servers                   74       IPv4Address            True
StreetTalk Servers                                  75       IPv4Address            True
StreetTalk Directory Assistance (STDA) Servers      76       IPv4Address            True
```

Figure 7.18: Viewing DHCPv4 option definitions

There's more...

In *step 2*, you create a new scope and give it a scope name. However, as you can see in *step 5* and elsewhere, the DHCP cmdlets do not provide a `-DHCPScopeName` parameter. Instead, you specify a `ScopeID`. In general, this is the subnet for the IP addresses in the scope, `10.10.10.0/24`. But even then, as you can see in *step 7*, the cmdlet accepts any IP address in the `10.10.10.0/24` subnet as the subnet ID, including `10.10.10.10` as shown.

Using DHCP

After installing the DHCP service and configuring the scope(s) and option values, your DHCP services can issue IP configuration data to any client. Since the DHCP protocol acts at the IP level, the protocol performs no authentication when any DHCP client uses the protocol to request IP configuration details. That means that any client you attach to the physical subnet can ask for and receive IP confirmation details.

In the *Configuring IP addressing* recipe, you set a static IP address for `SRV2`. In this recipe, you reconfigure this server to obtain a DHCP-based IP address (and the options you set in the *Configuring DHCP scopes and options* recipe).

Getting ready

You run this recipe on `SRV2`, which you've reconfigured to get its address via DHCP. You also need `DC1`, a domain controller for the Reskit.Org domain, and the DHCP server that you set up and configured in earlier recipes in this chapter.

How to do it...

1. Adding DHCP RSAT tools

```
Import-Module -Name ServerManager -WarningAction SilentlyContinue
Install-WindowsFeature -Name RSAT-DHCP
```

2. Importing the DHCP module

```
Import-Module -Name DHCPServer -WarningAction SilentlyContinue
```

3. Viewing the scopes on `DC1`

```
Get-DhcpServerv4Scope -ComputerName DC1
```

4. Getting V4 scope statistics from `DC1`

```
Get-DhcpServerv4ScopeStatistics -ComputerName DC1
```

5. Discovering a free IP address

```
Get-DhcpServerv4FreeIPAddress -ComputerName DC1 -ScopeId 10.10.10.42
```

6. Getting SRV2 NIC configuration

```
$NIC = Get-NetIPConfiguration -InterfaceIndex 6
```

7. Getting IP interface details

```
$NIC |
  Get-NetIPInterface  |
    Where-Object AddressFamily -eq 'IPv4'
```

8. Enabling DHCP on the NIC

```
$NIC |
  Get-NetIPInterface  |
    Where-Object AddressFamily -eq 'IPv4' |
      Set-NetIPInterface -Dhcp Enabled
```

9. Checking IP address assigned

```
Get-NetIPAddress -InterfaceAlias "Ethernet"   |
  Where-Object AddressFamily -eq 'IPv4'
```

10. Getting updated V4 scope statistics from DC1

```
Get-DhcpServerv4ScopeStatistics -ComputerName DC1
```

11. Discovering the next free IP address

```
Get-DhcpServerv4FreeIPAddress -ComputerName DC1 -ScopeId 10.10.10.42
```

12. Checking IPv4 DNS name resolution

```
Resolve-DnsName -Name SRV2.Reskit.Org -Type A
```

How it works...

In *step 1*, you install the RSAT-DHCP feature to add the DHCP server's PowerShell module on SRV1, with output like this:

```
PS C:\Foo> # 1. Adding DHCP RSAT tools
PS C:\Foo> Import-Module -Name ServerManager -WarningAction SilentlyContinue
PS C:\Foo> Install-WindowsFeature -Name RSAT-DHCP

Success Restart Needed Exit Code Feature Result
------- -------------- --------- --------------
True    No             Success   {Remote Server Administration Tools, DHCP Se…
```

Figure 7.19: Adding DHCP RSAT tools

The DHCP server module is not .NET Core compatible. In *step 2*, you explicitly load this module using `Import-Module`, which creates no output.

In *step 3*, you look at the scopes available on `DC1` (the DHCP server you installed in the *Installing DHCP* recipe). The output looks like this:

```
PS C:\Foo> # 3. Viewing the scopes on DC1
PS C:\Foo> Get-DhcpServerv4Scope -ComputerName DC1

ScopeId         SubnetMask      Name            State    StartRange      EndRange        LeaseDuration
-------         ----------      ----            -----    ----------      --------        -------------
10.10.10.0      255.255.255.0   ReskitOrg       Active   10.10.10.150    10.10.10.199
```

Figure 7.20: Viewing the scopes on DC1

In *step 4*, you examine the scope statistics for the DHCP scope you created in the *Configuring DHCP scopes and options* recipe, which produces output like this:

```
PS C:\Foo> # 4. Getting V4 scope statistics from DC1
PS C:\Foo> Get-DhcpServerv4ScopeStatistics -ComputerName DC1

ScopeId     Free   InUse   PercentageInUse  Reserved   Pending   SuperscopeName
-------     ----   -----   ---------------  --------   -------   --------------
10.10.10.0  50     0       0                0          0
```

Figure 7.21: Getting V2 scope statistics from DC1

In *step 5*, you use the `Get-DhcpServerv4FreeIPAddress` cmdlet to find the first available free IP address in the scope. The output resembles this:

```
PS C:\Foo> # 5. Discovering a free IP address
PS C:\Foo> Get-DhcpServerv4FreeIPAddress -ComputerName dc1 -ScopeId 10.10.10.42
10.10.10.150
```

Figure 7.22: Discovering a free IP address in the scope

In *step 6*, you get the NIC details and store them in the `$NIC` variable, producing no output. Note that you specify an `InterfaceIndex` of 6, which should be your VM's NIC.

In *step 7*, you use that variable to get the details of the NIC, with output like this:

```
PS C:\Foo> # 7. Getting IP interface
PS C:\Foo> $NIC |
             Get-NetIPInterface  |
               Where-Object AddressFamily -eq 'IPv4'

ifIndex InterfaceAlias  AddressFamily NlMtu(Bytes) InterfaceMetric Dhcp     ConnectionState PolicyStore
------- --------------  ------------- ------------ --------------- ----     --------------- -----------
6       Ethernet        IPv4                  1500              15 Disabled Connected       ActiveStore
```

Figure 7.23: Getting the IP interface details

In *step 8*, you change the NIC to get configuration details from the DHCP server. This step creates no output. In *step 9*, you view the NIC's IPV4 address; this time, one assigned by DHCP. The output looks like this:

```
PS C:\Foo> # 9. Checking IP address assigned
PS C:\Foo> Get-NetIPAddress -InterfaceAlias "Ethernet"  |
             Where-Object AddressFamily -eq 'IPv4'

IPAddress         : 10.10.10.150
InterfaceIndex    : 6
InterfaceAlias    : Ethernet
AddressFamily     : IPv4
Type              : Unicast
PrefixLength      : 24
PrefixOrigin      : Dhcp
SuffixOrigin      : Dhcp
AddressState      : Preferred
ValidLifetime     : 7.23:59:38
PreferredLifetime : 7.23:59:38
SkipAsSource      : False
PolicyStore       : ActiveStore
```

Figure 7.24: Checking IP address assigned

In *step 10*, you re-examine the scope statistics, with output like this:

```
PS C:\Foo> # 10. Getting updated V4 scope statistics from DC1
PS C:\Foo> Get-DhcpServerv4ScopeStatistics -ComputerName DC1

ScopeId     Free  InUse   PercentageInUse  Reserved   Pending        SuperscopeName
-------     ----  -----   ---------------  --------   -------        --------------
10.10.10.0  49    1       2                0          0
```

Figure 7.25: Getting updated V4 scope statistics from DC1

With *step 11*, you recheck and discover the next free IP address in the DHCP scope, with output like this:

```
PS C:\Foo> # 11. Discovering the next free IP address
PS C:\Foo> Get-DhcpServerv4FreeIPAddress -ComputerName dc1 -ScopeId 10.10.10.42
10.10.10.151
```

Figure 7.26: Discovering the next free IP address

In the final step, you check that SRV2 has registered its new IP address in the DNS server on DC1. The output looks like:

```
PS C:\Foo> # 12. Checking IPv4 DNS name resolution
PS C:\Foo> Resolve-DnsName -Name SRV2.Reskit.Org -Type A

Name              Type   TTL   Section   IPAddress
----              ----   ---   -------   ---------
SRV2.Reskit.Org   A      1200  Question  10.10.10.150
```

Figure 7.27: Checking IPv4 DNS name resolution

There's more...

In this recipe, you use the DHCP server cmdlets to get information from the DHCP server on DC1. These cmdlets show how you can obtain information from the DHCP server, including the next free IP address and statistics on the scope (free/used addresses and more).

In *step 6*, you get the network configuration for the NIC with an `InterfaceIndex` of 6. In some cases, you may find that Windows has assigned a different interface index for the NIC. As an alternative, you could use the `-InterfaceAlias` parameter to specify the name of the interface.

In *step 7*, you get the IP interface details to allow you, in *step 8*, to convert the NIC from a static IP address into a dynamic one, based on DHCP.

In *step 9*, you view the DHCP supplied IP address information for SRV2. If you perform *step 8* and immediately run *step 9*, you may find that the NIC shows an **Automatic Private IP Addressing** (**APIPA**) IP address in the `169.254.0.0/24` subnet. This address is transient. When you change to DHCP (as you did in *step 8*), Windows removes the static address and creates an APIPA address. Once SRV2 contacts the DHCP server and negotiates an address, you see the output shown for *step 9*, as in *Figure 7.24*. Obtaining a lease can take a few seconds, so be patient.

Implementing DHCP failover and load balancing

As shown in the two previous recipes, the installation and configuration of a single on-premises DHCP server is straightforward. However, a single DHCP server represents a single point of failure, which is never a good thing. The solution is always to have a second DHCP server. In earlier versions of Windows, you could do this with two DHCP servers, each with a scope. Typically, you used to split the full set of IP addresses and allow each server to have part of that set. The traditional "wisdom" was to do an 80/20 split – have 80% of the scope supplied by your primary DHCP server and 20% on the backup server.

Independent DHCP servers are an error-prone approach and were never ideal since these independent servers did not coordinate scope details. That 80/20 "rule" was a recommendation for one specific customer scenario (a large firm in the Pacific Northwest) and possibly was not meant to become a best practice.

In Windows Server 2012, Microsoft added a DHCP failover and load balancing mechanism that simplified deploying DHCP in an organization. You can now set up two DHCP servers, define a DHCP scope, and allow both servers to work in tandem.

In this recipe, you install DHCP on a second server, DC2, and then configure and use the failover and load balancing capabilities of Windows Server.

Getting ready

This recipe uses the two DCs you have installed, DC1 and DC2. You should also have installed DHCP on DNS (*Installing DHCP*) and configured a DNS zone (*Configuring DHCP scopes and options*). You run this recipe on DC2.

How to do it...

1. Installing the DHCP server feature on DC2

```
Import-Module -Name ServerManager -WarningAction SilentlyContinue
$FEATUREHT = @{
  Name                   = 'DHCP'
  IncludeManagementTools = $True
}
Install-WindowsFeature @FEATUREHT
```

2. Letting DHCP know it is fully configured

```
$IPHT = @{
  Path  = 'HKLM:\SOFTWARE\Microsoft\ServerManager\Roles\12'
  Name  = 'ConfigurationState'
  Value = 2
}
Set-ItemProperty @IPHT
```

3. Authorizing the DHCP server in AD

```
Import-Module -Name DHCPServer -WarningAction 'SilentlyContinue'
Add-DhcpServerInDC -DnsName DC2.Reskit.Org
```

4. Viewing authorized DHCP servers in the Reskit domain

```
Get-DhcpServerInDC
```

5. Configuring failover and load balancing

```
$FAILOVERHT = @{
  ComputerName       = 'DC1.Reskit.Org'
  PartnerServer      = 'DC2.Reskit.Org'
  Name               = 'DC1-DC2'
  ScopeID            = '10.10.10.0'
  LoadBalancePercent = 60
  SharedSecret       = 'j3RryIsTheB3est!'
  Force              = $true
  Verbose            = $True
}
Invoke-Command -ComputerName DC1.Reskit.org -ScriptBlock {
  Add-DhcpServerv4Failover @Using:FAILOVERHT
}
```

6. Getting active leases in the scope (from both servers!)

```
$DHCPServers = 'DC1.Reskit.Org', 'DC2.Reskit.Org'
$DHCPServers |
  ForEach-Object {
    "Server $_" | Format-Table
    Get-DhcpServerv4Scope -ComputerName $_ | Format-Table
  }
```

7. Viewing DHCP server statistics from both DHCP servers

```
$DHCPServers |
  ForEach-Object {
    "Server $_" | Format-Table
    Get-DhcpServerv4ScopeStatistics -ComputerName $_  | Format-Table
  }
```

How it works...

Using the first three steps in this recipe, you install the DHCP server feature on DC2. These steps duplicate the approach you used in the *Installing DHCP* recipe to install DHCP on DC1.

In *step 1*, you import the Server Manager module and then install the DHCP feature, with output like this:

```
PS C:\Foo> # 1. Installing the DHCP server feature on DC2
PS C:\Foo> Import-Module -Name ServerManager -WarningAction SilentlyContinue
PS C:\Foo> $FEATUREHT = @{
             Name                  = 'DHCP'
             IncludeManagementTools = $True
           }
PS C:\Foo> Install-WindowsFeature @FEATUREHT

Success Restart Needed Exit Code      Feature Result
------- -------------- ---------      --------------
True    No             Success        {DHCP Server, DHCP Server Tools}
```

Figure 7.28: Installing the DHCP server feature on DC2

In *step 2*, you set a registry key to tell Windows that you have completed the DHCP feature installation. Without this step, any time you use the Server Manager GUI on `DC2`, you would see a message indicating that additional configuration is required. This step produces no output.

In *step 3*, you import the DHCP server PowerShell module and add `DC2` to the list of authorized DHCP servers in the Reskit domain. This step produces no output.

In *step 4*, you examine the set of authorized DHCP servers, which produces the following output:

```
PS C:\Foo> # 4. Viewing authorized DHCP servers in the Reskit domain
PS C:\Foo> Get-DhcpServerInDC

IPAddress            DnsName
---------            -------
10.10.10.10          dc1.reskit.org
10.10.10.11          dc2.reskit.org
```

Figure 7.29: Viewing authorized DHCP servers in the Reskit domain

In *step 5*, you configure DHCP load balancing and failover, with output like this:

```
PS C:\Foo> # 5. Configuring fail-over and load balancing
PS C:\Foo> $FAILOVERHT = @{
             ComputerName       = 'DC1.Reskit.Org'
             PartnerServer      = 'DC2.Reskit.Org'
             Name               = 'DC1-DC2'
             ScopeID            = '10.10.10.0'
             LoadBalancePercent = 60
             SharedSecret       = 'j3RryIsTheB3est!'
             Force              = $true
             Verbose            = $True
           }

PS C:\Foo> Invoke-Command -ComputerName DC1.Reskit.Org -ScriptBlock {
             Add-DhcpServerv4Failover @Using:FAILOVERHT
           }
VERBOSE: A new failover relationship will be created between servers DC1.Reskit.Org and DC2.Reskit.Org. The configuration
of the specified scopes on server DC1.Reskit.Org will be replicated to the partner server.
VERBOSE: Add scopes on partner server DC2.Reskit.Org ...............................In progress.
VERBOSE: Update properties for scope 10.10.10.0 (1 of 1) on partner server DC2.Reskit.Org .............In progress.
VERBOSE: Update properties for scope 10.10.10.0 (1 of 1) on partner server DC2.Reskit.Org .............Successful.
VERBOSE: Update delay offer for scope 10.10.10.0 (1 of 1) on partner server DC2.Reskit.Org ............In progress.
VERBOSE: Update delay offer for scope 10.10.10.0 (1 of 1) on partner server DC2.Reskit.Org ............Successful.
VERBOSE: Update NAP properties for scope 10.10.10.0 (1 of 1) on partner server DC2.Reskit.Org .........In progress.
VERBOSE: Update NAP properties for scope 10.10.10.0 (1 of 1) on partner server DC2.Reskit.Org .........Successful.
VERBOSE: Update superscope for scope 10.10.10.0 (1 of 1) on partner server DC2.Reskit.Org .............In progress.
VERBOSE: Update superscope for scope 10.10.10.0 (1 of 1) on partner server DC2.Reskit.Org .............Successful.
VERBOSE: Update IP ranges for scope 10.10.10.0 (1 of 1) on partner server DC2.Reskit.Org ..............In progress.
VERBOSE: Update IP ranges for scope 10.10.10.0 (1 of 1) on partner server DC2.Reskit.Org ..............Successful.
VERBOSE: Update exclusions for scope 10.10.10.0 (1 of 1) on partner server DC2.Reskit.Org .............In progress.
VERBOSE: Update exclusions for scope 10.10.10.0 (1 of 1) on partner server DC2.Reskit.Org .............Successful.
VERBOSE: Update reservations for scope 10.10.10.0 (1 of 1) on partner server DC2.Reskit.Org ...........In progress.
VERBOSE: Update reservations for scope 10.10.10.0 (1 of 1) on partner server DC2.Reskit.Org ...........Successful.
VERBOSE: Update policies for scope 10.10.10.0 (1 of 1) on partner server DC2.Reskit.Org ..............In progress.
VERBOSE: Update policies for scope 10.10.10.0 (1 of 1) on partner server DC2.Reskit.Org ..............Successful.
VERBOSE: Update options for scope 10.10.10.0 (1 of 1) on partner server DC2.Reskit.Org ...............In progress.
VERBOSE: Update options for scope 10.10.10.0 (1 of 1) on partner server DC2.Reskit.Org ...............Successful.
VERBOSE: Add scopes on partner server DC2.Reskit.Org ...............................Successful.
VERBOSE: Disable scopes on partner server DC2.Reskit.Org ...........................In progress.
VERBOSE: Disable scopes on partner server DC2.Reskit.Org ...........................Successful.
VERBOSE: Creation of failover configuration on partner server DC2.Reskit.Org .........In progress.
VERBOSE: Creation of failover configuration on partner server DC2.Reskit.Org .........Successful.
VERBOSE: Creation of failover configuration on host server DC1.Reskit.Org ............In progress.
VERBOSE: Creation of failover configuration on host server DC1.Reskit.Org ............Successful.
VERBOSE: Activate scopes on partner server DC2.Reskit.Org ...........................In progress.
VERBOSE: Activate scopes on partner server DC2.Reskit.Org ...........................Successful.
```

Figure 7.30: Configuring failover and load balancing

With *step 6*, you examine the scopes on both DHCP servers, with the following output:

```
PS C:\Foo> # 6. Getting active leases in the scope (from both servers!)
PS C:\Foo> $DHCPServers = 'DC1.Reskit.Org', 'DC2.Reskit.Org'
PS C:\Foo> $DHCPServers |
             ForEach-Object {
               "Server $_" | Format-Table
               Get-DhcpServerv4Scope -ComputerName $_ | Format-Table
             }

Server DC1.Reskit.Org

ScopeId     SubnetMask     Name      State  StartRange    EndRange      LeaseDuration
-------     ----------     ----      -----  ----------    --------      -------------
10.10.10.0 255.255.255.0 ReskitOrg Active 10.10.10.150 10.10.10.199

Server DC2.Reskit.Org

ScopeId     SubnetMask     Name      State  StartRange    EndRange      LeaseDuration
-------     ----------     ----      -----  ----------    --------      -------------
10.10.10.0 255.255.255.0 ReskitOrg Active 10.10.10.150 10.10.10.199
```

Figure 7.31: Getting active leases in the scope from both servers

In *step 7*, you view the DHCP server statistics from your two DHCP servers, with output like this:

```
PS C:\Foo> # 7. Viewing DHCP server statistics from both DHCP servers
PS C:\Foo> $DHCPServers |
             ForEach-Object {
                "Server $_" | Format-Table
                Get-DhcpServerv4ScopeStatistics -ComputerName $_  | Format-Table
               }

Server DC1.Reskit.Org

ScopeId     Free InUse PercentageInUse Reserved Pending SuperscopeName
-------     ---- ----- --------------- -------- ------- --------------
10.10.10.0 49   1     2               0        0

Server DC2.Reskit.Org

ScopeId     Free InUse PercentageInUse Reserved Pending SuperscopeName
-------     ---- ----- --------------- -------- ------- --------------
10.10.10.0 49   1     2               0        0
```

Figure 7.32: Viewing DHCP statistics from both servers

There's more...

Unlike in the *Installing DHCP* recipe, in this recipe, you do not add the DHCP-related local security groups on `DC2`. If you plan to delegate administrative privileges, you may wish to add those groups, as you did in the earlier recipe.

In *step 5*, you establish a load balancing and failover relationship between the two DHCP servers. By using the `-Verbose` switch for the `Add-DhcpServerV4Failover` cmdlet, you can see precisely what the command is doing, step by step. As you can see in the output in *Figure 7.30*, this command copies full details of all the scopes on `DC1` to `DC2`. You should note that the relationship is on a scope-by-scope basis.

Depending on your needs, you can configure different sorts of relationships between DHCP servers using the `-ServerRole` parameter. For more details on this parameter and others that you can use to fine-tune the relationship between the two partners, see `https://docs.microsoft.com/powershell/module/dhcpserver/add-dhcpserverv4failover`.

The DHCP failover feature provides several additional settings to control precisely when to failover and for how long (and when to failback). These settings allow you to control, for example, what might happen during a planned reboot of your DHCP server.

Deploying DNS in the Enterprise

When you installed Active Directory in *Chapter 6*, *Managing Active Directory*, you added a single DNS server on `DC1`. When you added a replica DC, `DC2`, and when you added the child domain with `UKDC1`, you did not set up any additional DNS servers in your forest. In an enterprise organization, this is not best practice. You always want to configure your clients and servers so that they use at least two DNS servers. For servers with a static DNS setting, you should also update the DHCP DNS server option settings to ensure that your DHCP servers also provide two DNS server entries to DHCP clients.

In most organizations, there are several DNS service configuration options you may wish to set. These include whether to allow DNS server recursion on the server, the maximum size of the DNS cache, and whether to use **Extended DNS** (**EDNS**).

EDNS (also referred to as EDNS0 or, more recently, EDNS(0)) is an extension mechanism that enables more recent DNS servers to interact with older servers that may not be capable of specific actions. For more details on EDNS(0), see `https://en.wikipedia.org/wiki/Extension_Mechanisms_for_DNS`.

This recipe is a starting point. There are other DNS server options you may wish to consider updating, such as DNS security. Additionally, you may need to reconfigure other servers and update the static IP address settings (to point to both DNS servers). And finally, in the `Reskit.Org` forest, you should also be updating the child domain `UK.Reskit.Org`.

In this recipe, you update the domain-wide DNS settings in the Reskit.Org domain. These settings include ensuring that you configure DC2's DNS service in the same way as DC1, updating DNS server configuration settings, configuring both DC1 and DC2 to point to the correct DNS servers, and updating DHCP to issue both DNS server addresses. You then examine the DNS settings.

In the *Configuring IP addressing* recipe, you configured SRV2, a workgroup host, to update DNS. You configured the Reskit.Org zone on DC1 to allow non-domain members to update their DNS registrations using dynamic DNS. This configuration is not a best practice configuration. In the final steps of this recipe, you update the DNS zones for Reskit.Org to allow only secure updates and join SRV2 to the Reskit.Org domain with a static IP address.

Getting ready

You run this recipe on DC2, with DC1 and SRV2 also operational. You run this recipe after setting up DNS on both DC1 and DC2 and implementing DHCP.

How to do it...

1. Installing the DNS feature on DC2

```
Import-Module -Name ServerManager -WarningAction SilentlyContinue
Install-WindowsFeature -Name DNS -IncludeManagementTools
```

2. Creating a script block to set DNS server options

```
$SB1 = {
  # Enable recursion on this server
  Set-DnsServerRecursion -Enable $true
  # Configure DNS Server cache maximum size
  Set-DnsServerCache  -MaxKBSize 20480  # 28 MB
  # Enable EDNS
  $EDNSHT = @{
    EnableProbes    = $true
    EnableReception = $true
  }
  Set-DnsServerEDns @EDNSHT
  # Enable Global Name Zone
  Set-DnsServerGlobalNameZone -Enable $true
}
```

3. Reconfiguring DNS on DC2 and DC1

```
Invoke-Command -ScriptBlock $SB1
Invoke-Command -ScriptBlock $SB1 -ComputerName DC1
```

4. Creating a script block to configure DC2 to have two DNS servers

```
$SB2 = {
  $NIC =
     Get-NetIPInterface -InterfaceAlias "Ethernet" -AddressFamily IPv4
  $DNSSERVERS = ('127.0.0.1','10.10.10.10')
  $DNSHT = @{
    InterfaceIndex  = $NIC.InterfaceIndex
    ServerAddresses = $DNSSERVERS
  }
  Set-DnsClientServerAddress @DNSHT
  Start-Service -Name DNS
}
```

5. Configuring DC2 to have two DNS servers

```
Invoke-Command -ScriptBlock $SB2
```

6. Creating a script block to configure DC1 to have two DNS servers

```
$SB3 = {
  $NIC =
    Get-NetIPInterface -InterfaceAlias "Ethernet" -AddressFamily IPv4
  $DNSSERVERS = ('127.0.0.1','10.10.10.11')
  $DNSHT = @{
    InterfaceIndex  = $NIC.InterfaceIndex
    ServerAddresses = $DNSSERVERS
  }
  Set-DnsClientServerAddress @DNSHT
  Start-Service -Name DNS
}
```

7. Configuring DC1 to have two DNS servers

```
Invoke-Command -ScriptBlock $SB3 -ComputerName DC1
```

8. Updating DHCP scope to add two DNS entries

```
$DNSOPTIONHT = @{
  DnsServer    = '10.10.10.11',
                 '10.10.10.10'    # Client DNS Servers
  DnsDomain    = 'Reskit.Org'
  Force        = $true
}
Set-DhcpServerV4OptionValue @DNSOPTIONHT -ComputerName DC1
Set-DhcpServerV4OptionValue @DNSOPTIONHT -ComputerName DC2
```

9. Getting DNS service details

```
$DNSRV = Get-DNSServer -ComputerName DC2.Reskit.Org
```

10. Viewing recursion settings

```
$DNSRV |
  Select-Object -ExpandProperty ServerRecursion
```

11. Viewing server cache settings

```
$DNSRV |
  Select-Object -ExpandProperty ServerCache
```

12. Viewing EDNS settings

```
$DNSRV |
  Select-Object -ExpandProperty ServerEdns
```

13. Setting Reskit.Org zone to be secure only

```
$DNSSSB = {
  $SBHT = @{
    Name          = 'Reskit.Org'
    DynamicUpdate = 'Secure'
  }
  Set-DnsServerPrimaryZone @SBHT
}
Invoke-Command -ComputerName DC1 -ScriptBlock $DNSSSB
Invoke-Command -ComputerName DC2 -ScriptBlock $DNSSSB
```

Run the next step on SRV2.

14. Adding SRV2 to domain

```
$User  = 'Reskit\Administrator'
$Pass  = 'Pa$$w0rd'
$PSS   = $Pass | ConvertTo-SecureString -Force -AsPlainText
$CRED  = [PSCredential]::new($User,$PSS)
$Sess  = New-PSSession -UseWindowsPowerShell
Invoke-Command -Session $Sess -Scriptblock {
  $ACHT = @{
    Credential = $using:Cred
    Domain     = 'Reskit.org'
    Force      = $True
  }
  Add-Computer @ACHT
  Restart-Computer
} | Out-Null
```

How it works...

In *step 1*, you install the DNS feature on DC1, with output like this:

```
PS C:\Foo> # 1. Installing the DNS feature on DC2
PS C:\Foo> Import-Module -Name ServerManager -WarningAction SilentlyContinue
PS C:\Foo> Install-WindowsFeature -Name DNS -IncludeManagementTools

Success Restart Needed Exit Code      Feature Result
------- -------------- ---------      --------------
True    No             Success        {DNS Server, DNS Server Tools}
```

Figure 7.33: Installing the DNS feature on DC2

In *step 2*, you create a script block that contains the DNS server options. In *step 3*, you run this script block on both DC1 and DC2. In *step 4*, you create a script block to use to set the IP configuration on DC2. In *step 5*, you run this script block to reconfigure DC2. In *step 6* and *step 7*, you reconfigure the static IP setting on DC1. In *step 8*, you update the DHCP server's DHCP options to ensure the DHCP server on both hosts supplies two DNS server IP addresses to clients of either DHCP server. These seven steps produce no output.

In *step 9*, you get the DNS server details from DC2, producing no output. In *step 10*, you examine the DNS server recursion settings, with output like this:

```
PS C:\Foo> # 10. Viewing recursion settings
PS C:\Foo> $DNSRV |
             Select-Object -ExpandProperty ServerRecursion

Enable               : True
AdditionalTimeout(s) : 4
RetryInterval(s)     : 3
Timeout(s)           : 8
SecureResponse       : True
```

Figure 7.34: Viewing DNS server recursion settings

In *step 11*, you examine the DNS server cache settings on DC2, with output like this:

```
PS C:\Foo> # 11. Viewing server cache settings
PS C:\Foo> $DNSRV |
             Select-Object -ExpandProperty ServerCache

MaxTTL                            : 1.00:00:00
MaxNegativeTTL                    : 00:15:00
MaxKBSize                         : 20480
EnablePollutionProtection         : True
LockingPercent                    : 100
StoreEmptyAuthenticationResponse  : True
IgnorePolicies                    : False
```

Figure 7.35: Viewing server cache settings

In *step 12*, you view the EDNS settings on DC2, with output like this:

```
PS C:\Foo> # 12. Viewing EDNS settings
PS C:\Foo> $DNSRV |
             Select-Object -ExpandProperty ServerEdns

CacheTimeout EnableProbes EnableReception
------------ ------------ ---------------
00:15:00     True         True
```

Figure 7.36: Viewing EDNS settings on DC2

In *step 13*, you change the DNS domain for `Reskit.Org` so that it only accepts secure dynamic updates on both DNS servers in the domain.

In the final step of this recipe, *step 14*, you add SRV2 to the `Reskit.Org` domain. This step produces no output.

There's more...

In this recipe, you install and configure DNS on DC2. You also set DNS server options on both DNS servers (DC1 and DC2). In *step 5* and *step 7*, you configure your domain controllers so that they point to themselves first and to the other DNS servers second. Configuring DNS in this manner is a best practice.

In an earlier recipe, you configured your DHCP servers so that they provided a single DNS server IP address to DHCP clients. In *step 8*, you update DHCP to provide two DNS server IP addresses.

You then obtain and view the DNS server recursion, server cache, and EDNS settings on DC2. You could extend these steps by examining the settings on DC2 and comparing them to ensure consistent configuration.

In the *Configuring IP addressing* recipe, you used a workgroup computer, SRV2, and enabled it to register with DNS by setting the domain to accept non-secure updates. Using non-secure DNS updates is not a best practice as it could allow a rogue computer to "steal" a real DNS name. In *step 13*, you ensure that the `Reskit.Org` DNS domain only allows secure updates. Then, in *step 14* and *step 15*, you add SRV2 to the domain. For the remainder of this book, SRV2 is a domain-joined computer and not a workgroup system. These last three steps produce no output.

In this recipe, you set up DNS for use in the Reskit.Org domain. You made no changes to DNS with respect to the UK.Reskit.Org sub-domain. In production, you would want at least a second DC and would need to update your DNS server address settings for servers in the UK domain.

Another production aspect regards replication of domains. In this chapter, you created a DNS zone for `Reskit.Org`. The recipes you use set that domain to be AD integrated and to be replicated to every DC in the `Reskit.Org` forest. This means that all DNS server changes to the `Reskit.Org` domain are replicated automatically to `UKDC1`. Depending on your usage scenario, you may wish to adjust the zone replication scope for AD-integrated zones.

Configuring DNS forwarding

When a DNS server gets a query for a **resource record** (**RR**) not held by the server, it can use recursion to discover a DNS server that can resolve the RR. If, for example, you use `Resolve-DNSName` to resolve `www.packt.com`, the configured DNS server may not hold a zone that would help. Your DNS service then looks to the DNS root servers to discover a DNS server that can via the recursion process. Eventually, the process finds a DNS server that can resolve the RR. Your DNS server then caches these details locally in the DNS server cache.

If you are resolving publicly available names, this process works great. But you might have internally supplied DNS names that the DNS can't resolve via the mechanism. An example might be when two companies merge. There may be internal hostnames (for example, `intranet.kapoho.com` and `intranet.reskit.org`) that your organization's internal DNS servers can resolve but are not available from publicly facing DNS servers. In that scenario, you can set up **conditional forwarding**. Conditional forwarding enables one DNS server to forward a query to another DNS server or set of servers and not use recursion. You can learn a bit more about conditional forwarding here: `https://medium.com/tech-jobs-academy/dns-forwarding-and-conditional-forwarding-f3118bc93984`.

An alternative to using conditional forwarding is to use stub zones. You can learn more about the differences between conditional forwarding and stub zones here: `https://blogs.msmvps.com/acefekay/2018/03/20/what-should-i-use-a-stub-conditional-forwader-forwarder-or-secondary-zone/`.

Getting ready

In this recipe, you use `DC1`, a domain controller and DNS server for Reskit.Org. You have previously promoted `DC1` to be a DC and installed/configured DNS by following the recipes earlier in this chapter and book.

How to do it...

1. Obtaining the IP addresses of DNS servers for `packt.com`

```
$NS = Resolve-DnsName -Name packt.com -Type NS |
  Where-Object Name -eq 'packt.com'
$NS
```

2. Obtaining the IPV4 addresses for these hosts

```
$NSIPS = foreach ($Server in $NS) {
  (Resolve-DnsName -Name $Server.NameHost -Type A).IPAddress
}
$NSIPS
```

3. Adding conditional forwarder on `DC1`

```
$CFHT = @{
  Name          = 'Packt.Com'
  MasterServers = $NSIPS
}
Add-DnsServerConditionalForwarderZone @CFHT
```

4. Checking zone on `DC1`

```
Get-DnsServerZone -Name Packt.Com
```

5. Testing conditional forwarding

```
Resolve-DNSName -Name WWW.Packt.Com -Server DC1 |
 Format-Table
```

How it works...

In *step 1*, you resolve the name servers serving the `packt.com` domain on the Internet, and then you display the results, like this:

```
PS C:\Foo> # 1.Obtaining the IP addresses of DNS servers for packt.com
PS C:\Foo> $NS = Resolve-DnsName -Name packt.com -Type NS |
             Where-Object Name -eq 'packt.com'
PS C:\Foo> $NS

Name       Type  TTL   Section  NameHost`
----       ----  ---   -------  --------
packt.Com  NS    86400 Answer   max.ns.cloudflare.Com
packt.Com  NS    86400 Answer   eva.ns.cloudflare.Com
```

Figure 7.37: Obtaining the IP addresses of the DNS servers for packt.com

In *step 2*, you use the DNS servers you just retrieved and resolve their IPv4 addresses from the hostnames (which you got in *step 1*). This step produces the following output:

```
PS C:\Foo> # 2.Obtaining the IPV4 addresses for these hosts
PS C:\Foo> $NSIPS = foreach ($Server in $NS) {
             (Resolve-DnsName -Name $Server.NameHost -Type A).IPAddress
           }
PS C:\Foo> $NSIPS
173.245.59.132
172.64.33.132
108.162.193.132
172.64.32.114
108.162.192.114
173.245.58.114
```

Figure 7.38: Obtaining the IPv4 addresses for hosts

In *step 3*, which generates no output, you create a DNS forwarding zone for `packt.com`, which you populate with the IP addresses returned in *step 2*.

In *step 4*, you view the conditional forwarder domain defined on `DC1`, with output like this:

```
PS C:\Foo> # 4. Checking zone on DC1
PS C:\Foo> Get-DnsServerZone -Name Packt.Com

ZoneName   ZoneType   IsAutoCreated  IsDsIntegrated  IsReverseLookupZone  IsSigned
--------   --------   -------------  --------------  -------------------  --------
Packt.Com  Forwarder  False          False           False
```

Figure 7.39: Checking the zone on DC1

With the final step, *step 5*, you resolve `www.packt.com`, which returns both IPv4 and IPv6 addresses, like this:

```
PS C:\Foo> # 5. Testing conditional forwarding
PS C:\Foo> Resolve-DNSName -Name WWW.Packt.Com -Server DC1 |
             Format-Table

Name           Type TTL Section IPAddress
----           ---- --- ------- ---------
WWW.Packt.Com AAAA 300 Answer  2606:4700:10::6816:43b4
WWW.Packt.Com AAAA 300 Answer  2606:4700:10::ac43:a6e
WWW.Packt.Com AAAA 300 Answer  2606:4700:10::6816:42b4
WWW.Packt.Com A    300 Answer  104.22.66.180
WWW.Packt.Com A    300 Answer  104.22.67.180
WWW.Packt.Com A    300 Answer  172.67.10.110
```

Figure 7.40: Testing conditional forwarding

There's more...

In *step 1*, you discover the name server names for the DNS servers that serve `packt.com`. In this case, these servers are part of Cloudflare's distributed DNS service. For more information on how this service works, see `https://www.cloudflare.com/dns/`.

In *step 2*, you resolve those DNS server names into IP addresses. In *step 3*, you create a conditional forwarding domain that forwards queries for `packt.com` (such as `www.packt.com`) to one of the six IP addresses you saw in *step 2*.

In *step 4*, you view the conditional forwarding zone on `DC1`. Since the zone is not DS integrated, DNS does not replicate it to `DC2`. In production, you should repeat *step 3* on `DC2`.

In *step 5*, you resolve `www.packt.com` via conditional forwarding. Since the name servers you discovered support IPv6 AAAA address records, this step returns both IPv4 and IPv6 addresses. If you are in an environment that supports IPv6, you should consider modifying *step 2* so that it returns the IPv4 and IPv6 addresses and then use them in *step 3*.

Managing DNS zones and resource records

The DNS service enables you to resolve names to other information. Most DNS usage resolves a hostname to its IP (IPv4 or IPv6) addresses. But there are other resolutions, such as determining email servers or for anti-spam, that also rely on DNS.

DNS servers hold **zones**. A DNS zone is a container for a set of RRs related to a specific DNS domain. When you enter `www.packt.com`, your browser uses DNS to resolve that website name into an IP address and contacts the server at that IP address. If you use an email client to send mail, for example, to `DoctorDNS@Gmail.Com`, the email client uses DNS to discover an email server to which to send the mail.

Before you can use DNS to hold a RR, you must first create a DNS forward lookup zone. A zone is one part of the global (or internal) DNS namespace. You can configure different zones to hold different parts of your namespace. In *Chapter 6*, *Managing Active Directory*, you added a child domain to the `Reskit.Org` forest, `UK.Reskit.Org`. You could, for example, have one zone holding RRs for `Reskit.Org` on `DC1` and `DC2`, while delegating to a new zone, `UK.Reskit.Org`, on `UKDC1`.

A reverse lookup zone is one that does the reverse – you use it to obtain a hostname from its IP address. You may find reverse lookup zones useful, but you do not need them for most domain usage. The old `nslookup.exe` command is one tool that uses the reverse lookup zone, for example. For more details on DNS in Windows, see `https://docs.microsoft.com/windows-server/networking/dns/dns-top`.

Traditionally, DNS stores and obtains zone details from files on the DNS server. When the DNS service starts up, it reads these zone files and updates the files as needed when it receives updates. If you have multiple DNS servers, you need to configure your service to perform replication to ensure all DNS servers are in sync.

With Windows 2000, Microsoft added AD-integrated zones. DNS stores the DNS data for these zone types within AD. These zones replicate their zone data via AD replication, which simplifies the setup. This also means that when you created the AD service on `DC1`, DNS created a zone in the DNS server on `DC1` and replicated the zone data to all DCs in the forest. For more information on AD-integrated zones, see `https://docs.microsoft.com/windows-server/identity/ad-ds/plan/active-directory-integrated-dns-zones`.

In this recipe, you create a DNS forward and reverse look up zone and then create RRs in those zones. Once added, you test DNS resolution.

Getting ready

You run this recipe on `DC1`, but you need both `DC1` and `DC2` online. You have installed AD and DNS on both servers, and by doing so, you created a single forward lookup zone for the `Reskit.Org` forest.

How to do it...

1. Creating a new primary forward DNS zone for `Cookham.Net`

```
$ZHT1 = @{
  Name              = 'Cookham.Net'
  ResponsiblePerson = 'dnsadmin.cookham.net.'
  ReplicationScope  = 'Forest'
  ComputerName      = 'DC1.Reskit.Org'
}
Add-DnsServerPrimaryZone @ZHT1
```

2. Creating a reverse lookup zone

```
$ZHT2 = @{
  NetworkID         = '10.10.10.0/24'
  ResponsiblePerson = 'dnsadmin.reskit.org.'
  ReplicationScope  = 'Forest'
  ComputerName      = 'DC1.Reskit.Org'
}
Add-DnsServerPrimaryZone @ZHT2
```

3. Registering DNS for `DC1`, `DC2`

```
Register-DnsClient
Invoke-Command -ComputerName DC2 -ScriptBlock {Register-DnsClient}
```

4. Checking the DNS zones on DC1

```
Get-DNSServerZone -ComputerName DC1
```

5. Adding resource records to Cookham.Net zone

```
# Adding an A record
$RRHT1 = @{
  ZoneName      =  'Cookham.Net'
  A             =  $true
  Name          = 'Home'
  AllowUpdateAny =  $true
  IPv4Address    = '10.42.42.42'
}
Add-DnsServerResourceRecord @RRHT1

# Adding a Cname record
$RRHT2 = @{
  ZoneName      = 'Cookham.Net'
  Name          = 'MAIL'
  HostNameAlias = 'Home.Cookham.Net'
}
Add-DnsServerResourceRecordCName @RRHT2

# Adding an MX record
$MXHT = @{
  Preference     = 10
  Name           = '.'
  TimeToLive     = '4:00:00'
  MailExchange   = 'Mail.Cookham.Net'
  ZoneName       = 'Cookham.Net'
}
Add-DnsServerResourceRecordMX @MXHT
```

6. Restarting DNS service to ensure replication

```
Restart-Service -Name DNS
$SB = {Restart-Service -Name DNS}
Invoke-Command -ComputerName DC2 -ScriptBlock $SB
```

7. Checking results of RRs in Cookham.Net zone

```
Get-DnsServerResourceRecord -ZoneName 'Cookham.Net'
```

8. Testing DNS resolution on DC2, DC1

```
# Testing The CNAME from DC1
Resolve-DnsName -Server DC1.Reskit.Org -Name 'Mail.Cookham.Net'
# Testing the MX on DC2
Resolve-DnsName -Server DC2.Reskit.Org -Name 'Cookham.Net'
```

9. Testing the reverse lookup zone

```
Resolve-DnsName -Name '10.10.10.10'
```

How it works...

In *step 1*, which you run on `DC1`, you create a new primary forward DNS zone for the DNS domain `Cookham.Net`. In *step 2*, you create a primary reverse lookup zone for `10.10.10.0/24`. In *step 3*, you run `Register-DNSClient` on both `DC1` and `DC2`. This ensures that both DCs have updated their DNS details. These three steps produce no console output.

In *step 4*, you check the DNS zones held by the DNS service on `DC1`. The output looks like this:

```
PS C:\Foo> # 4. Checking the DNS zones on DC1
PS C:\Foo> Get-DNSServerZone -ComputerName DC1

ZoneName                ZoneType   IsAutoCreated  IsDsIntegrated  IsReverseLookupZone  IsSigned
--------                --------   -------------  --------------  -------------------  --------
_msdcs.Reskit.Org       Primary    False          True            False                False
0.in-addr.arpa          Primary    True           False           True                 False
10.10.10.in-addr.arpa   Primary    False          True            True                 False
127.in-addr.arpa        Primary    True           False           True                 False
255.in-addr.arpa        Primary    True           False           True                 False
Cookham.Net             Primary    False          True            False                False
Packt.Com               Forwarder  False          False           False
Reskit.Org              Primary    False          True            False                False
TrustAnchors            Primary    False          True            False                False
```

Figure 7.41: Checking the DNS zones on DC1

In *step 5*, you add three RRs to `Cookham.Net`: an A record, a CNAME record, and an MX record. In *step 6*, you restart the DNS service on both `DC1` and `DC2` to ensure replication has ensured both DNS servers are up to date. These two steps generate no console output.

In *step 7*, you get all the DNS resource records for `Cookham.Net`, which looks like this:

```
PS C:\Foo> # 7. Checking results of RRs in Cookham.Net zone
PS C:\Foo> Get-DnsServerResourceRecord -ZoneName 'Cookham.Net'

HostName  RecordType Type  Timestamp  TimeToLive  RecordData
--------  ---------- ----  ---------  ----------  ----------
@         NS         2     0          01:00:00    dc1.reskit.org.
@         NS         2     0          01:00:00    dc2.reskit.org.
@         SOA        6     0          01:00:00    [5][dc1.reskit.org.][dnsadmin.cookham.net.]
@         MX         15    0          04:00:00    [10][Mail.Cookham.Net.]
Home      A          1     0          01:00:00    10.42.42.42
MAIL      CNAME      5     0          01:00:00    Home.Cookham.Net.
```

Figure 7.42: Checking the DNS RRs for Cookham.Net

In *step 8*, you test the DNS name resolution from `DC1` and `DC2`. You first resolve the `Mail.Cookham.Net` CNAME RR from `DC1`, then check the MX record from `DC2`. The output from these two commands is as follows:

```
PS C:\Foo> # 8. Testing DNS resolution on DC2, DC1
PS C:\Foo> # Testing The Cname
PS C:\Foo> Resolve-DnsName -Server DC1.Reskit.Org -Name 'Mail.Cookham.Net'

Name                           Type   TTL   Section    NameHost
----                           ----   ---   -------    --------
Mail.Cookham.Net               CNAME  3600  Answer     Home.Cookham.Net

Name       : Home.Cookham.Net
QueryType  : A
TTL        : 3600
Section    : Answer
IP4Address : 10.42.42.42

PS C:\Foo> # Testing the MX on DC2
PS C:\Foo> Resolve-DnsName -Server DC2.Reskit.Org -Name 'Cookham.Net'  -Type MX |
             Format-Table

Name         Type TTL  Section NameExchange     Preference
----         ---- ---  ------- ------------     ----------
Cookham.Net MX   3600 Answer  Mail.Cookham.Net 10
```

Figure 7.43: Checking the DNS name resolution from DC1 and DC2

In *step 9*, you test the reverse lookup zone and resolve `10.10.10.10` to a domain name, like this:

```
PS C:\Foo> # 9. Testing the reverse lookup zone
PS C:\Foo> Resolve-DnsName -Name '10.10.10.10'

Name                          Type  TTL   Section   NameHost
----                          ----  ---   -------   --------
10.10.10.10.in-addr.arpa.     PTR   1200  Question  DC1.Reskit.Org
```

Figure 7.44: Testing the reverse lookup zone

There's more...

In *step 5*, you create some new RRs for the `Cookham.Net` zone, which you test in later steps. To ensure that AD and DNS replicate the new RRs from, in this case, `DC1` to `DC2`, in *step 6*, you restart the DNS service on both DCs. Restarting the DNS service ensures replication between the DNS services.

8

Implementing Enterprise Security

In this chapter, we cover the following recipes:

- Implementing Just Enough Administration (JEA)
- Examining Applications and Services Logs
- Discovering logon events in the event log
- Deploying PowerShell group policies
- Using PowerShell Script Block Logging
- Configuring AD password policies
- Managing Windows Defender Antivirus

Introduction

Security within every organization is vital, more so now than ever, with today's near-constant threats from any number of attackers. You need to ensure every aspect of your organization is secure, from physical security to the security of your network and computer infrastructure.

Since the earliest times, security-savvy folks have preached the gospel of security in depth. Having as many layers as is possible and realistic is just a good thing. As the theory goes – the bad guys have to defeat all your layers to defeat you, while you only need to hold one to stay safe.

PowerShell is a powerful tool for IT professionals wanting to be secure and stay secure. There is so much you can do with PowerShell to help your organization deploy excellent security over your network and computer infrastructure. In this chapter, we look at several ways to use PowerShell to improve your Windows infrastructure's security.

Just Enough Administration (**JEA**) is a feature that enables you to implement fine-grained administration, giving users just enough power to enable them to do their job and no more. A core objective of JEA is to reduce the number of users who are members of very high-privilege groups, including the local Administrators, Domain Admins, and Enterprise Admins groups.

In Windows, just about any component that does anything logs information to Windows event logs. These include the classic logs (first implemented with Windows NT 3.1), plus the Application and Services Logs Microsoft added to Windows Vista. They provide a massive amount of information to help you manage your systems. One particular event that can be of interest is logon events – who logged on and when. You can use this information to track unusual or suspicious logons.

You can manage certain aspects of PowerShell 7, like Windows PowerShell, using Group Policy or by manually setting policy registry keys. With attackers increasingly using file-less PowerShell attacks, script block logging is one way of detecting suspicious behavior. You can use these event log entries for active detection by deploying a **Security Information and Event Management** (**SIEM**) tool, such as SolarWinds Security Event Manager or RSA NetWitness. Or you can store the events for manual review.

A critical security consideration for any size organization is your password policy. You have a considerable amount of flexibility over your Windows password policies. Windows 10 and Windows Server 2022 have a default password policy that you can change. You can set a default domain password policy if you want longer or shorter passwords or complex or non-complex passwords. For those cases where you wish to have a different password policy for specific users, you can use AD's fine-grained password feature, which enables you to set a password policy for an **organizational unit** (**OU**).

Windows Server 2022 and Windows 10 come with a built-in antivirus and antimalware product, Microsoft Defender Antivirus, or MDA. This was formerly just Microsoft Defender. MDA is part of a more extensive suite of products under the umbrella name of Microsoft Defender for Endpoint. See `https://www.microsoft.com/microsoft-365/security/endpoint-defender` for more information. Windows 10 and Windows Server come with a Defender module to help you manage Defender on a server.

Implementing Just Enough Administration (JEA)

Just Enough Administration, also known as JEA, is a security framework providing you with the ability to implement fine-grained administrative delegation. With JEA, you enable a user to have just enough administrative power to do their job, and no more. JEA is a more secure alternative to just adding users to the Domain Administrator or Enterprise Administrator groups.

With JEA, you could, for example, give a junior administrator the rights to access your **domain controllers** (**DCs**) to administer the DNS service on the DC. JEA allows you to constrain what the user can do on the protected server. For example, you could allow the user to stop and start the DNS service (using `Stop-Service` and `Start-Service`) but no other services.

JEA makes use of three objects:

- **JEA role capabilities file** (**.psrc**): This file defines a role in terms of its capabilities. You would configure the JEA role `RKDnsAdmins` to define a limited set of cmdlets that the role has access to on the domain controller, namely those related to administering DNS on a DC.
- **JEA session configuration file** (**.pssc**): This file defines who can access a PowerShell remoting session and what they can do within the session. You could allow anyone in the `RKDnsAdmins` domain security group to access the server using a JEA endpoint. The session configuration file defines the JEA session's actions by reference to the role capabilities file. A JEA-protected remoting session can only be used by certain people who can do whatever the role capabilities file dictates.
- **A PowerShell remoting endpoint**: Once you have the role capabilities and session configuration files created, you register the JEA endpoint to the server you are protecting with JEA.

Once the JEA endpoint is registered, a user who is a member of the domain security group, `RKDnsAdmins`, can use `Invoke-Command` or `Enter-PSSession`, specifying the remote server and the JEA-protected endpoint to access the protected server. Once inside the remoting session, the user can only do what the role capabilities file allows.

The following diagram shows the components of JEA:

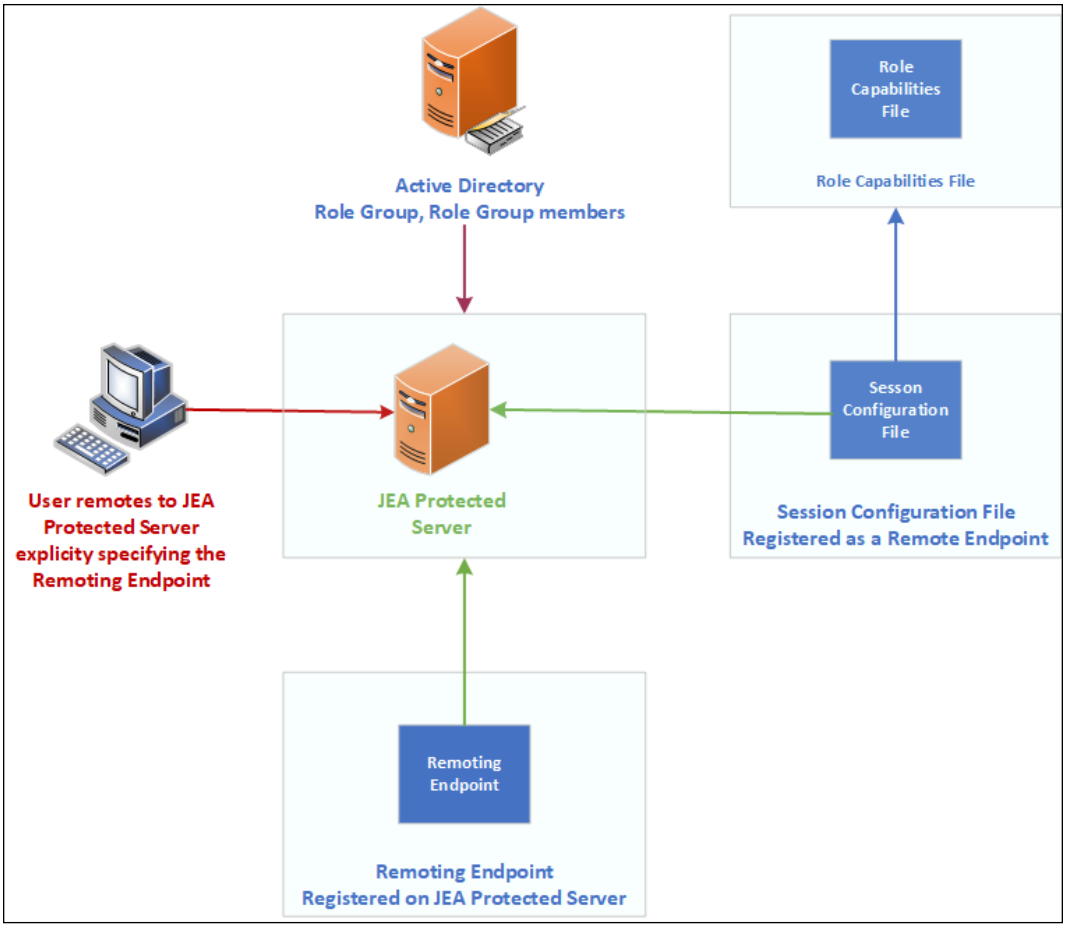


Figure 8.1: Components of JEA

Getting ready

This recipe uses `DC1`, a DC in the `Reskit.Org` domain on which you set up JEA for inbound connections. You installed `DC1` as a domain controller and configured users, groups, and OUs, both in *Chapter 5*, *Exploring .NET*. You run the first part of this recipe on `DC1`.

You would typically use a client computer to access the DC to manage DNS in production. For this recipe, adding an extra client host is replaced by using `DC1` to test JEA without requiring an additional host. Of course, in production, you should test JEA on a client host.

How to do it...

1. Creating a transcripts folder

```
New-Item -Path C:\JEATranscripts -ItemType Directory |
  Out-Null
```

2. Creating a role capabilities folder

```
$JEACF = "C:\JEACapabilities"
New-Item -Path $JEACF -ItemType Directory |
  Out-Null
```

3. Creating a JEA session configuration folder

```
$SCF = 'C:\JEASessionConfiguration'
New-Item -Path $SCF -ItemType Directory |
  Out-Null
```

4. Creating `DNSAdminsJEA` as a global security group

```
$DNSGHT = @{
  Name          = 'DNSAdminsJEA'
  Description   = 'DNS Admins for JEA'
  GroupCategory = 'Security'
  GroupScope    = 'Global'
}
New-ADGroup @DNSGHT
Get-ADGroup -Identity 'DNSAdminsJEA' |
  Move-ADObject -TargetPath 'OU=IT, DC=Reskit, DC=Org'
```

5. Adding `JerryG` to the DNS Admins group

```
$ADGHT = @{
  Identity  = 'DNSAdminsJEA'
  Members   = 'JerryG'
}
Add-ADGroupMember @ADGHT
```

6. Creating a role capabilities file

```
$RCF = Join-Path -Path $JEACF -ChildPath "DnsAdmins.psrc"
$RCHT = @{
  Path            = $RCF
  Author          = 'Reskit Administration'
  CompanyName     = 'Reskit.Org'
  Description     = 'DnsAdminsJEA role capabilities'
  AliasDefinition = @{Name='gh';Value='Get-Help'}
  ModulesToImport = 'Microsoft.PowerShell.Core','DnsServer'
```

```
  VisibleCmdlets  = (@{ Name        = "Restart-Computer";
                        Parameters  = @{Name = "ComputerName"}
                        ValidateSet = 'DC1, DC2'},
                      'DNSSERVER\*',
                     @{ Name        = "Stop-Service";
                        Parameters  = @{Name = "DNS"}},
                     @{ Name        = "Start-Service";
                        Parameters  = @{Name = "DNS"}}
                     )
  VisibleExternalCommands = ('C:\Windows\System32\whoami.exe',
                             'C:\Windows\System32\ipconfig.exe')
  VisibleFunctions = 'Get-HW'
  FunctionDefinitions = @{
    Name = 'Get-HW'
    Scriptblock = {'Hello JEA World'}}
}
New-PSRoleCapabilityFile @RCHT
```

7. Creating a JEA session configuration file

```
$P   = Join-Path -Path $SCF -ChildPath 'DnsAdmins.pssc'
$RDHT = @{
  'DnsAdminsJEA' =
      @{'RoleCapabilityFiles' =
        'C:\JEACapabilities\DnsAdmins.psrc'}
}
$PSCHT= @{
  Author              = 'DoctorDNS@Gmail.Com'
  Description         = 'Session Definition for DnsAdminsJEA'
  SessionType         = 'RestrictedRemoteServer'   # ie JEA!
  Path                = $P       # Role Capabilties file
  RunAsVirtualAccount = $true
  TranscriptDirectory = 'C:\JeaTranscripts'
  RoleDefinitions     = $RDHT     # tk role mapping
}
New-PSSessionConfigurationFile @PSCHT
```

8. Testing the session configuration file

```
Test-PSSessionConfigurationFile -Path $P
```

9. Enabling remoting on DC1

```
Enable-PSRemoting -Force |
  Out-Null
```

10. Registering the JEA session configuration remoting endpoint

```
$SCHT = @{
  Path  = $P
  Name  = 'DnsAdminsJEA'
  Force =  $true
}
Register-PSSessionConfiguration @SCHT
```

11. Viewing remoting endpoints

```
Get-PSSessionConfiguration  |
  Format-Table -Property NAME, PSVERSION, Run*Account
```

12. Verifying what the user can do

```
$SCHT = @{
  ConfigurationName = 'DnsAdminsJEA'
  Username          = 'Reskit\JerryG'
}
Get-PSSessionCapability  @SCHT |
  Sort-Object -Property Module
```

13. Creating credentials for user `JerryG`

```
$U    = 'JerryG@Reskit.Org'
$P    = ConvertTo-SecureString 'Pa$$w0rd' -AsPlainText -Force
$Cred = [PSCredential]::New($U,$P)
```

14. Defining three script blocks and an invocation splatting hash table

```
$SB1   = {Get-Command}
$SB2   = {Get-HW}
$SB3   = {Get-Command -Name  '*-DNSSERVER*'}
$ICMHT = @{
  ComputerName      = 'DC1.Reskit.Org'
  Credential        = $Cred
  ConfigurationName = 'DnsAdminsJEA'
}
```

15. Getting commands available within the JEA session

```
Invoke-Command -ScriptBlock $SB1 @ICMHT |
  Sort-Object -Property Module |
    Select-Object -First 15
```

16. Invoking a JEA-defined function in a JEA session as `JerryG`

```
Invoke-Command -ScriptBlock $SB2 @ICMHT
```

17. Getting `DNSServer` commands available to `JerryG`

```
$C = Invoke-Command -ScriptBlock $SB3 @ICMHT
"$($C.Count) DNS commands available"
```

18. Examining the contents of the transcripts folder

```
Get-ChildItem -Path $PSCHT.TranscriptDirectory
```

19. Examining a transcript

```
Get-ChildItem -Path $PSCHT.TranscriptDirectory |
  Select-Object -First 1  |
    Get-Content
```

How it works...

In *step 1*, you create a new folder that you use to hold JEA transcripts. In *step 2,* you create a folder to hold role capabilities files. In *step 3*, you create a folder to hold JEA session configuration files. These three steps produce no output.

In *step 4*, you create a global security group you use with JEA, and in *step 5*, you add the user `JerryG` to that group. Neither step produces output. By default, Windows creates this new group in the Users container in the AD, which may not be ideal. To ensure the account is in the right OU, you use `Move-ADGroup` to move it to the correct place.

In *step 6*, you create a role capabilities file and store it in the role capabilities folder you created in *step 2*. The role capabilities file contains metadata (including author and organization), aliases a user can access, modules to import in the JEA session, the visible cmdlets, console commands, and functions that the JEA server can access.

The JEA role capabilities file contains, in this recipe, a new function, `Get-HW`. This pretty simple function shows how you can define additional functions for the JEA user to access, possibly related to their specific role.

In *step 7,* you create a JEA session configuration file, which you store in the folder you created in *step 3*. *Steps 6* and *7* produce no output.

In *step 8*, you use the `Test-PSSessionConfigurationFile` command to test the session configuration file. The command produces the following output:

```
PS C:\Foo> # 8. Testing the session configuration file
PS C:\Foo> Test-PSSessionConfigurationFile -Path $P
True
```

Figure 8.2: Testing the session configuration file

In *step 9*, you use the `Enable-PSRemoting` command to ensure that you have configured `DC1` for WinRM remoting. This step creates no output.

In *step 10*, you complete the setup by registering a JEA session configuration remoting endpoint, producing no output.

In *step 11*, you use the `Get-PSSessionConfiguration` command to view the remoting endpoints available on `DC1`, with output like this:

```
PS C:\Foo> # 11. Viewing remoting endpoints
PS C:\Foo> Get-PSSessionConfiguration  |
             Format-Table -Property NAME, PSVersion, Run*Account

Name             PSVersion RunAsVirtualAccount
----             --------- -------------------
DnsAdminsJEA     7.1       True
PowerShell.7     7.1       false
PowerShell.7.1.1 7.1       false
```

Figure 8.3: Viewing remoting endpoints available on DC1

In *step 12*, you verify the commands that the JEA session configuration allows the user, `JerryG`, to access within the `DNSAdminsJEA` remoting endpoint, which produces the following output:

```
PS C:\Foo> # 12. Verifying what the user can do
PS C:\Foo> $SCHT = @{
             ConfigurationName = 'DnsAdminsJEA'
             Username          = 'Reskit\JerryG'
           }
PS C:\Foo> Get-PSSessionCapability  @SCHT |
             Sort-Object -Property Module


CommandType   Name                                Version    Source
-----------   ----                                -------    ------
Alias         clear -> Clear-Host
Function      Select-Object
Function      Restart-Computer
Function      Out-Default
Function      Measure-Object
Function      Get-HW
Function      Get-Help
Function      Get-FormatData
Application   ipconfig.exe                        10.0.2027… C:\Windows\system32\ipconfig.exe
Function      Get-Command
Function      Exit-PSSession
Function      Clear-Host
Application   whoami.exe                          10.0.2027… C:\Windows\system32\whoami.exe
Alias         gh -> Get-Help
Alias         gcm -> Get-Command
Alias         measure -> Measure-Object
Alias         select -> Select-Object
Alias         exsn -> Exit-PSSession
Alias         cls -> Clear-Host
Function      Add-DnsServerRecursionScope         2.0.0.0    DnsServer
Function      Reset-DnsServerZoneKeyMasterRole    2.0.0.0    DnsServer
Function      Restore-DnsServerPrimaryZone        2.0.0.0    DnsServer
Function      Restore-DnsServerSecondaryZone      2.0.0.0    DnsServer
Function      Resume-DnsServerZone                2.0.0.0    DnsServer
Function      Set-DnsServerCache                  2.0.0.0    DnsServer
Function      Remove-DnsServerZoneTransferPolicy  2.0.0.0    DnsServer
Function      Set-DnsServerClientSubnet           2.0.0.0    DnsServer
```

Figure 8.4: Verifying what the user JerryG can do

In *step 13*, you create a PowerShell credential object for `JerryG`. In *step 14*, you create a Windows credential object for `JerryG@Reskit.Org`. These two steps create no output.

In *step 14*, you create three script blocks and an invocation hash table for use in later steps, producing no output. In *step 15*, you invoke the `$SB1` script block inside a JEA session, with output (truncated) like this:

```
PS C:\Foo> # 15. Getting commands available within the JEA session
PS C:\Foo> Invoke-Command -ScriptBlock $SB1 @ICMHT |
             Sort-Object -Property Module |
               Select-Object -First 15

CommandType     Name                                               Version    Source        PSComputerName
-----------     ----                                               -------    ------        --------------
Function        Get-HW                                                                      DC1.Reskit.Org
Function        Select-Object                                                               DC1.Reskit.Org
Function        Clear-Host                                                                  DC1.Reskit.Org
Function        Restart-Computer                                                            DC1.Reskit.Org
Function        Exit-PSSession                                                              DC1.Reskit.Org
Function        Out-Default                                                                 DC1.Reskit.Org
Function        Get-Command                                                                 DC1.Reskit.Org
Function        Get-Help                                                                    DC1.Reskit.Org
Function        Get-FormatData                                                              DC1.Reskit.Org
Function        Measure-Object                                                              DC1.Reskit.Org
Function        Register-DnsServerDirectoryPartition               2.0.0.0    DnsServer     DC1.Reskit.Org
Function        Set-DnsServerConditionalForwarderZone              2.0.0.0    DnsServer     DC1.Reskit.Org
Function        Set-DnsServerClientSubnet                          2.0.0.0    DnsServer     DC1.Reskit.Org
Function        Set-DnsServerCache                                 2.0.0.0    DnsServer     DC1.Reskit.Org
Function        Set-DnsServer                                      2.0.0.0    DnsServer     DC1.Reskit.Org
```

Figure 8.5: Getting commands available within the JEA session

In *step 16*, you invoke the `$SB2` script block to call the `Get-HW` function defined in the role capabilities file:

```
PS C:\Foo> # 16. Invoking a JEA-defined function in a JEA session as JerryG
PS C:\Foo> Invoke-Command -ScriptBlock $SB2 @ICMHT
Hello JEA World
```

Figure 8.6: Invoking a JEA-defined function in a JEA session as JerryG

In *step 17*, you invoke the `$SB3` script block, which counts the number of commands available in the DNS server module that the user `JerryG` has permissions to use. The output is like this:

```
PS C:\Foo> # 17. Getting DNSServer commands available to JerryG
PS C:\Foo> $C = Invoke-Command -ScriptBlock $SB3 @ICMHT
PS C:\Foo> "$($C.Count) DNS commands available"
131 DNS commands available
```

Figure 8.7: Counting the number of DNS commands available to JerryG

When you set up JEA, you indicated JEA should create a transcript for each JEA session. In *step 18*, you examine the transcripts in the transcript folder, with output like this:

```
PS C:\Foo> # 18. Examining the contents of the transcripts folder
PS C:\Foo> Get-ChildItem -Path $PSCHT.TranscriptDirectory

    Directory: C:\JEATranscripts

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a---          21/01/2021     16:35          13073 PowerShell_transcript.DC1.7qAp4PE1.20210121163511.txt
-a---          21/01/2021     16:37          12844 PowerShell_transcript.DC1.BKqjIcrW.20210121163704.txt
-a---          21/01/2021     16:33          13688 PowerShell_transcript.DC1.cg9xfV+m.20210121163300.txt
-a---          21/01/2021     16:36            805 PowerShell_transcript.DC1.FrVp28pN.20210121163624.txt
```

Figure 8.8: Examining the transcripts folder contents

In the final step, *step 19*, you examine the first transcript in the transcripts folder, with output (truncated for publishing) that should look like this:

```
PS C:\Foo> # 19. Examining a transcript
PS C:\Foo> Get-ChildItem -Path $PSCHT.TranscriptDirectory |
             Select-Object -First 1  |
               Get-Content
**********************
PowerShell transcript start
Start time: 20210121163511
Username: RESKIT\JerryG
RunAs User: WinRM Virtual Users\WinRM VA_1_RESKIT_JerryG
Configuration Name: DnsAdminsJEA
Machine: DC1 (Microsoft Windows NT 10.0.20270.0)
Host Application: C:\Windows\system32\wsmprovhost.exe -Embedding
Process ID: 10040
PSVersion: 7.1.1
PSEdition: Core
GitCommitId: 7.1.1
OS: Microsoft Windows 10.0.20270
Platform: Win32NT
PSCompatibleVersions: 1.0, 2.0, 3.0, 4.0, 5.0, 5.1.10032.0, 6.0.0, 6.1.0, 6.2.0, 7.0.0, 7.1.1
PSRemotingProtocolVersion: 2.3
SerializationVersion: 1.1.0.1
WSManStackVersion: 3.0
**********************
PS>CommandInvocation(Get-Command): "Get-Command"
>> ParameterBinding(Get-Command): name="ListImported"; value="False"
>> ParameterBinding(Get-Command): name="ShowCommandInfo"; value="False"

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Function        Add-DnsServerClientSubnet                          2.0.0.0    DnsServer
Function        Add-DnsServerConditionalForwarderZone              2.0.0.0    DnsServer
Function        Add-DnsServerDirectoryPartition                    2.0.0.0    DnsServer
Function        Add-DnsServerForwarder                             2.0.0.0    DnsServer
Function        Add-DnsServerPrimaryZone                           2.0.0.0    DnsServer
Function        Add-DnsServerQueryResolutionPolicy                 2.0.0.0    DnsServer
Function        Add-DnsServerRecursionScope                        2.0.0.0    DnsServer
Function        Add-DnsServerResourceRecord                        2.0.0.0    DnsServer
Function        Add-DnsServerResourceRecordA                       2.0.0.0    DnsServer
```

Figure 8.9: Examining a transcript

There's more...

In *step 6*, you create a JEA role capabilities file. In this file, you specify what actions a user can perform within a JEA session. This includes the commands a user can run, modules they can load, JEA session-specific functions they can access, and more. You can, for example, specify that the JEA user can run a cmdlet that takes parameters, but you only allow specific parameter values. In short, the role capabilities file allows you to customize precisely what the JEA user can do in a JEA session. For more information on the JEA role capabilities and the role capabilities file, see `https://docs.microsoft.com/powershell/scripting/learn/remoting/jea/role-capabilities`.

If you are deploying JEA in an enterprise, you might have many servers and services managed with JEA sessions and a disparate user base. In such cases, you may wish to create a platform for your deployment to simplify creating the role definitions and your JEA deployment in general.

In *step 9*, you use the `Enable-PSRemoting` command. This command ensures you enable WinRM remoting and creates two standard PowerShell 7 remoting endpoints, in addition to the one you create in *step 10*.

In *step 15*, you run `$SB1` inside a JEA session on `DC1`. This script invokes `Get-Command` to list all the commands available to any member of the `DNSAdminsJEA` group. The output is truncated in the figure to take up a bit less space for publishing. The complete output lists all the commands available.

In *step 16*, you use the `Get-HW` function defined in the JEA role capabilities file. This function only exists within the JEA session.

In *step 18*, you examine the transcripts in the JEA transcripts folder. Depending on what you have done so far, you may see a different number of transcripts. Each transcript represents one use of a JEA session. The transcript contains full details of the user's commands inside the session and documents that the user initiated the session and when. The transcript provides valuable information for subsequent analysis where needed.

In the final step, *step 19*, you examine one of the JEA session transcripts. In *Figure 8.9*, you see the transcript generated by *step 15*. You also need to manage the transcripts folder, including archiving or removing older transcripts. If you are going to be implementing JEA widely, you may wish to develop some summary reporting based on each transcript's contents, including which users used any JEA session and when.

Examining Applications and Services Logs

Since the first version of Windows NT in 1993, anytime anything happens on Windows, the component responsible writes details to an event log. In the earlier versions of Windows Server, there were four different Windows logs:

- `Application`: holds events related to software you have installed on the server
- `Security`: holds events related to the security of your server
- `Setup`: holds events related to **Knowledge Base** (**KB**) installation and events that occurred during installation
- `System`: holds events that relate to this system, such as system start and system shut down

As well as these logs, other applications and features can add additional logs. You can see the classic and the additional logs using the Windows PowerShell `Get-Eventlog` cmdlet.

With Window Vista, Microsoft made some significant improvements to the event logging features. A substantial improvement was adding the Applications and Services Logs, which contain over four hundred individual logs. These extra logs allow Windows components to write to application-specific logs rather than the System or Application classic event logs, making it easier to find the events on a given host. There are hundreds of these Application and Services Logs that provide application-specific or service-specific event entries, but Windows does not enable all the logs by default. With PowerShell 7, you use `Get-WinEvent` to work with all of the event logs, including these newer ones.

In this recipe, you examine the logs and how to get log event details.

Getting ready

You run this recipe on `SRV1`, a domain-joined Windows Server. You also need `DC1`, a domain controller in the `Reskit.Org` domain. You have installed PowerShell 7 and Visual Studio Code on each system.

How to do it...

1. Registering PowerShell event log provider

```
& $PSHOME\RegisterManifest.ps1
```

2. Discovering classic event logs on `SRV1`

```
Get-EventLog -LogName *
```

3. Discovering and measuring all event logs on this host

```
$Logs = Get-WinEvent -ListLog *
"There are $($Logs.count) total event logs on SRV1"
```

4. Discovering and measuring all event logs on `DC1`

```
$SB1     = {Get-WinEvent -ListLog *}
$LogsDC1 = Invoke-Command -ComputerName DC1 -ScriptBlock $SB1
"There are $($LogsDC1.count) total event logs on DC1"
```

5. Discovering log member details

```
$Logs | Get-Member
```

6. Measuring enabled logs on SRV1

```
$Logs |
  Where-Object IsEnabled |
    Measure-Object |
      Select-Object -Property Count
```

7. Measuring enabled logs on DC1

```
$LogsDC1 |
  Where-Object IsEnabled |
    Measure-Object |
      Select-Object -Property Count
```

8. Measuring enabled logs that have records on SRV1

```
$Logs |
  Where-Object IsEnabled |
    Where-Object Recordcount -gt 0 |
      Measure-Object |
        Select-Object -Property Count
```

9. Discovering PowerShell-related logs

```
$Logs |
  Where-Object LogName -match 'powershell'
```

10. Examining PowerShellCore event log

```
Get-Winevent -LogName 'PowerShellCore/Operational' |
  Select-Object -First 10
```

How it works...

In *step 1*, you ensure that Windows has had the PowerShell event log provider registered. This step creates no output.

In *step 2*, you use `Get-EventLog` to discover the classic event logs on `SRV1`, with output like this:

```
PS C:\Foo> # 2. Discovering classic event logs on SRV1
PS C:\Foo> Get-EventLog -LogName *

  Max(K) Retain OverflowAction       Entries Log
  ------ ------ --------------       ------- ---
  20,480      0 OverwriteAsNeeded        552 Application
  20,480      0 OverwriteAsNeeded          0 HardwareEvents
     512      7 OverwriteOlder             0 Internet Explorer
  20,480      0 OverwriteAsNeeded          0 Key Management Service
     512      7 OverwriteOlder           376 Microsoft-ServerManagementExperience
  20,480      0 OverwriteAsNeeded       6,340 Security
  20,480      0 OverwriteAsNeeded       4,483 System
  15,360      0 OverwriteAsNeeded          30 Windows PowerShell
```

Figure 8.10: Discovering classic event logs on SRV1

In *step 3*, you use the `Get-WinEvent` cmdlet to discover all of the event logs on `SRV1`. The output looks like this:

```
PS C:\Foo> # 3. Discovering and measuring all event logs on this host
PS C:\Foo> $Logs = Get-WinEvent -ListLog *
PS C:\Foo> "There are $($Logs.Count) total event logs on SRV1"
There are 434 total event logs on SRV1
```

Figure 8.11: Discovering and counting all event logs on SRV1

In *step 4*, you discover and measure the number of event logs on the domain controller `DC1`, with output that looks like this:

```
PS C:\Foo> # 4. Discovering and measuring all event logs on DC1
PS C:\Foo> $SB1     = {Get-WinEvent -ListLog *}
PS C:\Foo> $LogsDC1 = Invoke-Command -ComputerName DC1 -ScriptBlock $SB1
PS C:\Foo> "There are $($LogsDC1.Count) total event logs on DC1"
There are 417 total event logs on DC1
```

Figure 8.12: Discovering and counting all event logs on DC1

In *step 5*, you use `Get-Member` to discover the properties of event logs you can use when querying. The output looks like this:

```
PS C:\Foo> # 5. Discovering log member details
PS C:\Foo> $Logs | Get-Member

   TypeName: System.Diagnostics.Eventing.Reader.EventLogConfiguration

Name                              MemberType   Definition
----                              ----------   ----------
Dispose                           Method       void Dispose(), void IDisposable.Dispose()
Equals                            Method       bool Equals(System.Object obj)
GetHashCode                       Method       int GetHashCode()
GetType                           Method       type GetType()
SaveChanges                       Method       void SaveChanges()
ToString                          Method       string ToString()
FileSize                          NoteProperty long FileSize=69632
IsLogFull                         NoteProperty bool IsLogFull=False
LastAccessTime                    NoteProperty datetime LastAccessTime=25/01/2021 12:40:50
LastWriteTime                     NoteProperty datetime LastWriteTime=25/01/2021 12:40:50
OldestRecordNumber                NoteProperty long OldestRecordNumber=1
RecordCount                       NoteProperty long RecordCount=30
IsClassicLog                      Property     bool IsClassicLog {get;}
IsEnabled                         Property     bool IsEnabled {get;set;}
LogFilePath                       Property     string LogFilePath {get;set;}
LogIsolation                      Property     System.Diagnostics.Eventing.Reader.EventLogIsolation LogIsolation {get;}
LogMode                           Property     System.Diagnostics.Eventing.Reader.EventLogMode LogMode {get;set;}
LogName                           Property     string LogName {get;}
LogType                           Property     System.Diagnostics.Eventing.Reader.EventLogType LogType {get;}
MaximumSizeInBytes                Property     long MaximumSizeInBytes {get;set;}
OwningProviderName                Property     string OwningProviderName {get;}
ProviderBufferSize                Property     System.Nullable[int] ProviderBufferSize {get;}
ProviderControlGuid               Property     System.Nullable[guid] ProviderControlGuid {get;}
ProviderKeywords                  Property     System.Nullable[long] ProviderKeywords {get;set;}
ProviderLatency                   Property     System.Nullable[int] ProviderLatency {get;}
ProviderLevel                     Property     System.Nullable[int] ProviderLevel {get;set;}
ProviderMaximumNumberOfBuffers    Property     System.Nullable[int] ProviderMaximumNumberOfBuffers {get;}
ProviderMinimumNumberOfBuffers    Property     System.Nullable[int] ProviderMinimumNumberOfBuffers {get;}
ProviderNames                     Property     System.Collections.Generic.IEnumerable[string] ProviderNames {get;}
SecurityDescriptor                Property     string SecurityDescriptor {get;set;}
```

Figure 8.13: Discovering log member details

Windows does not enable all event logs by default. In *step 6*, you discover the enabled logs on `SRV1`. The output looks like this:

```
PS C:\Foo> # 6. Measuring enabled logs on SRV1
PS C:\Foo> $Logs |
             Where-Object IsEnabled |
               Measure-Object |
                 Select-Object -Property Count

Count
-----
  353
```

Figure 8.14: Measuring enabled logs on SRV1

In *step 7*, you measure the enabled event logs on `DC1`, with output like this:

```
PS C:\Foo> # 7. Measuring enabled logs on DC1
PS C:\Foo> $LogsDC1 |
             Where-Object IsEnabled |
               Measure-Object |
                 Select-Object [-Property Count

Count
-----
  342
```

Figure 8.15: Measuring enabled logs on DC1

With *step 8*, you count the number of event logs you have enabled on `SRV1` containing event log entries. The output looks like this:

```
PS C:\Foo> # 8. Measuring enabled logs that have records on SRV1
PS C:\Foo> $Logs |
             Where-Object IsEnabled |
               Where-Object Recordcount -gt 0 |
                 Measure-Object |
                   Select-Object -Property Count

Count
-----
  106
```

Figure 8.16: Measuring enabled logs that have records on SRV1

In *step 9*, you discover which event logs could contain events for Windows PowerShell or PowerShell 7 (aka PowerShell Core). The output is as follows:

```
PS C:\Foo> # 9. Discovering PowerShell-related logs
PS C:\Foo> $Logs |
             Where-Object LogName -match 'powershell'

LogMode   MaximumSizeInBytes RecordCount LogName
-------   ------------------ ----------- -------
Circular           15728640          30 Windows PowerShell
Circular           15728640          71 PowerShellCore/Operational
Circular           15728640          10 Microsoft-Windows-PowerShell/Operational
Retain           1048985600           0 Microsoft-Windows-PowerShell/Admin
Circular            1052672           0 Microsoft-Windows-PowerShell-DesiredStateConfiguration-FileDownloadManager/Operational
```

Figure 8.17: Discovering PowerShell-related logs

In the final step in this recipe, *step 10*, you examine the `PowerShellCore` event log, with output like this:

```
PS C:\Foo> # 10. Examining PowerShellCore event log
PS C:\Foo> Get-Winevent -LogName 'PowerShellCore/Operational' |
             Select-Object -First 10

   ProviderName: PowerShellCore

TimeCreated                     Id LevelDisplayName Message
-----------                     -- ---------------- -------
25/01/2021 14:32:29          40962 Information      PowerShell console is ready for user input
25/01/2021 14:32:29          53504 Information      Windows PowerShell has started an IPC listening thread on process: 8244 in AppDomain: DefaultAppDomain.
25/01/2021 14:32:28          40961 Information      PowerShell console is starting up
25/01/2021 14:31:01           4100 Warning          Error Message = The specified wildcard character pattern is not valid: rp0[ertu…
25/01/2021 14:15:54           8197 Verbose          Runspace state changed to Closed
25/01/2021 14:15:54           8197 Verbose          Runspace state changed to Closing
25/01/2021 14:15:54          12039 Information      Modifying activity Id and correlating
25/01/2021 14:15:54           8196 Information      Modifying activity Id and correlating
25/01/2021 14:15:54          12039 Information      Modifying activity Id and correlating
25/01/2021 14:15:54           8196 Information      Modifying activity Id and correlating
```

Figure 8.18: Examining the PowerShellCore event log

There's more...

In *step 3* and *step 4*, you get a count of the number of event logs on `SRV1` and `DC1`. As you can see, the number of logs differs. Different Windows features and applications can add additional event logs for your use. In *step 6* and *step 7*, you also see the number of enabled logs on both systems. With *step 8*, you see how many enabled logs (on `SRV1`) actually contain event log entries.

In *step 9*, you see the event logs on `SRV1` related to Windows PowerShell and PowerShell 7. You examine the PowerShell Core logs in more detail in several recipes in this chapter.

Discovering logon events in the event log

Each time you attempt to log on, whether you are successful or not, Windows logs the attempt. These log events can help you determine who logged on to a computer and when.

In Windows, there are several different logon types. A logon type of 2 indicates a local console logon (that is, logging on to a physical host), while a logon type of 10 indicates logon over RDP. Other logon types include service logon (type 5), batch or scheduled task (type 4), and console unlock (type 7).

You can read more details in this article: `https://docs.microsoft.com/previous-versions/windows/it-pro/windows-server-2003/cc787567(v=ws.10)`. Note that this document is somewhat outdated and Microsoft has not updated it for later versions of Windows, although the information continues to be correct.

In this recipe, you use PowerShell to examine the Security event log and look at the logon events.

Getting ready

You run this recipe on DC1, a domain controller in the Reskit.Org forest.

How to do it...

1. Getting Security log events

```
$SecLog = Get-WinEvent -ListLog Security
"Security Event log entries:    [{0,10:N0}]" -f $Seclog.RecordCount
```

2. Getting all Windows Security log event details

```
$SecEvents = Get-WinEvent -LogName Security
"Found $($SecEvents.count) security events on DC1"
```

3. Examining Security event log event members

```
$SecEvents |
  Get-Member
```

4. Summarizing security events by event ID

```
$SecEvents |
  Sort-Object -Property Id |
    Group-Object -Property ID |
      Sort-Object -Property Name |
        Format-Table -Property Name, Count
```

5. Getting all successful logon events on DC1

```
$Logons = $SecEvents | Where-Object ID -eq 4624   # logon event
"Found $($Logons.Count) logon events on DC1"
```

6. Getting all failed logon events on DC1

```
$FLogons = $SecEvents | Where-Object ID -eq 4625   # failed logon event
"Found $($FLogons.Count) failed logon events on DC1"
```

7. Creating a summary array of successful logon events

```
$LogonEvents = @()
Foreach ($Logon in $Logons) {
  $XMLMSG = [xml] $Logon.ToXml()
  $Text = '#text'
  $HostName   = $XMLMSG.Event.EventData.data.$Text[1]
  $HostDomain = $XMLMSG.Event.EventData.data.$Text[2]
  $Account    = $XMLMSG.Event.EventData.data.$Text[5]
  $AcctDomain = $XMLMSG.Event.EventData.data.$Text[6]
  $LogonType  = $XMLMSG.Event.EventData.data.$Text[8]
  $LogonEvent = New-Object -Type PSCustomObject -Property @{
```

```
      Account   = "$AcctDomain\$Account"
      Host      = "$HostDomain\$Hostname"
      LogonType = $LogonType
      Time      = $Logon.TimeCreated
    }
    $LogonEvents += $logonEvent
}
```

8. Summarizing successful logon events on DC1

```
$LogonEvents |
  Group-Object -Property LogonType |
    Sort-Object -Property Name |
      Format-Table Name, Count
```

9. Creating a summary array of failed logon events on DC1

```
$FLogonEvents = @()
Foreach ($FLogon in $FLogons) {
  $XMLMSG = [xml] $FLogon.ToXml()
  $Text = '#text'
  $HostName   = $XMLMSG.Event.EventData.data.$Text[1]
  $HostDomain = $XMLMSG.Event.EventData.data.$Text[2]
  $Account    = $XMLMSG.Event.EventData.data.$Text[5]
  $AcctDomain = $XMLMSG.Event.EventData.data.$Text[6]
  $LogonType  = $XMLMSG.Event.EventData.data.$Text[8]
  $LogonEvent = New-Object -Type PSCustomObject -Property @{
     Account   = "$AcctDomain\$Account"
     Host      = "$HostDomain\$Hostname"
     LogonType = $LogonType
     Time      = $FLogon.TimeCreated
  }
  $FLogonEvents += $LogonEvent
}
```

10. Summarizing failed logon events on DC1

```
$FLogonEvents |
  Group-Object -Property Account |
    Sort-Object -Property Name |
      Format-Table Name, Count
```

How it works...

In *step 1*, you use the `Get-WinEvent` cmdlet to retrieve details about the Security log on DC1. Then you display the number of events in the log. The output looks like this:

```
PS C:\Foo> # 1. Getting Security log events
PS C:\Foo> $SecLog = Get-WinEvent -ListLog Security
PS C:\Foo> "Security Event log entries:    [{0,10:N0}]" -f $Seclog.RecordCount
Security Event log entries:    [    29,940]
```

Figure 8.19: Getting Security log events

In *step 2*, you use `Get-WinEvent` to retrieve all events from the Security log and display a count of the events returned, with output like this:

```
PS C:\Foo> # 2. Getting all Windows Security log event details
PS C:\Foo> $SecEvents = Get-WinEvent -LogName Security
"Found $($SecEvents.count) security events"
Found 29943 security events on DC1
```

Figure 8.20: Getting all Windows Security log event details

The `Get-WinEvent` cmdlet returns objects that contain individual event log entries. Each object is of the type `System.Diagnostics.Eventing.Reader.EventLogRecord`. In *step 3*, you view the members of this .NET object class, with output like this:

```
PS C:\Foo> # 3: Examining Security event log entry members
PS C:\Foo> $SecEvents | Get-Member

   TypeName: System.Diagnostics.Eventing.Reader.EventLogRecord
Name                MemberType   Definition
----                ----------   ----------
Dispose             Method       void Dispose(), void IDisposable.Dispose()
Equals              Method       bool Equals(System.Object obj)
FormatDescription   Method       string FormatDescription(), string FormatDescription(System.Collections.Ge…
GetHashCode         Method       int GetHashCode()
GetPropertyValues   Method       System.Collections.Generic.IList[System.Object] GetPropertyValues(System.D…
GetType             Method       type GetType()
ToString            Method       string ToString()
ToXml               Method       string ToXml()
Message             NoteProperty string Message=An account was logged off.…
ActivityId          Property     System.Nullable[guid] ActivityId {get;}
Bookmark            Property     System.Diagnostics.Eventing.Reader.EventBookmark Bookmark {get;}
ContainerLog        Property     string ContainerLog {get;}
Id                  Property     int Id {get;}
Keywords            Property     System.Nullable[long] Keywords {get;}
KeywordsDisplayNames Property    System.Collections.Generic.IEnumerable[string] KeywordsDisplayNames {get;}
Level               Property     System.Nullable[byte] Level {get;}
LevelDisplayName    Property     string LevelDisplayName {get;}
LogName             Property     string LogName {get;}
MachineName         Property     string MachineName {get;}
MatchedQueryIds     Property     System.Collections.Generic.IEnumerable[int] MatchedQueryIds {get;}
Opcode              Property     System.Nullable[short] Opcode {get;}
OpcodeDisplayName   Property     string OpcodeDisplayName {get;}
ProcessId           Property     System.Nullable[int] ProcessId {get;}
Properties          Property     System.Collections.Generic.IList[System.Diagnostics.Eventing.Reader.EventP…
ProviderId          Property     System.Nullable[guid] ProviderId {get;}
ProviderName        Property     string ProviderName {get;}
Qualifiers          Property     System.Nullable[int] Qualifiers {get;}
RecordId            Property     System.Nullable[long] RecordId {get;}
RelatedActivityId   Property     System.Nullable[guid] RelatedActivityId {get;}
Task                Property     System.Nullable[int] Task {get;}
TaskDisplayName     Property     string TaskDisplayName {get;}
ThreadId            Property     System.Nullable[int] ThreadId {get;}
TimeCreated         Property     System.Nullable[datetime] TimeCreated {get;}
UserId              Property     System.Security.Principal.SecurityIdentifier UserId {get;}
Version             Property     System.Nullable[byte] Version {get;}
```

Figure 8.21: Examining Security event log entry members

Once you have retrieved the events in the Security log, you can examine the different security event types, held in the ID field of each log record. In *step 4*, you view and count the different event IDs in the Security log, which looks like this:

```
PS C:\Foo> # 4. Summarizing security events by event ID
PS C:\Foo> $SecEvents |
             Sort-Object -Property Id |
               Group-Object -Property ID |
                 Sort-Object -Property Name |
                   Format-Table -Property Name, Count

Name Count
---- -----
4616     6
4624 10507
4625     3
4634  9813
4647     1
4648    45
4672  8671
4768   210
4769   354
4770    27
4776     3
4799     7
4907    66
5058     1
5059     1
5061     1
5379   227
```

Figure 8.22: Summarizing security events by event ID

There are two related logon events you can track. Log entries with an event ID of `4624` represent successful logon events, while `4625` represents failed logons. In *step 5*, you get ALL the successful logon events, with output like this:

```
PS C:\Foo> # 5. Getting all successful logon events on DC1
PS C:\Foo> $Logons = $SecEvents | Where-Object ID -eq 4624   # logon event
PS C:\Foo> "Found $($Logons.count) logon events"
Found 10507 logon events
```

Figure 8.23: Getting all successful logon events on DC1

In *step 6*, you count the number of logon failures on `DC1`, which looks like this:

```
PS C:\Foo> # 6. Getting all failed logon events on DC1
PS C:\Foo> $FLogons = $SecEvents | Where-Object ID -eq 4625   # failed logon event
PS C:\Foo> "Found $($FLogons.Count) failed logon events on DC1"
Found 3 failed logon events on DC1
```

Figure 8.24: Getting all failed logon events on DC1

In *step 7*, you create a summary array of all the successful logons. This step produces no output. In *step 8*, you summarize the logon events, with output like this:

```
PS C:\Foo> # 8. Summarizing successful logon events on DC1
PS C:\Foo> $LogonEvents |
             Group-Object -Property LogonType |
               Sort-Object -Property Name |
                 Select-Object -Property Name,Count

Name Count
---- -----
10       2
2       13
3     9801
5      691
```

Figure 8.25: Summarizing successful logon events on DC1

In *step 9*, you summarize the failed logon events on `DC1`. You display the details of unsuccessful logons with *step 10*, which looks like this:

```
PS C:\Foo> # 10. Summarizing failed logon events on DC1
PS C:\Foo> $FLogonEvents |
             Group-Object -Property Account |
               Sort-Object -Property Name |
                 Format-Table Name, Count

Name                 Count
----                 -----
cookham\bar              2
RESKIT\Administrator     1
```

Figure 8.26: Summarizing failed logon events on DC1

There's more...

In *step 1*, you retrieve a summary of the events in the Security log and display the number of events in the log. In *step 2*, you retrieve and count the number of entries. As you can see in the figures above, the counts do not match. The event counts may differ since Windows is constantly logging additional events to the Security log. The additional events are events generated by background tasks or services. This minor discrepancy is not unexpected and is harmless.

In *step 3*, you view the members of log event objects. You can discover more about the members of the class at `https://docs.microsoft.com/dotnet/api/system.diagnostics.eventing.reader.eventlogrecord`.

In *step 6*, you obtain unsuccessful logon events. To obtain unsuccessful logons, you need to ensure you have attempted to log on to `DC1` but with invalid credentials. As you see in the output of *step 10* in *Figure 8.26*, there were two users involved with the three unsuccessful logon attempts on `DC1`.

In *step 9*, you view the failed logons. Depending on which user you have attempted to log on to this server (and failed), the results you see in this step may differ from the above figure.

Deploying PowerShell group policies

Group policies are groups of policies you can deploy that control a user or computer environment. The policies define what a given user can and cannot do on a given Windows computer. For example, you can create a **Group Policy Object** (**GPO**) to set policies that define what screen saver to use, allow the user to see the Control Panel, or specify a default PowerShell execution policy. There are over 2,500 individual settings that you can deploy.

After you create a GPO and specify the policies to deploy, you can apply it to any OU in your domain. An OU is a container object within AD that can contain both other OUs and leaf objects such as AD user, computer, or group objects. You use OUs to support both the deployment of GPOs and the delegation of AD administration.

You can apply a GPO to an OU, but also to the domain or to an AD site. Additionally, you can specify whether policies within a given GPO are to apply to users, computers, or both. GPOs provide you with considerable flexibility in how you restrict what users can do on a workstation or a server.

With Windows PowerShell 5.1, Microsoft includes a set of five Group Policy settings. The PowerShell team has extended the policies you can use in PowerShell 7. By default, the installation of PowerShell 7, even on a DC, does not install the necessary GPO administrative template files.

In the PowerShell home folder (`$PSHOME`), you can find the policy template files and a script to install them. After installing PowerShell on your domain controller, you run the installation script in the `$PSHOME` folder and install the policy definitions. You either do this on all DCs or to the central policy store if you use one.

For more details on PowerShell 7's group policies, see `https://docs.microsoft.com/powershell/module/microsoft.powershell.core/about/about_group_policy_settings?view=powershell-7.1`.

In this recipe, you discover the files necessary to add PowerShell 7 GPO support, run the installer, then create a GPO to deploy a set of PowerShell-related policies.

Getting ready

You run this recipe on `DC1` after you install PowerShell 7 and Visual Studio. `DC1` is a domain controller in the `Reskit.Org` domain that you have used in earlier chapters.

How to do it...

1. Discovering the GPO-related files

```
Get-ChildItem -Path $PSHOME -Filter *Core*Policy*
```

2. Installing the PowerShell 7 group policy files

```
$LOC = 'C:\Program Files\PowerShell\7\' +         # $PSHome
       'InstallPSCorePolicyDefinitions.ps1'       # Script
& $LOC -VERBOSE
```

3. Creating and displaying a new GPO for the IT group

```
$PshGPO = New-GPO -Name 'PowerShell GPO for IT'
```

4. Enabling module logging

```
$GPOKEY1 =
  'HKCU\Software\Policies\Microsoft\PowerShellCore\ModuleLogging'
$GPOHT1 = @{
  DisplayName    = $PshGPO.DisplayName
  Key            = $GPOKEY1
  Type           = [Microsoft.Win32.RegistryValueKind]::DWord
  ValueName      = 'EnableModuleLogging'
  Value          = 1
}
Set-GPRegistryValue @GPOHT1 | Out-Null
```

5. Configuring module names to log

```
$GPOHT2 = @{
  DisplayName    = $PshGPO.DisplayName
  Key            = "$GPOKEY1\ModuleNames"
  Type           = [Microsoft.Win32.RegistryValueKind]::String
  ValueName      = 'ITModule1', 'ITModule2'
  Value          = 'ITModule1', 'ITModule2'
 }
Set-GPRegistryValue @GPOHT2 | Out-Null
```

6. Enabling Script Block Logging

```
$GPOKey3 =
  'HKCU\Software\Policies\Microsoft\PowerShellCore\ScriptBlockLogging'
$GPOHT3  = @{
    DisplayName    = $PshGPO.DisplayName
    Key            = $GPOKEY3
    Type           = [Microsoft.Win32.RegistryValueKind]::DWord
    ValueName      = 'EnableScriptBlockLogging'
    Value          = 1
```

```
   }
Set-GPRegistryValue @GPOHT3 | Out-Null
```

7. Enabling an unrestricted execution policy

```
$GPOKey4 =
  'HKCU\Software\Policies\Microsoft\PowerShellCore'
# create the key value to enable
$GPOHT4 =  @{
    DisplayName    = $PshGPO.DisplayName
    Key            = $GPOKEY4
    Type           = [Microsoft.Win32.RegistryValueKind]::DWord
    ValueName      = 'EnableScripts'
    Value          = 1
  }
  Set-GPRegistryValue @GPOHT4 | Out-Null
# Set the default execution policy
$GPOHT4 = @{
  DisplayName    = $PshGPO.DisplayName
  Key            = "$GPOKEY4"
  Type           = [Microsoft.Win32.RegistryValueKind]::String
  ValueName      = 'ExecutionPolicy'
  Value          = 'Unrestricted'
}
Set-GPRegistryValue @GPOHT4
```

8. Assigning the GPO to the IT OU

```
$Target = "OU=IT, DC=Reskit, DC=Org"
New-GPLink -DisplayName $PshGPO.Displayname -Target $Target |
   Out-Null
```

9. Creating an RSOP report

```
$RSOPHT = @{
  ReportType = 'HTML'
  Path       = 'C:\Foo\GPOReport.Html'
  User       = 'Reskit\JerryG'
}
Get-GPResultantSetOfPolicy @RSOPHT
& $RSOPHT.Path
```

How it works...

In *step 1*, you discover the PowerShell 7 GPO files, with output like this:

```
PS C:\Foo> # 1. Discovering the GPO-related files
PS C:\Foo> Get-ChildItem -Path $PSHOME -Filter *Core*Policy*

    Directory: C:\Program Files\PowerShell\7

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a---          13/01/2021    18:47          15882 InstallPSCorePolicyDefinitions.ps1
-a---          13/01/2021    18:39           9675 PowerShellCoreExecutionPolicy.adml
-a---          13/01/2021    18:39           6198 PowerShellCoreExecutionPolicy.admx
```

Figure 8.27: Discovering the GPO-related files

In *step 2*, you first create a string that holds the location of the GPO installation file. Then you run this file to install the GPO files, which looks like this:

```
PS C:\Foo> # 2. Installing the PowerShell 7 group policy files
PS C:\Foo> $LOC = 'C:\Program Files\PowerShell\7\' +         # $PSHome
                  'InstallPSCorePolicyDefinitions.ps1'       # Script
PS C:\Foo> & $LOC -VERBOSE
VERBOSE: Copying C:\Program Files\PowerShell\7\PowerShellCoreExecutionPolicy.admx to
                 C:\WINDOWS\PolicyDefinitions
VERBOSE: PowerShellCoreExecutionPolicy.admx was installed successfully
VERBOSE: Copying C:\Program Files\PowerShell\7\PowerShellCoreExecutionPolicy.adml to
                 C:\WINDOWS\PolicyDefinitions\en-US
VERBOSE: PowerShellCoreExecutionPolicy.adml was installed successfully
```

Figure 8.28: Installing the PowerShell 7 group policy files

In *step 3*, you create a new GPO, which creates the following output:

```
PS C:\Foo> # 3. Creating and displaying a new GPO for IT group
PS C:\Foo> $PshGPO = New-GPO -Name 'PowerShell GPO for IT'
PS C:\Foo> $PshGPO

DisplayName      : PowerShell GPO for IT
DomainName       : Reskit.Org
Owner            : RESKIT\Domain Admins
Id               : 237d2399-5e58-4aae-aaf3-54d420b7f854
GpoStatus        : AllSettingsEnabled
Description      :
CreationTime     : 27/01/2021 13:09:10
ModificationTime : 27/01/2021 13:09:10
UserVersion      :
ComputerVersion  :
WmiFilter        :
```

Figure 8.29: Creating and displaying a new GPO for the IT group

In *step 4*, you configure the GPO to enable module logging, and in *step 5*, you configure the module names to log. In *step 6*, you enable Script Block Logging, and in *step 7*, you configure the GPO to enable an unrestricted PowerShell execution policy. These four steps produce no output.

In *step 8*, you assign this GPO to the IT OU, creating no output. In the final step, *step 9*, you create and view a report on the resultant set of policies, which looks like this:

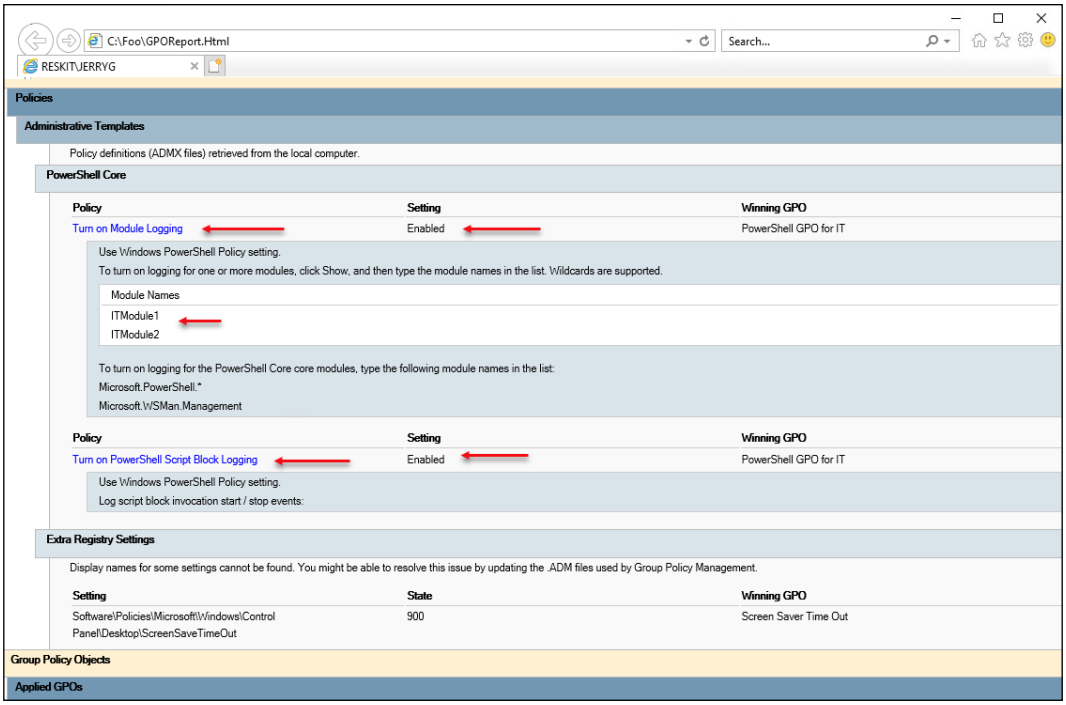


Figure 8.30: Policies report

There's more...

This recipe creates a new GPO, configures the object with specific policy values, and then assigns it to the IT OU in the `Reskit.Org` domain. When any user in the IT group logs on, PowerShell performs the specified logging and uses an unrestricted execution policy. You can see in the RSOP report, produced in *step 9*, which policy settings PowerShell applies. In order to ensure you get sensible output in your RSOP report, you need to ensure that the user `JerryG` has logged on to the DC.

Using PowerShell Script Block Logging

In the *Deploying PowerShell group policies* recipe, you saw how you could deploy policies related to PowerShell 7. One of these policies, Script Block Logging, causes PowerShell 7 to generate log events whenever you cause the execution of a script block that PowerShell deems noteworthy.

In addition to using Group Policy to invoke Script Block Logging, you can also configure the local registry. In effect, this mimics using the Local Group Policy editor. Using the editor provides a nice interface to the policies, but you can't really automate a GUI. If you are making a single change to a single policy, then the GUI may be more convenient. But if you are making changes to or creating more policies, using a PowerShell script may be more productive.

Getting ready

You run this recipe on `DC1`, a domain controller in the `Reskit.Org` domain. You must log on as `Reskit\Administrator`, a member of the Domain Administrators group.

How to do it...

1. Clearing the PowerShell Core operational log

```
WevtUtil cl 'PowerShellCore/Operational'
```

2. Enabling Script Block Logging for the current user

```
$SBLPath = 'HKCU:\Software\Policies\Microsoft\PowerShellCore' +
           '\ScriptBlockLogging'
if (-not (Test-Path $SBLPath))  {
        $null = New-Item $SBLPath -Force
    }
Set-ItemProperty $SBLPath -Name EnableScriptBlockLogging -Value '1'
```

3. Examining the PowerShell Core event log for `4104` events

```
Get-Winevent -LogName 'PowerShellCore/Operational' |
  Where-Object Id -eq 4104
```

4. Examining logged event details

```
Get-Winevent -LogName 'PowerShellCore/Operational' |
  Where-Object Id -eq 4104  |
    Select-Object -First 1 |
      Format-List -Property ID, LogName, Message
```

5. Creating another script block that PowerShell does not log

```
$SBtolog = {Get-CimInstance -Class Win32_ComputerSystem | Out-Null}
$Before = Get-WinEvent -LogName 'PowerShellCore/Operational'
Invoke-Command -ScriptBlock $SBtolog
$After = Get-WinEvent -LogName 'PowerShellCore/Operational'
```

6. Comparing the events before and after you invoke the command

```
"Before:  $($Before.Count) events"
"After :  $($After.Count) events"
```

7. Removing registry policy entry

```
Remove-Item -Path $SBLPath
```

How it works...

In *step 1*, you use the `wevtutil.exe` console application to clear the PowerShell Core operational log. In *step 2*, you update the current user's registry to enable Script Block Logging (for the currently logged-on user `Reskit\Administrator`). These steps produce no output.

In *step 3*, you examine the PowerShell Core log for `4104` events, with output like this:

```
PS C:\Foo> # 3.Examining the PowerShell Core event log for 4104 events
PS C:\Foo> Get-Winevent -LogName 'PowerShellCore/Operational' |
            Where-Object Id -eq 4104

   ProviderName: PowerShellCore

TimeCreated             Id LevelDisplayName Message
-----------             -- ---------------- -------
31/01/2021 12:02:44   4104 Warning          Creating Scriptblock text (1 of 1):…
```

Figure 8.31: Examining the PowerShell Core event log for 4104 events

In *step 4*, you view the details of the event log entry you saw in the previous step, with output that looks like this:

```
PS C:\Foo> # 4. Examining logged event details
PS C:\Foo> Get-Winevent -LogName 'PowerShellCore/Operational' |
             Where-Object Id -eq 4104  |
               Select-Object -First 1 |
                 Format-List -Property ID, Logname, Message

Id      : 4104
LogName : PowerShellCore/Operational
Message : Creating Scriptblock text (1 of 1):
          # 1. Clear PowerShell Core operational log
          WevtUtil cl 'PowerShellCore/Operational'

          # 2. Enabling script block logging for current user
          $SBLPath = 'HKCU:\Software\Policies\Microsoft\PowerShellCore' +
                     '\ScriptBlockLogging'
          if (-not (Test-Path $SBLPath))  {
                $null = New-Item $SBLPath -Force
              }
          Set-ItemProperty $SBLPath -Name EnableScriptBlockLogging -Value '1'

          ScriptBlock ID: ccdf2e53-e827-4f40-aeb0-3af47414b2d1
          Path:
```

Figure 8.32: Examining logged event details

In *step 5*, you create and execute another script block, one that Script Block Logging does not consider important enough to log. This step creates a count of the total number of event log entries before and after invoking the script block. This step produces no output, but in *step 6*, you view the before and after counts, like this:

```
PS C:\Foo> # 6. Comparing the events before and after you invoke the command
PS C:\Foo> "Before:  $($Before.Count) events"
PS C:\Foo> "After :  $($After.Count) events"
Before:  1 events
After :  1 events
```

Figure 8.33: Comparing the events before and after invoking the command

In the final step, *step 7*, you remove the policy entry from the registry, producing no output.

There's more...

In *step 1*, you use the `wevtutil.exe` console application to clear an event log. With Windows PowerShell, you have the `Clear-EventLog` cmdlet, which you can use to clear an event log. This cmdlet does not exist in PowerShell 7, which is why you use a Win32 console application to clear the log.

In *step 6*, you can see that the script block you executed did not result in PowerShell logging the script block. That is to be expected, and it does reduce the amount of logging carried out.

Configuring AD password policies

Passwords are essential for security as they help ensure that a person is who they say they are and thus are allowed to perform some action such as logging on to a host or editing a file. Password policies allow you to define your password attributes, including minimum length and whether complex passwords are required. You can also set the number of times a user enters an invalid password before that user is locked out (and a lockout duration). For more details on improving authentication security, see `https://www.microsoftpressstore.com/articles/article.aspx?p=2224364&seqNum=2`.

In AD, you can apply a default domain password policy. This policy applies to all users in the domain. In most cases, this is adequate for the organization. But in some cases, you may wish to apply a more stringent password policy to certain users or groups of users. You use AD's fine-grained password policy to manage these more restrictive passwords. A "fine-grained" policy is one you can apply to just a single user, as opposed to every user in the domain.

Getting ready

You run this recipe on `DC1`, a domain controller in the `Reskit.Org` domain. You must log on as a Domain Administrator.

How to do it...

1. Discovering the current domain password policy

   ```
   Get-ADDefaultDomainPasswordPolicy
   ```

2. Discovering if there is a fine-grained password policy for `JerryG`

   ```
   Get-ADFineGrainedPasswordPolicy -Identity 'JerryG'
   ```

3. Updating the default password policy

```
$DPWPHT = [Ordered] @{
    LockoutDuration             = '00:45:00'
    LockoutObservationWindow    = '00:30:00'
    ComplexityEnabled           = $true
    ReversibleEncryptionEnabled = $false
    MinPasswordLength           = 6
}
Get-ADDefaultDomainPasswordPolicy -Current LoggedOnUser |
  Set-ADDefaultDomainPasswordPolicy @DPWPHT
```

4. Checking updated default password policy

```
Get-ADDefaultDomainPasswordPolicy
```

5. Creating a fine-grained password policy

```
$PD = 'DNS Admins Group Fine-grained Password Policy'
$FGPHT = @{
  Name                        = 'DNSPWP'
  Precedence                  = 500
  ComplexityEnabled           = $true
  Description                 = $PD
  DisplayName                 = 'DNS Admins Password Policy'
  LockoutDuration             = '0.12:00:00'
  LockoutObservationWindow = '0.00:15:00'
  LockoutThreshold            = 4
}
New-ADFineGrainedPasswordPolicy @FGPHT
```

6. Assigning the policy to DNS Admins

```
$DNSADmins = Get-ADGroup -Identity DNSAdmins
$ADDHT = @{
  Identity  = 'DNSPWP'
  Subjects  = $DNSADmins
}
Add-ADFineGrainedPasswordPolicySubject  @ADDHT
```

7. Assigning the policy to `JerryG`

```
$Jerry = Get-ADUser -Identity JerryG
Add-ADFineGrainedPasswordPolicySubject -Identity DNSPWP -Subjects $Jerry
```

8. Checking on policy application for the group

```
Get-ADGroup 'DNSAdmins' -Properties * |
  Select-Object -Property msDS-PSOApplied
```

9. Checking on policy application for the user

```
Get-ADUser JerryG -Properties * |
  Select-Object -Property msDS-PSOApplied
```

10. Getting DNS Admins policy

```
Get-ADFineGrainedPasswordPolicy -Identity DNSPWP
```

11. Checking on JerryG's resultant password policy

```
Get-ADUserResultantPasswordPolicy -Identity JerryG
```

How it works...

In *step 1*, you retrieve the default AD password policy, which looks like this:

```
PS C:\Foo> # 1. Discovering the current domain password policy
PS C:\Foo> Get-ADDefaultDomainPasswordPolicy

ComplexityEnabled           : True
DistinguishedName           : DC=Reskit,DC=Org
LockoutDuration             : 00:30:00
LockoutObservationWindow    : 00:30:00
LockoutThreshold            : 0
MaxPasswordAge              : 42.00:00:00
MinPasswordAge              : 1.00:00:00
MinPasswordLength           : 7
objectClass                 : {domainDNS}
objectGuid                  : b1cc231b-02bb-42d2-9dba-fe2cd5696113
PasswordHistoryCount        : 24
ReversibleEncryptionEnabled : False
```

Figure 8.34: Discovering the current domain password policy

In *step 2*, you check to see if there are any fine-grained password policies for the user `JerryG`, which looks like this:

```
PS C:\Foo> # 2. Discovering if there is a fine-grained password policy for JerryG
PS C:\Foo> Get-ADFineGrainedPasswordPolicy -Identity 'Reskit\IT\JerryG'
Get-ADFineGrainedPasswordPolicy: Cannot find an object with identity:
'Reskit\IT\JerryG' under: 'DC=Reskit,DC=Org'
```

Figure 8.35: Checking for fine-grained password policies

In *step 3*, you update the default password policy for the domain, changing a few settings. This produces no output. In *step 4*, you review the updated default password policy, which looks like this:

```
PS C:\Foo> # 4. Checking updated default password policy
PS C:\Foo> Get-ADDefaultDomainPasswordPolicy

ComplexityEnabled           : True
DistinguishedName           : DC=Reskit,DC=Org
LockoutDuration             : 00:45:00
LockoutObservationWindow    : 00:30:00
LockoutThreshold            : 0
MaxPasswordAge              : 42.00:00:00
MinPasswordAge              : 1.00:00:00
MinPasswordLength           : 6
objectClass                 : {domainDNS}
objectGuid                  : b1cc231b-02bb-42d2-9dba-fe2cd5696113
PasswordHistoryCount        : 24
ReversibleEncryptionEnabled : False
```

Figure 8.36: Checking the updated default password policy

In *step 5*, you create a new fine-grained password policy with some overrides to the default domain policy you looked at above. In *step 6*, you assign the policy to the DNS Admins group, and in *step 7*, you apply this policy explicitly to the user `JerryG`. These three steps create no output.

In *step 8*, you check on the policy application for the DNS Admins group, which looks like this:

```
PS C:\Foo> # 8. Checking on policy application for the group
PS C:\Foo> Get-ADGroup 'DNSAdmins' -Properties * |
             Select-Object -Property msDS-PSOApplied

msDS-PSOApplied
---------------
{CN=DNDPWP,CN=Password Settings Container,CN=System,DC=Reskit,DC=Org}
```

Figure 8.37: Checking on the policy application for the DNS Admins group

In *step 9*, you check on the password policy applied to the user `JerryG`, which looks like this:

```
PS C:\Foo> # 9. Checking on policy application for the user
PS C:\Foo> Get-ADUser JerryG -Properties * |
             Select-Object -Property msDS-PSOApplied

msDS-PSOApplied
---------------
{CN=DNSPWP,CN=Password Settings Container,CN=System,DC=Reskit,DC=Org}
```

Figure 8.38: Checking on policy application for the user

In *step 10*, you examine the DNS Admins password policy, with output like this:

```
PS C:\Foo> # 10. Getting DNS Admins policy
PS C:\Foo> Get-ADFineGrainedPasswordPolicy -Identity DNSPWP

AppliesTo                   : {CN=DnsAdmins,CN=Users,DC=Reskit,DC=Org, CN=Jerry Garcia,OU=IT,DC=Reskit,DC=Org}
ComplexityEnabled           : True
DistinguishedName           : CN=DNSPWP,CN=Password Settings Container,CN=System,DC=Reskit,DC=Org
LockoutDuration             : 12:00:00
LockoutObservationWindow    : 00:42:00
LockoutThreshold            : 3
MaxPasswordAge              : 42.00:00:00
MinPasswordAge              : 1.00:00:00
MinPasswordLength           : 7
Name                        : DNSPWP
ObjectClass                 : msDS-PasswordSettings
ObjectGUID                  : 71e202f0-ac54-4772-9420-04a3aba69276
PasswordHistoryCount        : 24
Precedence                  : 500
ReversibleEncryptionEnabled : Tru
```

Figure 8.39: Getting the DNS Admins password policy

In the final step, *step 11*, you examine the resulting password policy for the user `JerryG`, which looks like this:

```
PS C:\Foo> # 11. Checking on JerryG's resultant password policy
PS C:\Foo> Get-ADUserResultantPasswordPolicy -Identity JerryG

AppliesTo                   : {CN=DnsAdmins,CN=Users,DC=Reskit,DC=Org, CN=Jerry Garcia,OU=IT,DC=Reskit,DC=Org}
ComplexityEnabled           : True
DistinguishedName           : CN=DNSPWP,CN=Password Settings Container,CN=System,DC=Reskit,DC=Org
LockoutDuration             : 12:00:00
LockoutObservationWindow    : 00:42:00
LockoutThreshold            : 3
MaxPasswordAge              : 42.00:00:00
MinPasswordAge              : 1.00:00:00
MinPasswordLength           : 7
Name                        : DNSPWP
ObjectClass                 : msDS-PasswordSettings
ObjectGUID                  : 71e202f0-ac54-4772-9420-04a3aba69276
PasswordHistoryCount        : 24
Precedence                  : 500
ReversibleEncryptionEnabled : True
```

Figure 8.40: Checking JerryG's resultant password policy

There's more...

In *step 1*, you view the existing default domain password policy. The settings you see in this step were created by the installation process when you installed Windows Server on `DC1`.

In *step 2*, you attempt to find a fine-grained password policy that would apply to the user `JerryG`, which does not exist.

In *step 5,* you create a new fine-grained password policy that you assign to the DNS Admins group (in *step 6*) and `JerryG` (in *step 7*). This assignment ensures the policy applies to `JerryG`, whether or not he is a DNS Admins group member.

In *step 11*, you see the password policy settings for the user `JerryG`. These settings derive from the default domain policy plus the settings you specified in the `DNSPWP` policy. In theory, you could have a user with the effective password policy settings coming from multiple policy objects (for example, one GPO for the domain, one for an OU, and so on), although you should probably avoid such complexity.

Managing Windows Defender Antivirus

Microsoft Defender Antivirus is the next-generation protection component of Microsoft Defender for Endpoint. Defender Antivirus provides antivirus and antimalware facilities. The product also does some packet analysis to detect network-level attacks.

The Windows installation process installs Defender on both Windows 10 and Windows Server 2022, by default, although you can remove the feature should you wish. For more details on Defender in Windows Server, see `https://docs.microsoft.com/windows/security/threat-protection/microsoft-defender-antivirus/microsoft-defender-antivirus-on-windows-server-2016`.

Testing any antivirus or antimalware application can be difficult. You want to ensure that the product, Defender, in this case, is working. But at the same time, you don't want to infect a server. One solution is to create a test file. The European Institute for Computer Anti-Virus Research (EICAR) has created a simple set of test files you can use to ensure your antivirus product works. EICAR has created several versions of this file, including both a text file and an executable. These files are harmless but, as you see, trigger Defender.

Getting ready

You run this recipe on `DC1`, a domain controller in the `Reskit.Org` domain.

How to do it...

1. Ensuring Defender and associated tools are installed

```
$DHT = @{
  Name                  =  'Windows-Defender'
  IncludeManagementTools = $true
}
$Defender = Install-WindowsFeature @DHT
If ($Defender.RestartNeeded -eq 'Yes') {
  Restart-Computer
}
```

2. Discovering the cmdlets in the Defender module

```
Import-Module -Name Defender
Get-Command -Module Defender
```

3. Checking the Defender service status

```
Get-Service  -Name WinDefend
```

4. Checking the operational status of Defender on this host

```
Get-MpComputerStatus
```

5. Getting and counting threat catalog

```
$ThreatCatalog = Get-MpThreatCatalog
"There are $($ThreatCatalog.count) threats in the catalog"
```

6. Viewing five threats in the catalog

```
$ThreatCatalog |
  Select-Object -First 5 |
    Format-Table -Property SeverityID, ThreatID, ThreatName
```

7. Setting key settings

```
# Enable real-time monitoring
Set-MpPreference -DisableRealtimeMonitoring 0
# Enable Cloud-DeliveredProtection
Set-MpPreference -MAPSReporting Advanced
# Enable sample submission
Set-MpPreference -SubmitSamplesConsent Always
# Enable checking signatures before scanning
Set-MpPreference -CheckForSignaturesBeforeRunningScan 1
# Enable email scanning
Set-MpPreference -DisableEmailScanning 0
```

8. Creating a false positive threat

```
$TF = 'C:\Foo\FalsePositive1.Txt'
$FP = ‹X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-› +
      'STANDARD-ANTIVIRUS-TEST-FILE!$H+H*'
$FP | Out-File -FilePath $TF
Get-Content -Path $TF
```

9. Running a quick scan on `C:\Foo`

```
$ScanType = 'QuickScan'
Start-MpScan -ScanType $ScanType -ScanPath C:\Foo
```

10. Viewing detected threats

```
Get-MpThreat
```

How it works...

In *step 1*, you use the `Install-WindowsFeature` command to ensure that you have installed both Defender and the tools you can use to manage Defender. This step may require a reboot. If so, this step reboots `DC1` without producing any output.

In *step 2*, you look at the Defender module to discover the cmdlets contained in the module. The output looks like this:

```
PS C:\Foo> # 2. Discovering the cmdlets in the Defender module
PS C:\Foo> Import-Module -Name Defender
PS C:\Foo> Get-Command -Module Defender

CommandType  Name                    Version  Source
-----------  ----                    -------  ------
Function     Add-MpPreference        1.0      Defender
Function     Get-MpComputerStatus    1.0      Defender
Function     Get-MpPreference        1.0      Defender
Function     Get-MpThreat            1.0      Defender
Function     Get-MpThreatCatalog     1.0      Defender
Function     Get-MpThreatDetection   1.0      Defender
Function     Remove-MpPreference     1.0      Defender
Function     Remove-MpThreat         1.0      Defender
Function     Set-MpPreference        1.0      Defender
Function     Start-MpScan            1.0      Defender
Function     Start-MpWDOScan         1.0      Defender
Function     Update-MpSignature      1.0      Defender
```

Figure 8.41: Discovering the cmdlets in the Defender module

In *step 3*, you check the status of the `WinDefend` service. You should see the following output:

```
PS C:\Foo> # 3. Checking the Defender service status
PS C:\Foo> Get-Service  -Name WinDefend

Status   Name       DisplayName
------   ----       -----------
Running  WinDefend  Microsoft Defender Antivirus Service
```

Figure 8.42: Checking the Defender service status

You use the `Get-MpComputerstatus` cmdlet to get the status of Defender on the local computer in *step 4*. The output looks like this:

```
PS C:\Foo> # 4.Checking the operational status of Defender on this host
PS C:\Foo> Get-MpComputerStatus

AMEngineVersion                 : 1.1.17800.5
AMProductVersion                : 4.18.2101.4
AMRunningMode                   : Normal
AMServiceEnabled                : True
AMServiceVersion                : 4.18.2101.4
AntispywareEnabled              : True
AntispywareSignatureAge         : 0
AntispywareSignatureLastUpdated : 29/01/2021 22:30:02
AntispywareSignatureVersion     : 1.329.3130.0
AntivirusEnabled                : True
AntivirusSignatureAge           : 0
AntivirusSignatureLastUpdated   : 29/01/2021 22:30:02
AntivirusSignatureVersion       : 1.329.3130.0
BehaviorMonitorEnabled          : True
ComputerID                      : E2B5FDA9-B412-494E-994A-5DA21BC8E0A4
ComputerState                   : 0
FullScanAge                     : 0
FullScanEndTime                 : 29/01/2021 21:33:50
FullScanStartTime               : 29/01/2021 21:21:40
IoavProtectionEnabled           : True
IsTamperProtected               : False
IsVirtualMachine                : True
LastFullScanSource              : 1
LastQuickScanSource             : 2
NISEnabled                      : True
NISEngineVersion                : 1.1.17800.5
NISSignatureAge                 : 0
NISSignatureLastUpdated         : 29/01/2021 22:30:02
NISSignatureVersion             : 1.329.3130.0
OnAccessProtectionEnabled       : True
QuickScanAge                    : 0
QuickScanEndTime                : 30/01/2021 03:07:26
QuickScanStartTime              : 30/01/2021 03:07:06
RealTimeProtectionEnabled       : True
RealTimeScanDirection           : 0
PSComputerName                  :
```

Figure 8.43: Checking the Defender status on the host

Defender uses details of individual threats that it stores in a threat catalog. Windows Update regularly updates this catalog as needed. In *step 5*, you produce a count of the number of threats in the catalog, which looks like this:

```
PS C:\Foo> # 5. Getting and counting threat catalog
PS C:\Foo> $ThreatCatalog = Get-MpThreatCatalog
PS C:\Foo> "There are $($ThreatCatalog.count) threats in the catalog"
There are 201761 threats in the catalog
```

Figure 8.44: Getting and counting the threat catalog

In *step 6*, you examine the first five threats in the Defender threat catalog, which looks like this:

```
PS C:\Foo> # 6. Viewing five threats in the catalog
PS C:\Foo> $ThreatCatalog |
             Select-Object -First 5 |
               Format-Table -Property SeverityID, ThreatID, ThreatName

SeverityID ThreatID ThreatName
---------- -------- ----------
         5     1596 TrojanSpy:Win32/AcidBattery
         5     1600 TrojanDownloader:Win32/AcidReign
         5     1604 Backdoor:Win32/AckCmd
         2     1605 Dialer:Win32/Aconti
         5     1622 MonitoringTool:Win32/ActiveKeylogger
```

Figure 8.45: Viewing the first five threats in the catalog

In *step 7*, you configure four important Defender settings. You can use the `Set-MpPreference` cmdlet to configure a range of preference settings for Windows Defender scans and updates. You can modify exclusion file name extensions, paths, or processes, and specify the default action for high, moderate, and low threat levels. You can view more details at `https://docs.microsoft.com/powershell/module/defender/set-mppreference`.

In *step 8*, you attempt to create a file that Defender regards as a threat. This file comes from the EICAR and as you can see is a benign text file. When you run this step, you get no output, although you may notice a Defender popup warning you that it has discovered a threat. If you run this step from the PowerShell console, you should get an error message.

In *step 9*, you run a quick scan on the `C:\Foo` folder where you attempted to create the test threat file. This step also produces no output.

In *step 10*, you view all the threats detected by Defender, which looks like this:

```
PS C:\Foo> # 10. Viewing detected threats
PS C:\Foo> Get-MpThreat


CategoryID        : 42
DidThreatExecute  : False
IsActive          : False
Resources         : {file:_C:\Foo\FalsePositive1.txt}
RollupStatus      : 33
SchemaVersion     : 1.0.0.0
SeverityID        : 5
ThreatID          : 2147519003
ThreatName        : Virus:DOS/EICAR_Test_File
TypeID            : 0
PSComputerName    :
```

Figure 8.46: Viewing all detected threats

There's more...

In *step 5*, you get a count of the number of threats of which Defender is aware as of the time of writing. When you run this step, you should see a higher number, reflecting newly discovered threats.

With *step 8*, you attempt to create a file that Defender recognizes as a threat. This file is the EICAR test file, which is harmless, but you can use it to test the basic functioning of Defender. In *step 10*, you view the threats Defender detected, and you can see it is the file identified as `EICAR_Test_File`.

9

Managing Storage

In this chapter, we cover the following recipes:

- Managing physical disks and volumes
- Managing filesystems
- Exploring providers and the FileSystem provider
- Managing Storage Replica
- Deploying Storage Spaces

Introduction

Windows Server 2022 provides a range of features that allows access to a wide variety of storage and storage devices. Windows supports spinning disks, USB memory sticks, and SSD devices (including NVMe SSD devices).

Before you can use a disk to hold files, you need to create partitions or volumes on the device and format them. When you first initialize a disk, you need to define which partitioning method to use. You have two choices:

- **Master Boot Record** (**MBR**)
- **GUID Partition Table** (**GPT**)

These days, most PCs use the GPT disk type for hard drives and SSDs. GPT is more robust and allows for volumes bigger than 2 TB. The older MBR disk type is used typically by 32-bit PCs, older PCs, and removable drives, such as memory cards or external disk drives.

For a good discussion of the differences between these two mechanisms, see `https://www.howtogeek.com/193669/whats-the-difference-between-gptand-mbr-when-partitioning-a-drive/`.

With a volume created, you can then format the disk volume. Windows supports five filesystems you can use: ReFS, NTFS, exFAT, UDF, and FAT32. For details of the latter four, refer to `https://docs.microsoft.com/windows/desktop/fileio/filesystem-functionality-comparison`. The ReFS filesystem is more recent and is based on NTFS, but lacks some features a file server might need (it has no encrypted files). A benefit of the ReFS filesystem is the automatic integrity checking. For a comparison between the ReFS and NTFS filesystems, see `https://www.iperiusbackup.net/en/refs-vs-ntfs-differences-and-performance-comparison-when-to-use/`. You examine partitioning and formatting volumes in the *Managing physical disks and volumes* recipe.

NTFS (and ReFS) volumes allow you to create **access control lists** (**ACLs**) that control access to files and folders stored in Windows volumes. Managing ACLs is somewhat tricky, and PowerShell itself lacks rich support for managing ACLs and ACL inheritance. To manage ACLs on NTFS volumes, as you see in the *Managing NTFS file and folder permissions* recipe in *Chapter 10*, *Managing Shared Data*, you can download and use a third-party module, `NTFSSecurity`.

Storage Replica is a feature of Windows Server (Datacenter only) that creates a disk on a remote systems, and have Windows keep the replica up to date. *Managing Storage Replica*, you create a replication partnership between two hosts and enable Windows Server to keep the replica up to date.

Storage Spaces is a technology provided by Microsoft in Window 10 and Windows Server that can help you protect against a disk drive's failure. In effect, Storage Spaces provides software RAID, which you investigate in *Deploying Storage Spaces*.

Managing physical disks and volumes

Windows Server 2022 requires a computer with at least one disk drive (in most cases, this is your `C:\` drive). You can connect a disk drive via different bus types, such as IDE, SATA, SAS, or USB. Before you can utilize a disk in Windows, you need to initialize it and create volumes or partitions.

You can use two partitioning schemes: the older format of MBR and the more recent GPT. The MBR scheme, first introduced with the PC DOS 2 in 1983, had some restrictions. For example, the largest partition supported with MBR is just 2 TB. And creating more than four partitions requires you to create an extended partition and create additional partitions inside the extended partition. The GPT scheme enables much larger drives (partition limits are OS-imposed) and up to 128 partitions per drive. You typically use GPT partitioning with Windows Server.

In this chapter, you use eight new virtual disk devices in the server, `SRV1` and create/use new volumes/partitions on those disks. At the start of this recipe, you add all eight virtual disks to the `SRV1` VM. In the recipe itself, you use just the first two of these new disks.

Getting ready

This recipe uses SRV1, a domain-joined host in the Reskit.Org domain, on which you have installed PowerShell 7 and VS Code. You also use SRV2 and should have DC1 online.

The recipes in this chapter make use of eight additional virtual disks. You can run the following script on the Hyper-V host to add the new disks to the SRV1 and SRV2 VMs:

```
# 0. Add new disks to the SRV1, SRV2 VMs
# Run on VM host
# 0.1 Turning off the VMs
Get-VM -Name SRV1, SRV2 | Stop-VM -Force
# 0.2 Getting Path for hard disks for SRV1, SRV2
$Path1   = Get-VMHardDiskDrive -VMName SRV1
$Path2   = Get-VMHardDiskDrive -VMName SRV2
$VMPath1 = Split-Path -Parent $Path1.Path
$VMPath2 = Split-Path -Parent $Path2.Path
# 0.3 Creating 8 disks to SRV1/2 for storage chapter
0..7 | ForEach-Object {
  New-VHD -Path $VMPath1\SRV1-D$_.vhdx -SizeBytes 64gb -Dynamic | Out-Null
  New-VHD -Path $VMPath2\SRV2-D$_.vhdx -SizeBytes 64gb -Dynamic | Out-Null
}
# 0.4 Adding disks to SRV1, SRV2
0..7 | ForEach-Object {
  $DHT1 = @{
    VMName           = 'SRV1'
    Path             = "$VMPath1\SRV1-D$_.vhdx"
    ControllerType   = 'SCSI'
    ControllerNumber = 0
  }
  $DHT2 = @{
    VMName           = 'SRV2'
    Path             =  "$VMPath2\SRV2-D$_.vhdx"
    ControllerType   = 'SCSI'
    ControllerNumber =  0
  }
  Add-VMHardDiskDrive $DHT1
  Add-VMHardDiskDrive $DHT2
}
# 0.5 Checking VM disks for SRV1, SRV2
Get-VMHardDiskDrive -VMName SRV1 | Format-Table
Get-VMHardDiskDrive -VMName SRV2 | Format-Table
# 0.6 Restarting VMs
Start-VM -VMname SRV1
Start-VM -VMName SRV2
```

Once you have created the eight new disks for the two VMs, you can begin the recipe on SRV1.

How to do it...

1. Getting the first new physical disk on SRV1

```
$Disks = Get-Disk |
           Where-Object PartitionStyle -eq Raw |
             Select-Object -First 1
$Disks | Format-Table -AutoSize
```

2. Initializing the first disk

```
$Disks |
  Where-Object PartitionStyle -eq Raw |
    Initialize-Disk -PartitionStyle GPT
```

3. Re-displaying all disks in SRV1

```
Get-Disk |
  Format-Table -AutoSize
```

4. Viewing volumes on SRV1

```
Get-Volume | Sort-Object -Property DriveLetter
```

5. Creating an F: volume in disk 1

```
$NVHT1 = @{
  DiskNumber   =  $Disks[0].DiskNumber
  FriendlyName = 'Files'
  FileSystem   = 'NTFS'
  DriveLetter  = 'F'
}
New-Volume @NVHT1
```

6. Creating two partitions in disk 2 – first create S volume

```
Initialize-Disk -Number 2 -PartitionStyle MBR
New-Partition -DiskNumber 2  -DriveLetter S -Size 32gb
```

7. Creating a second partition T on disk 2

```
New-Partition -DiskNumber 2  -DriveLetter T -UseMaximumSize
```

8. Formatting `S:` and `T:`

```
$NVHT1 = @{
  DriveLetter        = 's'
  FileSystem         = 'NTFS'
  NewFileSystemLabel = 'GD Shows'}
Format-Volume @NVHT1
$NVHT2 = @{
  DriveLetter        = 'T'
  FileSystem         = 'FAT32'
  NewFileSystemLabel = 'GD Pictures'}
Format-Volume @NVHT2
```

9. Getting partitions on `SRV1`

```
Get-Partition  |
  Sort-Object -Property DriveLetter |
    Format-Table -Property DriveLetter, Size, Type, *name
```

10. Getting volumes on `SRV1`

```
Get-Volume |
  Sort-Object -Property DriveLetter
```

11. Viewing disks in `SRV1`

```
Get-Disk | Format-Table
```

How it works...

In *step 1*, you use the `Get-Disk` cmdlet to get the first virtual disk in the `SRV1` VM, one of the eight new disks you added previously. The output of this step looks like this:

```
PS C:\Foo> # 1. Getting the first new physical disk on SRV1
PS C:\Foo> $Disks = Get-Disk |
           Where-Object PartitionStyle -eq Raw |
             Select-Object -First 1
PS C:\Foo> $Disks | Format-Table -AutoSize

Number Friendly Name     Serial Number HealthStatus OperationalStatus Total Size Partition Style
------ -------------     ------------- ------------ ----------------- ---------- ---------------
1      Msft Virtual Disk               Healthy      Offline                64 GB RAW
```

Figure 9.1: Getting the first new physical disk on SRV1

In *step 2*, you initialize the disk (disk number 1), which generates no output. Then, in *step 3*, you view all the disks in `SRV1`, with output that looks like this:

```
PS C:\Foo> # 3. Re-displaying all disks in SRV1
PS C:\Foo> Get-Disk |
             Format-Table -AutoSize

Number Friendly Name     Serial Number HealthStatus OperationalStatus Total Size Partition Style
------ -------------     ------------- ------------ ----------------- ---------- ----------------
0      Virtual HD                      Healthy      Online                128 GB MBR
1      Msft Virtual Disk               Healthy      Online                 64 GB GPT
2      Msft Virtual Disk               Healthy      Offline                64 GB RAW
3      Msft Virtual Disk               Healthy      Offline                64 GB RAW
4      Msft Virtual Disk               Healthy      Offline                64 GB RAW
5      Msft Virtual Disk               Healthy      Offline                64 GB RAW
6      Msft Virtual Disk               Healthy      Offline                64 GB RAW
7      Msft Virtual Disk               Healthy      Offline                64 GB RAW
8      Msft Virtual Disk               Healthy      Offline                64 GB RAW
```

Figure 9.2: Viewing all the disks in SRV1

In *step 4*, you use the `Get-Volume` command to get the available volumes on `SRV1`, with output like this:

```
PS C:\Foo> # 4. Viewing volumes on SRV1
PS C:\Foo> Get-Volume | Sort-Object -Property DriveLetter

DriveLetter FriendlyName         FileSystemType DriveType HealthStatus OperationalStatus SizeRemaining      Size
----------- ------------         -------------- --------- ------------ ----------------- -------------      ----
                                 NTFS           Fixed     Healthy      OK                     80.59 MB    496 MB
A                                Unknown        Removable Healthy      Unknown                 0 B         0 B
C                                NTFS           Fixed     Healthy      OK                    110.06 GB 127.51 GB
D           SSS_X64FRE_EN-US_DV9 Unknown        CD-ROM    Healthy      OK                      0 B     4.38 GB
```

Figure 9.3: Viewing volumes in SRV1

With *step 5*, you create a new volume, and format it with the NTFS filesystem. The output from this step looks like this:

```
PS C:\Foo> # 5. Creating a F: volume in disk 1
PS C:\Foo> Clear-Disk -Number $Disks[0].DiskNumber -Confirm:$false -RemoveData
PS C:\Foo> $NVHT1 = @{
             DiskNumber   =  $Disks[0].DiskNumber
             FriendlyName = 'Files'
             FileSystem   = 'NTFS'
             DriveLetter  = 'F'
           }
PS C:\Foo> New-Volume @NVHT1

DriveLetter FriendlyName FileSystemType DriveType HealthStatus OperationalStatus SizeRemaining      Size
----------- ------------ -------------- --------- ------------ ----------------- -------------      ----
F           Files        NTFS           Fixed     Healthy      OK                    63.89 GB 63.98 GB
```

Figure 9.4: Creating an F: volume

In *step 6*, you initialize disk 2 and create a 32 GB partition with the `S:` drive. The output of this step looks like this:

```
PS C:\Foo> # 6. Creating two partitions in disk 2 - first create S volume
PS C:\Foo> Initialize-Disk -Number 2 -PartitionStyle MBR
PS C:\Foo> New-Partition -DiskNumber 2  -DriveLetter S -Size 32gb

   DiskPath: \\?\scsi#disk&ven_msft&prod_virtual_disk#6&fa491f9&0&000001#{53f56307-b6bf-11d0-94f2-00a0c91efb8b}

PartitionNumber  DriveLetter Offset    Size Type
---------------  ----------- ------    ---- ----
1                S           1048576  32 GB Logical
```

Figure 9.5: Creating an S: partition

In *step 7*, you create a second partition on disk 2, with output like this:

```
PS C:\Foo> # 7. Creating a second partition T on disk 2
PS C:\Foo> New-Partition -DiskNumber 2  -DriveLetter T -UseMaximumSize

   DiskPath: \\?\scsi#disk&ven_msft&prod_virtual_disk#6&fa491f9&0&000002#{53f56307-b6bf-11d0-94f2-00a0c91efb8b}

PartitionNumber  DriveLetter Offset        Size Type
---------------  ----------- ------        ---- ----
2                T           343607786944  32 GB Logical
```

Figure 9.6: Creating a T: partition

In *step 8*, you format the newly created `S:` and `T:` drives. You use the NTFS filesystem for the `S:` drive, and the FAT32 filesystem for the `T:` drive, with output like this:

```
PS C:\Foo> # 8. Formatting S: and T:
PS C:\Foo> $NVHT1 = @{
             DriveLetter        = 'S'
             FileSystem         = 'NTFS'
             NewFileSystemLabel = 'GD Shows'}

PS C:\Foo> Format-Volume @NVHT1
DriveLetter FriendlyName FileSystemType DriveType HealthStatus OperationalStatus SizeRemaining  Size
----------- ------------ -------------- --------- ------------ ----------------- -------------  ----
S           GD Shows     NTFS           Fixed     Healthy      OK                    31.93 GB 32 GB

PS C:\Foo> $NVHT2 = @{
             DriveLetter        = 'T'
             FileSystem         = 'FAT32'
             NewFileSystemLabel = 'GD Pictures'}
PS C:\Foo> Format-Volume @NVHT2

DriveLetter FriendlyName    FileSystemType DriveType HealthStatus OperationalStatus SizeRemaining      Size
----------- ------------    -------------- --------- ------------ ----------------- -------------      ----
T           GD PICTURES     FAT32          Fixed     Healthy      OK                   31.983 GB 31.98 GB
```

Figure 9.7: Formatting the S: and T: drives

In *step 9*, you view the partitions available on `SRV1`, with output like this:

```
PS C:\Foo> # 9. Getting partitions on SRV1
PS C:\Foo> Get-Partition  |
             Sort-Object -Property DriveLetter |
               Format-Table -Property DriveLetter, Size, Type

DriveLetter         Size Type
-----------         ---- ----
                520093696 Unknown
                 16759808 Reserved
                 16759808 Reserved
            C 136915714048 IFS
            F  687016509944 Basic
            S  34359738368 IFS
            T  34357641216 FAT32 XINT13
```

Figure 9.8: Getting partitions on SRV1

In *step 10*, you view the volumes on `SRV1` using the `Get-Volume` command, with output like this:

```
PS C:\Foo> # 10. Getting volumes on SRV1
PS C:\Foo> Get-Volume |
             Sort-Object -Property DriveLetter

DriveLetter FriendlyName         FileSystemType DriveType HealthStatus OperationalStatus SizeRemaining      Size
----------- ------------         -------------- --------- ------------ ----------------- -------------      ----
                                 NTFS           Fixed     Healthy      OK                    80.59 MB    496 MB
A                                Unknown        Removable Healthy      Unknown                0 B        0 B
C                                NTFS           Fixed     Healthy      OK                   110.14 GB 127.51 GB
D           SSS_X64FRE_EN-US_DV9 Unknown        CD-ROM    Healthy      OK                     0 B      4.38 GB
F           Files                NTFS           Fixed     Healthy      OK                    63.89 GB  63.98 GB
S           GD Shows             NTFS           Fixed     Healthy      OK                    31.93 GB     32 GB
T           GD PICTURES          FAT32          Fixed     Healthy      OK                    31.98 GB  31.98 GB
```

Figure 9.9: Getting volumes on SRV1

In the final step in this recipe, *step 11*, you view all the disks in `SRV1`, with output like this:

```
PS C:\Foo> # 11. Viewing disks in SRV1
PS C:\Foo> Get-Disk | Format-Table

Number Friendly Name     Serial Number HealthStatus OperationalStatus Total Size Partition Style
------ -------------     ------------- ------------ ----------------- ---------- ---------------
0      Virtual HD                      Healthy      Online                128 GB MBR
1      Msft Virtual Disk               Healthy      Online                 64 GB GPT
2      Msft Virtual Disk               Healthy      Online                 64 GB MBR
3      Msft Virtual Disk               Healthy      Offline                64 GB RAW
4      Msft Virtual Disk               Healthy      Offline                64 GB RAW
5      Msft Virtual Disk               Healthy      Offline                64 GB RAW
6      Msft Virtual Disk               Healthy      Offline                64 GB RAW
7      Msft Virtual Disk               Healthy      Offline                64 GB RAW
8      Msft Virtual Disk               Healthy      Offline                64 GB RAW
```

Figure 9.10: Viewing all the disks in SRV1

There's more...

With Windows (Windows 10 and Windows Server 2022), you create usable data drives using either the `New-Volume` or `New-Partition` commands. The `New-Volume` cmdlet, shown in *step 5*, creates a partition on the specified disk, formats the partition with a filesystem, and gives it a label and a drive letter. In *step 6*, you initialize the disk with the MBR formatting and create a new partition in the disk. You create a second partition in *step 7* and format the two partitions in *step 8*.

In *step 10*, you view the volumes on `SRV1`. Notice that in *step 8*, you specified a volume label for the `T:` volume with uppercase and lowercase letters. However, Windows converts this friendly name to all uppercase when you create the partition. This is by design.

Managing filesystems

To make use of a "disk" device, whether a spinning disk, CD/DVD device, or a solid-state device, you must format that device/drive with a filesystem. In Windows, in addition to allowing you to specify which specific filesystem to use, you can also give the partition a drive letter and filesystem label while formatting the drive.

In most cases, you use NTFS as your filesystem of choice. It is robust and reliable and provides efficient access control as well as providing encryption and compression. ReFS might be a good choice for some specialized workloads, particularly on a physical Hyper-V host where you might use the ReFS filesystem on disks you use to hold your VMs' virtual hard drives. For interoperability with things like video and still cameras, you may need to use the FAT, FAT32, or exFAT filesystems.

For more details on the difference between NTFS, FAT, FAT32, and ExFAT filesystems, see `https://medium.com/hetman-software/the-difference-between-ntfs-fat-fat32-and-exfat-file-systems-ec5172c60ccd`. And for more details on the ReFS filesystem, see `https://docs.microsoft.com/windows-server/storage/refs/refs-overview`.

Getting ready

This recipe uses `SRV1`, a domain-joined host in the `Reskit.Org` domain, on which you have installed PowerShell 7 and VS Code. You also have `DC1` online. In the *Managing physical disks and volumes* recipe, you added eight virtual disks to the `SRV1` VM and used the first two. In this recipe, you use the third of those disks.

How to do it...

1. Getting disk to use on SRV1

```
$Disk = Get-Disk |
          Where-Object PartitionStyle -eq 'RAW' |
            Select-Object -First 1
```

2. Viewing disk

```
$Disk | Format-List
```

3. Viewing partitions on the disk

```
$Disk | Get-Partition
```

4. Initializing this disk and creating 4 partitions

```
Initialize-Disk -Number $Disk.DiskNumber -PartitionStyle GPT
New-Partition -DiskNumber $Disk.DiskNumber  -DriveLetter W -Size 15gb
New-Partition -DiskNumber $Disk.DiskNumber  -DriveLetter X -Size 15gb
New-Partition -DiskNumber $Disk.DiskNumber  -DriveLetter Y -Size 15gb
$UMHT= @{UseMaximumSize = $true}
New-Partition -DiskNumber $Disk.DiskNumber  -DriveLetter Z @UMHT
Formatting each partition

$FHT1 = @{
    DriveLetter        = 'W'
    FileSystem         = 'FAT'
    NewFileSystemLabel = 'w-fat'
}
Format-Volume @FHT1
$FHT2 = @{
    DriveLetter        = 'X'
    FileSystem         = 'exFAT'
    NewFileSystemLabel = 'x-exFAT'
}
Format-Volume @FHT2
$FHT3 = @{
    DriveLetter        = 'Y'
    FileSystem         = 'FAT32'
    NewFileSystemLabel = 'Y-FAT32'
}
Format-Volume  @FHT3
$FHT4 = @{
    DriveLetter        = 'Z'
    FileSystem         = 'ReFS'
    NewFileSystemLabel = 'Z-ReFS'
```

```
}
Format-Volume @FHT4
```

5. Getting volumes on SRV1

```
Get-Volume | Sort-Object DriveLetter
```

How it works...

In *step 1*, you get the first unused disk in SRV1 and store it in the $Disk variable. This step produces no output. In *step 2*, you view the disk object, with output like this:

```
PS C:\Foo> # 2. Viewing disk
PS C:\Foo> $Disk | Format-List

UniqueId           : 600224806AD90C2695DB591D9D0FC623
Number             : 3
Path               : \\?\scsi#disk&ven_msft&prod_virtual_disk#6&fa491f9&0&000002#{53f56307-b6bf-11d0-94f2-00a0c91efb8b}
Manufacturer       : Msft
Model              : Virtual Disk
SerialNumber       :
Size               : 64 GB
AllocatedSize      : 0
LogicalSectorSize  : 512
PhysicalSectorSize : 4096
NumberOfPartitions : 0
PartitionStyle     : RAW
IsReadOnly         : False
IsSystem           : False
IsBoot             : False
```

Figure 9.11: Viewing disk 3

In *step 3*, you view the partitions on disk 3. Since there are no partitions (yet) on this disk drive, this step produces no output.

In *step 4*, you create four partitions on disk 3. The first three occupy 15 GB each, with the fourth taking up all the remaining space in the disk. The output of this step looks like this:

```
PS C:\Foo> # 4. Initializing this disk amd creating 4 partitions
PS C:\Foo> New-Partition -DiskNumber $Disk.DiskNumber  -DriveLetter W -Size 15gb
PS C:\Foo> New-Partition -DiskNumber $Disk.DiskNumber  -DriveLetter X -Size 15gb
PS C:\Foo> New-Partition -DiskNumber $Disk.DiskNumber  -DriveLetter Y -Size 15gb
PS C:\Foo> $UMHT= @{UseMaximumSize = $true}
PS C:\Foo> New-Partition -DiskNumber $Disk.DiskNumber  -DriveLetter Z @UMHT

   DiskPath: \\?\scsi#disk&ven_msft&prod_virtual_disk#6&fa491f9&0&000002#{53f56307-b6bf-11d0-94f2-00a0c91efb8b}

PartitionNumber  DriveLetter Offset             Size Type
---------------  ----------- ------             ---- ----
2                W           16777216         15 GB Basic
3                X           16122904576      15 GB Basic
4                Y           32229031936      15 GB Basic
5                Z           48335159296   18.98 GB Basic
```

Figure 9.12: Creating partitions on disk 3

In *step 5*, you format each partition in disk 3 using a different filesystem. This includes attempting to create a format for the `W:` drive with the FAT filesystem, which generates an error since the largest partition you can use with FAT is 4 GB. The output looks like this:

```
PS C:\Foo> # 5. Formatting each partition
PS C:\Foo> $FHT1 = @{
             DriveLetter        = 'W'
             FileSystem         = 'FAT'
             NewFileSystemLabel = 'w-fat'
           }
PS C:\Foo> Format-Volume @FHT1
PS C:\Foo> $FHT2 = @{
             DriveLetter        = 'X'
             FileSystem         = 'exFAT'
             NewFileSystemLabel = 'x-exFAT'
           }
PS C:\Foo> Format-Volume @FHT2
PS C:\Foo> $FHT3 = @{
             DriveLetter        = 'Y'
             FileSystem         = 'FAT32'
             NewFileSystemLabel = 'Y-FAT32'
           }
PS C:\Foo> Format-Volume  @FHT3
PS C:\Foo> $FHT4 = @{
             DriveLetter        = 'Z'
             FileSystem         = 'ReFS'
             NewFileSystemLabel = 'Z-ReFS'
           }
PS C:\Foo> Format-Volume @FHT4
Format-Volume:
Line |
   7 |  Format-Volume @FHT1
     |  ~~~~~~~~~~~~~~~~~~~
     | Size Not Supported
Activity ID: {a0a18533-2fa2-4de3-93c8-a295c731873e}

DriveLetter FriendlyName FileSystemType DriveType HealthStatus OperationalStatus SizeRemaining     Size
----------- ------------ -------------- --------- ------------ ----------------- -------------     ----
X           x-exFAT      exFAT          Fixed     Healthy      OK                       15 GB    15 GB
Y           Y-FAT32      FAT32          Fixed     Healthy      OK                 14.98 GB 14.98 GB
Z           Z-ReFS       ReFS           Fixed     Healthy      OK                 17.87 GB 18.94 GB
```

Figure 9.13: Formatting partitions on disk 3

In *step 6*, you use the `Get-Volume` command to get all the disk volumes in `SRV1`, which looks like this:

```
PS C:\Foo> # 6. Getting volumes on SRV1
PS C:\Foo> Get-Volume | Sort-Object DriveLetter

DriveLetter FriendlyName         FileSystemType DriveType HealthStatus OperationalStatus SizeRemaining      Size
----------- ------------         -------------- --------- ------------ ----------------- -------------      ----
                                 NTFS           Fixed     Healthy      OK                     80.59 MB    496 MB
A                                Unknown        Removable Healthy      Unknown                   0 B      0 B
C                                NTFS           Fixed     Healthy      OK                   110.33 GB 127.51 GB
D           SSS_X64FRE_EN-US_DV9 Unknown        CD-ROM    Healthy      OK                        0 B   4.38 GB
F           Files                NTFS           Fixed     Healthy      OK                    63.89 GB  63.98 GB
S           GD Shows             NTFS           Fixed     Healthy      OK                    31.93 GB     32 GB
T           GD PICTURES          FAT32          Fixed     Healthy      OK                    31.98 GB  31.98 GB
W                                Unknown        Fixed     Healthy      Unknown                   0 B      0 B
X           x-exfat              exFAT          Fixed     Healthy      OK                       15 GB    15 GB
Y           Y-FAT32              FAT32          Fixed     Healthy      OK                    14.98 GB  14.98 GB
Z           z-refs               ReFS           Fixed     Healthy      OK                    17.87 GB  18.94 GB
```

Figure 9.14: Viewing all volumes on SRV1

There's more...

In *step 5*, you attempt to format the `W:` partition with the FAT filesystem. You created this partition as 15 GB, which is too big for FAT, hence the error message you see. The other three partitions are successfully formatted. Notice that the friendly name for the `Y:` partition is in uppercase, although you supplied a friendly name with uppercase and lowercase characters. `Format-Volume` converts the name to all uppercase when formatting the partition with the FAT32 filesystem.

Exploring providers and the FileSystem provider

One innovation in PowerShell that IT professionals soon learn to love is the PowerShell provider. A provider is a component that provides access to specialized data stores for easy management. The provider makes the data appear in a drive with a path similar to how you access files in file stores.

PowerShell 7.1 comes with the following providers:

- **Registry**: Provides access to registry keys and registry values (`https://docs.microsoft.com/powershell/module/microsoft.powershell.core/about/about_registry_provider?view=powershell-7.1`)
- **Alias**: Provides access to PowerShell's command aliases (`https://docs.microsoft.com/powershell/module/microsoft.powershell.core/about/about_alias_provider?view=powershell-7.1`)
- **Environment**: Provides access to Windows environment variables (`https://docs.microsoft.com/powershell/module/microsoft.powershell.core/about/about_environment_provider?view=powershell-7.1`)
- **FileSystem**: Provides access to files in a partition (`https://docs.microsoft.com/powershell/module/microsoft.powershell.core/about/about_filesystem_provider?view=powershell-7.1`)
- **Function**: Provides access to PowerShell's function definitions (`https://docs.microsoft.com/powershell/module/microsoft.powershell.core/about/about_function_provider?view=powershell-7.1`)
- **Variable**: Provides access to PowerShell's variables (`https://docs.microsoft.com/powershell/module/microsoft.powershell.core/about/about_variable_provider?view=powershell-7.1`)
- **Certificate**: Provides access to the current user and local host's X.509 digital certificate stores (`https://docs.microsoft.com/powershell/module/microsoft.powershell.security/about/about_certificate_provider?view=powershell-7.1`)

- **WSMan**: Provides a configuration surface that configures the WinRM service (`https://docs.microsoft.com/powershell/module/microsoft.wsman.management/about/about_wsman_provider?view=powershell-7.1`)

With PowerShell providers, you do not need a set of cmdlets for each underlying data store. You can use `Get-Item` or `Get-ChildItem` with any provider to return provider-specific data, as you can see in this recipe.

Other applications can add providers to a given host. For example, the IIS administration module creates an `IIS:` drive. And if you have your own data stores, you can also create providers. The SHiPS module, available from the PowerShell Gallery, enables you to build a provider using PowerShell. As an example of the SHiPS platform's capabilities, you can use a sample provider from the `CimPSDrive` module. This module contains a provider for the CIM repository. For more information on the SHiPS platform, see `https://github.com/PowerShell/SHiPS/tree/development/docs`. For more details on the CimPSDrive provider, see `https://github.com/PowerShell/CimPSDrive`.

Getting ready

This recipe uses `SRV1`, a domain-joined host in the `Reskit.Org` domain. You should have installed AD on this host and configured it as per earlier recipes in *Chapter 5*, *Exploring .NET*, and *Chapter 6*, *Managing Active Directory*. You used this server in previous recipes in this chapter.

How to do it...

1. Getting providers

   ```
   Get-PSProvider
   ```

2. Getting registry drives

   ```
   Get-PSDrive | Where-Object Provider -match 'registry'
   ```

3. Looking at a registry key

   ```
   $Path = 'HKLM:\SOFTWARE\Microsoft\Windows NT\CurrentVersion'
   Get-Item -Path $Path
   ```

4. Getting registered owner from the registry

   ```
   (Get-ItemProperty -Path $Path -Name RegisteredOwner).RegisteredOwner
   ```

5. Counting aliases in the `Alias:` drive

   ```
   Get-Item Alias:* | Measure-Object
   ```

6. Finding aliases for `Remove-Item`

```
Get-ChildItem Alias:* |
  Where-Object ResolvedCommand -match 'Remove-Item$'
```

7. Counting environment variables on `SRV1`

```
Get-Item ENV:* | Measure-Object
```

8. Displaying Windows installation folder

```
"Windows installation folder is [$env:windir]"
```

9. Checking on FileSystem provider drives on `SRV1`

```
Get-PSProvider -PSProvider FileSystem |
  Select-Object -ExpandProperty Drives |
    Sort-Object -Property Name
```

10. Getting home folder for FileSystem provider

```
$HF = Get-PSProvider -PSProvider FileSystem |
  Select-Object -ExpandProperty Home
"Home folder for SRV1 is [$HF]"
```

11. Checking `Function` drive

```
Get-Module | Remove-Module -WarningAction SilentlyContinue
$Functions = Get-ChildItem -Path Function:
"Functions available [$($Functions.Count)]"
```

12. Creating a new function

```
Function Get-HelloWorld {'Hello World'}
```

13. Checking `Function` drive

```
$Functions2 = Get-ChildItem -Path Function:
"Functions now available [$($Functions2.Count)]"
```

14. Viewing function definition

```
Get-Item Function:\Get-HelloWorld | Format-List *
```

15. Counting defined variables

```
$Variables = Get-ChildItem -Path Variable:
"Variables defined [$($Variables.count)]"
```

16. Checking on available functions

```
Get-Item Variable:Function*
```

17. Getting trusted root certificates for the local user

```
Get-ChildItem -Path Cert:\CurrentUser\TrustedPublisher
```

18. Examining ports in use by WinRM

```
Get-ChildItem -Path WSMan:\localhost\Client\DefaultPorts
Get-ChildItem -Path WSMan:\localhost\Service\DefaultPorts
```

19. Setting Trusted Hosts

```
Set-Item WSMan:\localhost\Client\TrustedHosts -Value '*' -Force
```

20. Installing SHiPS and CimPSDrive modules

```
Install-Module -Name SHiPS, CimPSDrive
```

21. Importing the CimPSDrive module and creating a drive

```
Import-Module -Name CimPSDrive
New-PSDrive -Name CIM -PSProvider SHiPS -Root CIMPSDrive#CMRoot
```

22. Examining BIOS information

```
Get-ChildItem CIM:\Localhost\CIMV2\Win32_Bios
```

How it works...

In *step 1*, you use the `Get-PSProvider` cmdlet to return the providers currently loaded in `SRV1`, with output like this:

```
PS C:\Foo> # 1. Getting providers
PS C:\Foo> Get-PSProvider

Name           Capabilities                       Drives
----           ------------                       ------
Registry       ShouldProcess                      {HKLM, HKCU}
Alias          ShouldProcess                      {Alias}
Environment    ShouldProcess                      {Env}
FileSystem     Filter, ShouldProcess, Credentials {C, F, S, T, X, Y, Z, Temp, A, D}
Function       ShouldProcess                      {Function}
Variable       ShouldProcess                      {Variable}
WSMan          Credentials                        {WSMan}
```

Figure 9.15: Viewing providers on SRV1

You can create a PowerShell provider drive within any provider (using `New-PSDrive`). To see the drives you have defined on `SRV1` that use the `Registry` provider, in *step 2*, you use the `Get-PSDrive` command with output like this:

```
PS C:\Foo> # 2. Getting registry drives
PS C:\Foo> Get-PSDrive | Where-Object Provider -match 'registry'

Name Used (GB)  Free (GB) Provider  Root                CurrentLocation
---- ---------  --------- --------  ----                ---------------
HKCU                      Registry  HKEY_CURRENT_USER
HKLM                      Registry  HKEY_LOCAL_MACHINE
```

Figure 9.16: Getting registry drives

In *step 3*, you use the Registry provider and the `HKLM:` drive to get details of the current version of Windows, with output like this:

```
PS C:\Foo> # 3. Looking at a registry key
PS C:\Foo> $Path = 'HKLM:\SOFTWARE\Microsoft\Windows NT\CurrentVersion'
PS C:\Foo> Get-Item -Path $Path

    Hive: HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT

Name             Property
----             --------
CurrentVersion   SystemRoot                : C:\WINDOWS
                 BaseBuildRevisionNumber   : 1
                 BuildBranch               : fe_release
                 BuildGUID                 : ffffffff-ffff-ffff-ffff-ffffffffffff
                 BuildLab                  : 20282.fe_release.210111-1506
                 BuildLabEx                : 20282.1.amd64fre.fe_release.210111-1506
                 CompositionEditionID      : ServerDatacenter
                 CurrentBuild              : 20282
                 CurrentBuildNumber        : 20282
                 CurrentMajorVersionNumber : 10
                 CurrentMinorVersionNumber : 0
                 CurrentType               : Multiprocessor Free
                 CurrentVersion            : 6.3
                 EditionID                 : ServerDatacenter
                 EditionSubManufacturer    :
                 EditionSubstring          :
                 EditionSubVersion         :
                 InstallationType          : Server
                 InstallDate               : 1610817552
                 ProductName               : Windows Server 2019 Datacenter
                 ReleaseId                 : 2004
                 SoftwareType              : System
                 UBR                       : 1
                 PathName                  : C:\Windows
                 PendingInstall            : 0
                 ProductId                 : 00133-32500-02427-AA928
                 DigitalProductId          : {164, 0, 0, 0, 3, 0, 0, 0, 48, 48, 49, 51, 51, 45, 51...
                 DigitalProductId4         : {248, 4, 0, 0, 4, 0, 0, 0, 48, 0, 51, 0, 54, 0, 49, 0...
                 RegisteredOrganization    : Reskit.Org
                 RegisteredOwner           : Packt book readers
                 InstallTime               : 132552911528167249
```

Figure 9.17: Examining a registry key

In *step 4*, you use the `Get-ItemProperty` cmdlet to get and display the registered owner of `SRV1`. Assuming that you have used the Reskit build scripts mentioned in the book's preface, the output of this step looks like this:

```
PS C:\Foo> # 4. Getting registered owner
PS C:\Foo> (Get-ItemProperty -Path $Path -Name RegisteredOwner).RegisteredOwner
Packt book readers
```

Figure 9.18: Viewing a registered owner on SRV2

By default, the `Alias:` drive contains all the aliases you define within a PowerShell session. In *step 5*, you retrieve and count the aliases in the `Alias:` drive, with output like this:

```
PS C:\Foo> # 5. Counting aliases in the Alias: drive
PS C:\Foo> Get-Item Alias:* | Measure-Object

Count             : 142
Average           :
Sum               :
Maximum           :
Minimum           :
StandardDeviation :
Property          :
```

Figure 9.19: Counting aliases on SRV1

You can use the `Alias:` drive to discover aliases for a particular command. In *step 6*, you discover the aliases defined for the `Remove-Item` cmdlet. The output of this step looks like this:

```
PS C:\Foo> # 6. Finding aliases for Remove-Item
PS C:\Foo> Get-ChildItem Alias:* |
             Where-Object ResolvedCommand -match 'Remove-Item$'

CommandType     Name                    Version    Source
-----------     ----                    -------    ------
Alias           ri -> Remove-Item
Alias           rm -> Remove-Item
Alias           rmdir -> Remove-Item
Alias           del -> Remove-Item
Alias           erase -> Remove-Item
Alias           rd -> Remove-Item
```

Figure 9.20: Finding aliases for Remove-Item

In *step 7*, you use the `Environment Variable` provider to count the number of environment variables currently defined, with output like this:

```
PS C:\Foo> # 7. Counting environment variables on SRV1
PS C:\Foo> Get-Item ENV:* | Measure-Object

Count             : 48
Average           :
Sum               :
Maximum           :
Minimum           :
StandardDeviation :
Property          :
```

Figure 9.21: Counting environment variables on SRV1

In *step 8*, you use the `ENV:` drive to get the environment variable holding the Windows installation folder (typically, `C:\Windows`, but you have options). The output of this step looks like this:

```
PS C:\Foo> # 8. Displaying Windows installation folder
PS C:\Foo> "Windows installation folder is [$env:windir]"
Windows installation folder is [C:\WINDOWS]
```

Figure 9.22: Getting the Windows installation folder

In *step 9*, you discover the FileSystem provider's drives. These drives, shown in the following output, include the drives (partitions/volumes) you created in earlier recipes and look like this:

```
PS C:\Foo> # 9. Checking on FileSystem provider drives on SRV1
PS C:\Foo> Get-PSProvider -PSProvider FileSystem |
             Select-Object -ExpandProperty Drives |
               Sort-Object -Property Name

Name   Used (GB)  Free (GB) Provider     Root                                CurrentLocation
----   ---------  --------- --------     ----                                ---------------
A                           FileSystem   A:\
C          17.14     110.37 FileSystem   C:\                                 Foo
D           4.38       0.00 FileSystem   D:\
F           0.09      63.89 FileSystem   F:\
S           0.07      31.93 FileSystem   S:\
T           0.00      31.98 FileSystem   T:\
Temp       17.14     110.37 FileSystem   C:\Users\Administrator.RESKIT\AppD…
X           0.00      15.00 FileSystem   X:\
Y           0.00      14.98 FileSystem   Y:\
Z           1.07      17.87 FileSystem   Z:\
```

Figure 9.23: Getting FileSystem provider drives

Each provider enables you to define a **home drive**. If you set a home drive, you can use the `Set-Location` command and specify a path of "~" to move to that home drive. You set the home drive for the filesystem provider in the `$Profile` file you set up in *Chapter 1*, *Installing and Configuring PowerShell*. In *step 10*, you get the home drive for the FileSystem provider. The output of this step looks like this:

```
PS C:\Foo> # 10. Getting home folder for FileSystem provider
PS C:\Foo> $HF = Get-PSProvider -PSProvider FileSystem |
             Select-Object -ExpandProperty Home
PS C:\Foo> "Home drive for SRV1 is [$HF]"
Home folder for SRV1 is [C:\Foo]
```

Figure 9.24: Getting the FileSystem provider home folder

In *step 11*, you remove all modules, and then get and count the functions in the `Function:` drive, with output like this:

```
PS C:\Foo> # 11. Checking Function drive
PS C:\Foo> Get-Module | Remove-Module -WarningAction SilentlyContinue
PS C:\Foo> $Functions = Get-ChildItem -Path Function:
PS C:\Foo> "Functions available [$($Functions.Count)]"
Functions available [35]
```

Figure 9.25: Getting and counting the functions available

In *step 12*, you create a simple function, which generates no output. In *step 13*, you re-check the `Function:` drive to discover your new function. The output of this step looks like this:

```
PS C:\Foo> # 13. Checking Function drive
PS C:\Foo> $Functions2 = Get-ChildItem -Path Function:
PS C:\Foo> "Functions now available [$($Functions2.Count)]"
Functions now available [36]
```

Figure 9.26: Getting and counting functions available

In *step 14*, you view the function definition for the `Get-HelloWorld` function contained in the `Function:` drive. The output looks like this:

```
PS C:\Foo> # 14. Viewing function definition
PS C:\Foo> Get-Item Function:\Get-HelloWorld | Format-List *

PSPath             : Microsoft.PowerShell.Core\Function::Get-HelloWorld
PSDrive            : Function
PSProvider         : Microsoft.PowerShell.Core\Function
PSIsContainer      : False
HelpUri            :
ScriptBlock        : 'Hello World'
CmdletBinding      : False
DefaultParameterSet :
Definition         : 'Hello World'
Options            : None
Description        :
Verb               : Get
Noun               : HelloWorld
HelpFile           :
OutputType         : {}
Name               : Get-HelloWorld
CommandType        : Function
Source             :
Version            :
Visibility         : Public
ModuleName         :
Module             :
RemotingCapability : PowerShell
Parameters         : {}
ParameterSets      : {}
```

Figure 9.27: Getting the function definition from the Function: drive

In *step 15*, you get and count the variables available in the `Variable:` drive. The output looks like this:

```
PS C:\Foo> # 15. Counting defined variables
PS C:\Foo> $Variables = Get-ChildItem -Path Variable:
PS C:\Foo> "Variables defined [$($Variables.count)]"
Variables defined [68]
```

Figure 9.28: Getting and counting variables in the Variable: drive

In *step 11* and *step 13*, you create two variables (`$Functions` and `$Functions2`). In *step 16*, specifying a wild card path, you search for variables in the `Variable:` drive that begin with "Function" with output like this:

```
PS C:\Foo> # 16. Checking on available functions
PS C:\Foo> Get-Item Variable:Function*

Name        Value
----        -----
Functions2  {A:, B:, C:, cd.., cd\, D:, E:, F:, G:, Get-HelloWorld…
Functions   {A:, B:, C:, cd.., cd\, D:, E:, F:, G:, Get-HelpDetailed…
```

Figure 9.29: Getting variables, starting with "Function"

In *step 17*, you get the certificate from the current user's trusted publisher store, with output like this:

```
PS C:\Foo> # 17. Getting trusted root certificates for the local machine
PS C:\Foo> Get-ChildItem -Path Cert:\LocalMachine\Root |
             Format-Table FriendlyName, Thumbprint

FriendlyName                                                Thumbprint
------------                                                ----------
Microsoft Root Certificate Authority                        CDD4EEAE6000AC7F40C3802C171E30148030C072
Thawte Timestamping CA                                      BE36A4562FB2EE05DBB3D32323ADF445084ED656
Microsoft Root Authority                                    A43489159A520F0D93D032CCAF37E7FE20A8B419
                                                            92B46C76E13054E104F230517E6E504D43AB10B5
Microsoft Root Certificate Authority 2011                   8F43288AD272F3103B6FB1428485EA3014C0BCFE
Microsoft Authenticode(tm) Root                             7F88CD7223F3C813818C994614A89C99FA3B5247
Microsoft Root Certificate Authority 2010                   3B1EFD3A66EA28B16697394703A72CA340A05BD5
Microsoft ECC TS Root Certificate Authority 2018            31F9FC8BA3805986B721EA7295C65B3A44534274
Microsoft Timestamp Root                                    245C97DF7514E7CF2DF8BE72AE957B9E04741E85
VeriSign Time Stamping CA                                   18F7C1FCC3090203FD5BAA2F861A754976C8DD25
Microsoft ECC Product Root Certificate Authority 2018       06F1AA330B927B753A40E68CDF22E34BCBEF3352
Microsoft Time Stamp Root Certificate Authority 2014        0119E81BE9A14CD8E22F40AC118C687ECBA3F4D8
DigiCert Global Root G2                                     DF3C24F9BFD666761B268073FE06D1CC8D4F82A4
DST Root CA X3                                              DAC9024F54D8F6DF94935FB1732638CA6AD77C13
GlobalSign Root CA - R3                                     D69B561148F01C77C54578C10926DF5B856976AD
DigiCert Baltimore Root                                     D4DE20D05E66FC53FE1A50882C78DB2852CAE474
Sectigo (AAA)                                               D1EB23A46D17D68FD92564C2F1F1601764D8E349
Starfield Root Certificate Authority - G2                   B51C067CEE2B0C3DF855AB2D92F4FE39D4E70F0E
Starfield Class 2 Certification Authority                   AD7E1C28B064EF8F6003402014C3D0E3370EB58A
DigiCert                                                    A8985D3A65E5E5C4B2D7D66D40C6DD2FB19C5436
Trustwave                                                   8782C6C304353BCFD29692D2593E7D44D934FF11
Google Trust Services - GlobalSign Root CA-R2               75E0ABB6138512271C04F85FDDDE38E4B7242EFE
VeriSign Class 3 Public Primary CA                          742C3192E607E424EB4549542BE1BBC53E6174E2
DigiCert                                                    5FB7EE0633E259DBAD0C4C9AE6D38F1A61C7DC25
VeriSign                                                    4EB6D578499B1CCF5F581EAD56BE3D9B6744A5E5
VeriSign Universal Root Certification Authority             3679CA35668772304D30A5FB873B0FA77BB70D54
Go Daddy Class 2 Certification Authority                    2796BAE63F1801E277261BA0D77770028F20EEE4
DigiCert                                                    0563B8630D62D75ABBC8AB1E4BDFB5A899B24D43
Microsoft Flighting Root 2014                               F8DB7E1C16F1FFD4AAAD4AAD8DFF0F2445184AEB
Microsoft ECC Development Root Certificate Authority 2018   6CA22E5501CC80885FF281DD8B3338E89398EE18
```

Figure 9.30: Getting certificates from the current user's trusted publisher certificate store

The WinRM service implements PowerShell remoting. You configure the client and server sides of WinRM by updating items in the `WSMan:` drive. You can view the ports used by the WSMan client and WSMan server services on the `SRV1` host, as shown in *step 18*. The output looks like this:

```
PS C:\Foo> # 18. Examining ports in use by WinRM
PS C:\Foo> Get-ChildItem -Path WSMan:\localhost\Client\DefaultPorts

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

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

PS C:\Foo> Get-ChildItem -Path WSMan:\localhost\Service\DefaultPorts

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

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

Figure 9.31: Getting WSMan service ports

In *step 19*, you set WinRM to trust any host by setting the `TrustedHosts` item in the WSMan drive. There is no output from this step.

In *step 20,* you install the SHiPS and CimPSDrive modules, creating no output. Then, in *step 21*, you import the `CimPSDrive` module and create a drive, which looks like this:

```
PS C:\Foo> # 21. Importing the CimPSDrive module and creating a drive
PS C:\Foo> Import-Module -Name CimPSDrive
PS C:\Foo> New-PSDrive -Name CIM -PSProvider SHiPS -Root CIMPSDrive#CMRoot

Name  Used (GB)   Free (GB) Provider   Root              CurrentLocation
----  ---------   --------- --------   ----              ---------------
CIM                         SHiPS      CIMPSDrive#CMRoot
```

Figure 9.32: Importing the CimPSDrive module and creating a drive

In *step 22*, you use the `CIM:` drive to view the system BIOS details from WMI. The output looks like this:

```
PS C:\Foo> # 22. Examining BIOS
PS C:\Foo> Get-ChildItem CIM:\Localhost\CIMV2\Win32_Bios

SMBIOSBIOSVersion : 090008
Manufacturer      : American Megatrends Inc.
Name              : BIOS Date: 12/07/18 15:46:29  Ver: 09.00.08
SerialNumber      : 4148-1377-3576-8329-5489-6416-80
Version           : VRTUAL - 12001807
PSComputerName    : localhost
```

Figure 9.33: Using the CIM: drive

There's more...

In *step 4*, you use the registry provider to return the registered owner of this system. This value was set when you installed Windows. If you use the Resource Kit build scripts on GitHub, the unattended XML file provides a username and organization. Of course, you can change this in the XML or subsequently.

In *step 11*, you check PowerShell's `Function:` drive. This drive holds an entry for every function within a PowerShell session. Since PowerShell has no `Remove-Function` command, to remove a function from your PowerShell session, you remove its entry (`Remove-Item -Path Function:<function to remove>`).

With *step 17*, you view the trusted root CA certificates in the local machine's certificate store. These root certificates are maintained by Microsoft, as part of the Microsoft Root Certificate Program, which supports the distribution of root certificates, enabling customers to trust Windows products. You can read more about this program at `https://docs.microsoft.com/security/trusted-root/program-requirements`.

In *step 19*, you set the WinRM TrustedHosts item to "*". This means that whenever PowerShell negotiates a remoting connection with a remote host, it trusts the remote host machine is who it says it is and is not an imposter. This has security implications – you should be careful about when and where you modify this setting.

The SHiPS framework, which you install in *step 20*, is a module that helps you to develop a provider. The framework is available, as you see in this recipe, from the PowerShell Gallery or from GitHub at `https://github.com/PowerShell/SHiPS`. For a more in-depth explanation of the SHiPS framework, see `https://4sysops.com/archives/create-a-custom-powershell-provider/`. This framework can be useful in enabling you to create new providers to unlock data in your organization.

Managing Storage Replica

Storage Replica (**SR**) is a feature of Windows Server 2022 that replicates storage volumes to other systems. SR is only available with the Windows Server 2022 Datacenter edition. With SR, you replicate all the files in one volume, for example, the `F:` drive, to a disk on another host, for example, `SRV2`. After setting up the SR partnership, as you update the `F:` drive, Windows automatically updates the target drive on `SRV2`, although you cannot see the files while Windows is replicating the volume (from `SRV1` to `SRV2`). An SR partnership also requires a drive on the source and destination hosts for internal logging. SR is useful for maintaining a complete replica of one or more disks, typically for disaster recovery.

Getting ready

You run this recipe on `SRV1`, a domain-joined host in the `Reskit.Org` domain, after adding and configuring additional virtual disks to this host. You must have installed PowerShell 7 and VS Code on this host. This recipe explicitly uses the `F:` drive you created earlier on `SRV1` in *Managing physical disks and volumes*, plus a new `G:` drive, which you create in disk number 2 (and the corresponding disks on `SRV2`).

How to do it...

1. Getting disk number of the disk holding the F partition

```
$Part = Get-Partition -DriveLetter F
"F drive on disk [$($Part.DiskNumber)]"
```

2. Creating `F:` volume on `SRV2`

```
$SB = {
  $NVHT = @{
   DiskNumber   =  $using:Part.DiskNumber
    FriendlyName = 'Files'
    FileSystem   = 'NTFS'
    DriveLetter  = 'F'
  }
  New-Volume @NVHT
}
Invoke-Command -ComputerName SRV2 -ScriptBlock $SB
```

3. Creating content on `F:` on `SRV1`

```
1..100 | ForEach-Object {
  $NF = "F:\CoolFolder$_"
  New-Item -Path $NF -ItemType Directory | Out-Null
  1..100 | ForEach {
    $NF2 = "$NF\CoolFile$_"
```

```
    "Cool File" | Out-File -PSPath $NF2
  }
}
```

4. Showing what is on `F:` locally

```
Get-ChildItem -Path F:\ -Recurse | Measure-Object
```

5. Examining the same drives remotely on SRV2

```
$SB2 = {
  Get-ChildItem -Path F:\ -Recurse |
    Measure-Object
}
Invoke-Command -ComputerName SRV2 -ScriptBlock $SB2
```

6. Adding the Storage Replica feature to SRV1

```
Add-WindowsFeature -Name Storage-Replica | Out-Null
```

7. Adding the Storage Replica feature to SRV2

```
$SB= {
  Add-WindowsFeature -Name Storage-Replica | Out-Null
}
Invoke-Command -ComputerName SRV2 -ScriptBlock $SB
```

8. Restarting SRV2 and waiting for the restart

```
$RSHT = @{
  ComputerName = 'SRV2'
  Force        = $true
}
Restart-Computer @RSHT -Wait -For PowerShell
```

9. Restarting SRV1 to finish the installation process

```
Restart-Computer
```

10. Creating a `G:` volume in disk 2 on SRV1

```
$SB4 = {
  $NVHT = @{
   DiskNumber   =  2
   FriendlyName = 'SRLOGS'
   DriveLetter  = 'G'
  }
  Clear-Disk -Number 2 -RemoveData -Confirm:$False
  Initialize-Disk -Number 2 | Out-Null
  New-Volume @NVHT
}
Invoke-Command -ComputerName SRV1 -ScriptBlock $SB4
```

11. Creating a G: volume on SRV2

```
Invoke-Command -ComputerName SRV2 -ScriptBlock $SB4
```

12. Viewing volumes on SRV1

```
Get-Volume | Sort-Object -Property Driveletter
```

13. Viewing volumes on SRV2

```
Invoke-Command -Computer SRV2 -Scriptblock {
    Get-Volume | Sort-Object -Property Driveletter
}
```

14. Creating an SR replica group

```
$SRHT =  @{
  SourceComputerName       = 'SRV1'
  SourceRGName             = 'SRV1RG'
  SourceVolumeName         = 'F:'
  SourceLogVolumeName      = 'G:'
  DestinationComputerName  = 'SRV2'
  DestinationRGName        = 'SRV2RG'
  DestinationVolumeName    = 'F:'
  DestinationLogVolumeName = 'G:'
  LogSizeInBytes           = 2gb
}
New-SRPartnership @SRHT
```

15. Examining the volumes on SRV2

```
$SB5 = {
  Get-Volume |
    Sort-Object -Property DriveLetter |
      Format-Table
}
Invoke-Command -ComputerName SRV2 -ScriptBlock $SB5
```

16. Reversing the replication

```
$SRHT2 = @{
  NewSourceComputerName   = 'SRV2'
  SourceRGName            = 'SRV2RG'
  DestinationComputerName = 'SRV1'
  DestinationRGName       = 'SRV1RG'
  Confirm                 = $false
}
Set-SRPartnership @SRHT2
```

17. Viewing the SR partnership

```
Get-SRPartnership
```

18. Examining the files remotely on SRV2

```
$SB6 = {
  Get-ChildItem -Path F:\ -Recurse |
    Measure-Object
}
Invoke-Command -ComputerName SRV2 -ScriptBlock $SB6
```

How it works...

In *step 1*, you get and display the disk number of the disk holding the `F:` volume on SRV1. The output looks like this:

```
PS C:\Foo> # 1. Getting disk number of the disk holding the F partition
PS C:\Foo> $Part = Get-Partition -DriveLetter F
PS C:\Foo> "F drive on disk [$($Part.DiskNumber)]"
F drive on disk [1]
```

Figure 9.34: Getting the F: drive on SRV1

In *step 2*, you create an `F:` volume on SRV2, which looks like this:

```
PS C:\Foo> # 2. Creating F: drive on SRV2
PS C:\Foo> $SB = {
PS C:\Foo> $NVHT = @{
               DiskNumber   =  $using:Part.DiskNumber
               FriendlyName = 'Files'
               FileSystem   = 'NTFS'
               DriveLetter  = 'F'
             }
             New-Volume @NVHT
           }
PS C:\Foo> Invoke-Command -ComputerName SRV2 -ScriptBlock $SB

DriveLetter FriendlyName FileSystemType DriveType HealthStatus OperationalStatus SizeRemaining     Size PSComputerName
----------- ------------ -------------- --------- ------------ ----------------- -------------     ---- --------------
F           Files        NTFS           Fixed     Healthy      OK                    63.84 GB 63.98 GB SRV2
```

Figure 9.35: Creating F: on SRV2

In *step 3*, you create content in the `F:` drive on SRV1 by creating 100 folders and adding 100 files to each of those folders. This step creates no output. In *step 4*, you use `Measure-Object` to count the number of files and folders in the `F:` drive on SRV1, which looks like this:

```
PS C:\Foo> # 4. Showing what is on F: locally
PS C:\Foo> Get-ChildItem -Path F:\ -Recurse | Measure-Object

Count             : 10100
Average           :
Sum               :
Maximum           :
Minimum           :
StandardDeviation :
Property          :
```

Figure 9.36: Measuring the files and folders on F: on SRV1

In *step 5*, you examine the `F:` drive on `SRV2`, which looks like this:

```
PS C:\Foo> # 5. Examining the same drives remotely on SRV2
PS C:\Foo> $SB2 = {
             Get-ChildItem -Path F:\ -Recurse |
               Measure-Object
           }
PS C:\Foo> Invoke-Command -ComputerName SRV2 -ScriptBlock $SB2

Count          : 0
Average        :
Sum            :
Maximum        :
Minimum        :
Property       :
PSComputerName : SRV2
```

Figure 9.37: Measuring the files and folders on F: on SRV2

In *step 6*, you add the Storage Replica feature to `SRV1`, producing no output to the console. In *step 7*, you add the Storage Replica feature to `SRV2`, with output that looks like this:

```
PS C:\Foo> # 7. Adding the SR feature to SRV2
PS C:\Foo> $SB= {
             Add-WindowsFeature -Name Storage-Replica | Out-Null
           }
PS C:\Foo> Invoke-Command -ComputerName SRV2 -ScriptBlock $SB
WARNING: You must restart this server to finish the installation process.
```

Figure 9.38: Adding Storage Replica to SRV2

You have to reboot `SRV2` to complete Storage Replica's installation, which you do in *step 8*. In *step 9*, you reboot `SRV1`, which completes the process of installing SR on both hosts.

After you reboot SRV2 (and SRV1), in *step 10*, you create a new volume on SRV1. This volume is to hold SR internal log files. The output looks like this:

```
PS C:\Foo> # 10. Creating a G: volume in disk 2 on SRV1
PS C:\Foo> $SB4 = {
             $NVHT = @{
               DiskNumber   =  2
               FriendlyName = 'SRLOGS'
               DriveLetter  = 'G'
             }
             Clear-Disk -Number 2 -RemoveData -Confirm:$False
             Initialize-Disk -Number 2 | Out-Null
             New-Volume @NVHT
           }
PS C:\Foo> Invoke-Command -ComputerName SRV1 -ScriptBlock $SB4

DriveLetter FriendlyName FileSystemType DriveType HealthStatus OperationalStatus SizeRemaining     Size PSComputerName
----------- ------------ -------------- --------- ------------ ----------------- -------------     ---- --------------
G           SRLOGS       NTFS           Fixed     Healthy      OK                    63.89 GB 63.98 GB SRV1
```

Figure 9.39: Adding a G: drive to SRV1

In *step 11*, you create a `G:` volume on SRV2, with output like this:

```
PS C:\Foo> # 11. Creating G: volume on SRV2
PS C:\Foo> Invoke-Command -ComputerName SRV2 -ScriptBlock $SB4

DriveLetter FriendlyName FileSystemType DriveType HealthStatus OperationalStatus SizeRemaining     Size PSComputerName
----------- ------------ -------------- --------- ------------ ----------------- -------------     ---- --------------
G           SRLOGS       NTFS           Fixed     Healthy      OK                    63.84 GB 63.98 GB SRV2
```

Figure 9.40: Adding a G: drive to SRV2

In *step 12*, you view the volumes on SRV1 with output like this:

```
PS C:\Foo> # 12. Viewing volumes on SRV1
PS C:\Foo> Get-Volume | Sort-Object -Property Driveletter

DriveLetter FriendlyName         FileSystemType DriveType HealthStatus OperationalStatus SizeRemaining      Size
----------- ------------         -------------- --------- ------------ ----------------- -------------      ----
                                 NTFS           Fixed     Healthy      OK                    80.59 MB    496 MB
A                                Unknown        Removable Healthy      Unknown                   0 B      0 B
C                                NTFS           Fixed     Healthy      OK                  110.34 GB 127.51 GB
D           SSS_X64FRE_EN-US_DV9 Unknown        CD-ROM    Healthy      OK                        0 B   4.38 GB
F           Files                NTFS           Fixed     Healthy      OK                   63.76 GB  63.98 GB
G           SRLOGS               NTFS           Fixed     Healthy      OK                   63.89 GB  63.98 GB
W                                Unknown        Fixed     Healthy      Unknown                   0 B      0 B
X           x-exFAT              exFAT          Fixed     Healthy      OK                      15 GB    15 GB
Y           Y-FAT32              FAT32          Fixed     Healthy      OK                   14.98 GB  14.98 GB
Z           Z-ReFS               ReFS           Fixed     Healthy      OK                   17.87 GB  18.94 GB
```

Figure 9.41: Viewing volumes on SRV1

In *step 13*, you view the volumes on SRV2 with output like this:

```
PS C:\Foo> # 13. Viewing volumes on SRV2
PS C:\Foo> Invoke-Command -Computer SRV2 -Scriptblock {
             Get-Volume | Sort-Object -Property Driveletter
           }

DriveLetter FriendlyName         FileSystemType DriveType HealthStatus OperationalStatus SizeRemaining      Size PSComputerName
----------- ------------         -------------- --------- ------------ ----------------- -------------      ---- --------------
                                 NTFS           Fixed     Healthy      OK                     79.58 MB    495 MB SRV2
A                                Unknown        Removable Healthy      Unknown                    0 B      0 B SRV2
C                                NTFS           Fixed     Healthy      OK                 105.07 GB 127.51 GB SRV2
D           SSS_X64FRE_EN-US_DV9 Unknown        CD-ROM    Healthy      OK                         0 B   4.38 GB SRV2
F           Files                NTFS           Fixed     Healthy      OK                  63.84 GB  63.98 GB SRV2
G           SRLOGS               NTFS           Fixed     Healthy      OK                  63.84 GB  63.98 GB SRV2
```

Figure 9.42: Viewing volumes on SRV2

In *step 14*, you create an SR replica group to replicate all the files on the `F:` drive in `SRV1` to the `F:` drive on `SRV2`, using the `G:` drive on both hosts for internal SR logging. The output of this step is like this:

```
PS C:\Foo> # 14. Creating an SR replica group
PS C:\Foo> $SRHT =  @{
             SourceComputerName       = 'SRV1'
             SourceRGName             = 'SRV1RG'
             SourceVolumeName         = 'F:'
             SourceLogVolumeName      = 'G:'
             DestinationComputerName  = 'SRV2'
             DestinationRGName        = 'SRV2RG'
             DestinationVolumeName    = 'F:'
             DestinationLogVolumeName = 'G:'
             LogSizeInBytes           = 2gb
           }
PS C:\Foo> New-SRPartnership @SRHT

RunspaceId              : 40ed48ce-142e-48bf-b2c4-f4ad19c4835f
DestinationComputerName : SRV2
DestinationRGName       : SRV2RG
Id                      : 750c9c67-4bc6-485a-bd8f-33c893dc8c2d
SourceComputerName      : SRV1
SourceRGName            : SRV1RG
```

Figure 9.43: Creating a replica group

In *step 15*, you re-examine the volumes on `SRV2`, which looks like this:

```
PS C:\Foo> # 15. Examining the volumes on SRV2
PS C:\Foo> $SB5 = {
             Get-Volume |
               Sort-Object -Property DriveLetter |
                 Format-Table
           }
PS C:\Foo> Invoke-Command -ComputerName SRV2 -ScriptBlock $SB5

DriveLetter FriendlyName         FileSystemType DriveType HealthStatus OperationalStatus SizeRemaining      Size
----------- ------------         -------------- --------- ------------ ----------------- -------------      ----
                                 NTFS           Fixed     Healthy      OK                    79.58 MB    495 MB
A                                Unknown        Removable Healthy      Unknown                0 B        0 B
C                                NTFS           Fixed     Healthy      OK                   105.07 GB 127.51 GB
D           SSS_X64FRE_EN-US_DV9 Unknown        CD-ROM    Healthy      OK                     0 B      4.38 GB
F                                Unknown        Fixed     Healthy      Unknown                0 B        0 B
G           SRLOGS               NTFS           Fixed     Healthy      OK                    61.84 GB  63.98 GB
```

Figure 9.44: Viewing volumes on SRV2

With *step 16*, you reverse the replication – the files on `SRV2` now get replicated to `SRV1`. This step creates no output. In *step 17*, you re-view the SR partnership with output like this:

```
PS C:\Foo> # 17. Viewing the SR partnership
PS C:\Foo> Get-SRPartnership

RunspaceId              : 40ed48ce-142e-48bf-b2c4-f4ad19c4835f
DestinationComputerName : SRV1
DestinationRGName       : SRV1RG
Id                      : 750c9c67-4bc6-485a-bd8f-33c893dc8c2d
SourceComputerName      : SRV2
SourceRGName            : SRV2RG
```

Figure 9.45: Viewing the SR partnership

In *step 18*, you view the files on the `F:` drive on `SRV2`, which looks like this:

```
PS C:\Foo> # 18. Examining the files remotely on SRV2
PS C:\Foo> $SB6 = {
             Get-ChildItem -Path F:\ -Recurse |
               Measure-Object
           }
PS C:\Foo> Invoke-Command -ComputerName SRV2 -ScriptBlock $SB6

Count          : 10100
Average        :
Sum            :
Maximum        :
Minimum        :
Property       :
PSComputerName : SRV2
```

Figure 9.46: Viewing files on F: on SRV2

There's more...

In the first five steps in this recipe, you create content on SRV1, which you intend to have Storage Replica replicate to SRV2. In practice, the data you are replicating would be the files in a file server or other files you want to synchronize.

In *step 9*, you reboot SRV2 remotely. If this is the first time you have rebooted SRV2 remotely using PowerShell, you may find that the command never returns even when SRV2 is demonstrably up and running. Just kill off the current PowerShell console (or close VS Code) and open a new console to continue the recipe.

After creating the SR partnership, replication from SRV1 to SRV2, you reverse the replication in *step 16*. Before you reverse the replication, you should ensure that the initial replication has been completed. Depending on the size of the volume, it could take quite a while. You can use the `(Get-SRGroup).Replicas.ReplicationStatus` command to check the status of the initial replication.

Deploying Storage Spaces

Storage Spaces is a technology in Windows 10 and Windows Server that implements software RAID. You can add multiple physical drives into your server or workstation, and create fault-tolerant volumes for your host. You can read more about Storage Spaces at `https://docs.microsoft.com/windows-server/storage/storage-spaces/overview`.

You can use Storage Spaces on a single host or server to protect against unexpected disk drive failures. You should note that Storage Spaces is separate from **Storage Spaces Direct** (aka **S2D**). S2D enables you to create a virtual SAN with multiple hosts providing SMB3 access to a scale-out file server.

Getting ready

You run this recipe on SRV1, a domain-joined host in the Reskit.Org domain. You also need DC1, a domain controller for the Reskit.org domain. This recipe makes use of five of the virtual disks you added to SRV1 at the start of the *Managing physical disks and volumes* recipe.

How to do it...

1. Viewing disks available for pooling

```
$Disks = Get-PhysicalDisk -CanPool $true
$Disks
```

2. Creating a storage pool

```
$SPHT = @{
    FriendlyName                 = 'RKSP'
    StorageSubsystemFriendlyName = "Windows Storage*"
    PhysicalDisks                = $Disks
}
New-StoragePool @SPHT
```

3. Creating a mirrored hard disk named `Mirror1`

```
$VDHT1 = @{
  StoragePoolFriendlyName   = 'RKSP'
  FriendlyName              = 'Mirror1'
  ResiliencySettingName     = 'Mirror'
  Size                      = 8GB
  ProvisioningType          = 'Thin'
}
New-VirtualDisk @VDHT1
```

4. Creating a three-way mirrored disk named `Mirror2`

```
$VDHT2 = @{
  StoragePoolFriendlyName    = 'RKSP'
  FriendlyName               = 'Mirror2'
  ResiliencySettingName      = 'Mirror'
  NumberOfDataCopies         = 3
  Size                       = 8GB
  ProvisioningType           = 'Thin'
}
New-VirtualDisk @VDHT2
```

5. Creating a volume in `Mirror1`

```
Get-VirtualDisk  -FriendlyName 'Mirror1' |
  Get-Disk |
    Initialize-Disk -PassThru |
      New-Partition -AssignDriveLetter -UseMaximumSize |
        Format-Volume
```

6. Creating a volume in `Mirror2`

```
Get-VirtualDisk  -FriendlyName 'Mirror2' |
  Get-Disk |
    Initialize-Disk -PassThru |
      New-Partition -AssignDriveLetter -UseMaximumSize |
        Format-Volume
```

7. Viewing volumes on `SRV2`

```
Get-Volume | Sort-Object -Property DriveLetter
```

How it works...

In *step 1*, you examine the disks available in `SRV1` at the start of this recipe:

```
PS C:\Foo> # 1. Viewing disks available for pooling
PS C:\Foo> $Disks = Get-PhysicalDisk -CanPool $true
PS C:\Foo> $Disks | Sort-Object -Property Number

Number FriendlyName      SerialNumber MediaType   CanPool OperationalStatus HealthStatus Usage        Size
------ ------------      ------------ ---------   ------- ----------------- ------------ -----        ----
4      Msft Virtual Disk              Unspecified True    OK                Healthy      Auto-Select 64 GB
5      Msft Virtual Disk              Unspecified True    OK                Healthy      Auto-Select 64 GB
6      Msft Virtual Disk              Unspecified True    OK                Healthy      Auto-Select 64 GB
7      Msft Virtual Disk              Unspecified True    OK                Healthy      Auto-Select 64 GB
8      Msft Virtual Disk              Unspecified True    OK                Healthy      Auto-Select 64 GB
```

Figure 9.47: Viewing disks available for pooling on SRV1

In *step 2*, you use the `New-StoragePool` cmdlet to create a new storage pool using the five disks you discovered in the previous step. The output looks like this:

```
PS C:\Foo> # 2. Creating a storage pool
PS C:\Foo> $SPHT = @{
             FriendlyName                = 'RKSP'
             StorageSubsystemFriendlyName = "Windows Storage*"
             PhysicalDisks               = $Disks
           }
PS C:\Foo> New-StoragePool @SPHT

FriendlyName OperationalStatus HealthStatus IsPrimordial IsReadOnly      Size AllocatedSize
------------ ----------------- ------------ ------------ ----------      ---- -------------
RKSP         OK                Healthy      False        False      317.42 GB       1.25 GB
```

Figure 9.48: Creating a new storage pool

In *step 3*, you create a new Storage Space called `Mirror1`. This Storage Space is effectively a virtual disk within a storage pool. The output of this step looks like this:

```
PS C:\Foo> # 3. Creating a mirrored hard disk named Mirror1
PS C:\Foo> $VDHT1 = @{
             StoragePoolFriendlyName   = 'RKSP'
             FriendlyName              = 'Mirror1'
             ResiliencySettingName     = 'Mirror'
             Size                      = 8GB
             ProvisioningType          = 'Thin'
           }
PS C:\Foo> New-VirtualDisk @VDHT1

FriendlyName ResiliencySettingName FaultDomainRedundancy OperationalStatus HealthStatus Size FootprintOnPool StorageEfficiency
------------ --------------------- --------------------- ----------------- ------------ ---- -------------- -----------------
Mirror1      Mirror                1                     OK                Healthy      8 GB          1.5 GB            33.33%
```

Figure 9.49: Creating a mirrored Storage Space

In *step 4*, you create a three-way mirrored disk called `Mirror2`. The output of this step looks like this:

```
PS C:\Foo> # 4. Creating a three way mirrored disk named Mirror2
PS C:\Foo> $VDHT2 = @{
             StoragePoolFriendlyName    = 'RKSP'
             FriendlyName               = 'Mirror2'
             ResiliencySettingName      = 'Mirror'
             NumberOfDataCopies         = 3
             Size                       = 8GB
             ProvisioningType           = 'Thin'
           }
PS C:\Foo> New-VirtualDisk @VDHT2

FriendlyName ResiliencySettingName FaultDomainRedundancy OperationalStatus HealthStatus Size FootprintOnPool StorageEfficiency
------------ --------------------- --------------------- ----------------- ------------ ---- -------------- -----------------
Mirror2      Mirror                2                     OK                Healthy      8 GB          1.5 GB            16.67%
```

Figure 9.50: Creating a three-way mirrored Storage Space

In *step 5*, you create a new volume in the `Mirror1` Storage Space, which looks like this:

```
PS C:\Foo> # 5. Creating volume in Mirror1
PS C:\Foo> Get-VirtualDisk  -FriendlyName 'Mirror1' |
             Get-Disk |
               Initialize-Disk -PassThru |
                 New-Partition -AssignDriveLetter -UseMaximumSize |
                   Format-Volume

DriveLetter FriendlyName FileSystemType DriveType HealthStatus OperationalStatus SizeRemaining    Size
----------- ------------ -------------- --------- ------------ ----------------- -------------    ----
E                        NTFS           Fixed     Healthy      OK                     7.95 GB 7.98 GB
```

Figure 9.51: Creating a disk inside the Mirror1 Storage Space

In *step 6*, you create a new volume in the three-way mirror Storage Space, `Mirror2`, with output like this:

```
PS C:\Foo> # 6. Creating a volume in Mirror2
PS C:\Foo> Get-VirtualDisk  -FriendlyName 'Mirror2' |
             Get-Disk |
               Initialize-Disk -PassThru |
                 New-Partition -AssignDriveLetter -UseMaximumSize |
                   Format-Volume

DriveLetter FriendlyName FileSystemType DriveType HealthStatus OperationalStatus SizeRemaining    Size
----------- ------------ -------------- --------- ------------ ----------------- -------------    ----
H                        NTFS           Fixed     Healthy      OK                     7.95 GB 7.98 GB
```

Figure 9.52: Creating a disk inside the Mirror2 Storage Space

In the final step in this recipe, *step 7*, you use the `Get-Volume` command to view all the volumes available in `SRV2`, with output like this:

```
PS C:\Foo> # 7.  Viewing volumes on SRV2
PS C:\Foo> Get-Volume | Sort-Object -Property DriveLetter

DriveLetter FriendlyName         FileSystemType DriveType HealthStatus OperationalStatus SizeRemaining      Size
----------- ------------         -------------- --------- ------------ ----------------- -------------      ----
                                 NTFS           Fixed     Healthy      OK                     80.59 MB    496 MB
A                                Unknown        Removable Healthy      Unknown                    0 B      0 B
C                                NTFS           Fixed     Healthy      OK                   109.91 GB 127.51 GB
D           SSS_X64FRE_EN-US_DV9 Unknown        CD-ROM    Healthy      OK                         0 B   4.38 GB
E                                NTFS           Fixed     Healthy      OK                     7.95 GB   7.98 GB
F                                Unknown        Fixed     Healthy      Unknown                    0 B      0 B
G           SRLOGS               NTFS           Fixed     Healthy      OK                    61.89 GB  63.98 GB
H                                NTFS           Fixed     Healthy      OK                     7.95 GB   7.98 GB
W                                Unknown        Fixed     Healthy      Unknown                    0 B      0 B
X           x-exFAT              exFAT          Fixed     Healthy      OK                       15 GB    15 GB
Y           Y-FAT32              FAT32          Fixed     Healthy      OK                     14.98 GB  14.98 GB
Z           Z-ReFS               ReFS           Fixed     Healthy      OK                     17.87 GB  18.94 GB
```

Figure 9.53: Viewing disks available on SRV2

There's more...

In *step 5* and *step 6*, you create two new drives in `SRV1`. The first is the `E:` drive, which you created in the mirror set, `Mirror1`, and the second is the `H:` drive, a disk created in the three-way mirror Storage Space. The drive you create in `Mirror1` is resilient to the loss of a single disk, whereas the `H:` drive created in `Mirror2` can sustain two disk failures and still provide full access to your information.

10

Managing Shared Data

In this chapter, we cover the following recipes:

- Managing NTFS file and folder permissions
- Setting up and securing an SMB file server
- Creating and securing SMB shares
- Accessing SMB shares
- Creating an iSCSI target
- Using an iSCSI target
- Implementing FSRM quotas
- Implementing FSRM reporting
- Implementing FSRM file screening

Introduction

Sharing data with other users on your network has been a feature of computer operating systems from the very earliest days of networking. This chapter looks at Windows Server 2022 features that enable you to share files and folders and use the data that you've shared.

Microsoft's LAN Manager was the company's first network offering. It enabled client computers to create, manage, and share files securely. LAN Manager's protocol to provide this client/server functionality was an early version of the **Server Message Block** (**SMB**) protocol.

SMB is a file-level storage protocol running over TCP/IP. With the SMB protocol, you can share files and folders securely and reliably. To increase reliability for SMB servers, you can install a cluster and cluster the file server role. A simple cluster solution is an active-passive solution – you have one cluster member sitting by if the other member fails. This solution works great as long as the underlying data is accessible. The **Scale-Out File Server** (**SOFS**) is a clustering-based solution that is active-active. Both nodes of the cluster can serve cluster clients.

This chapter shows you how to implement and leverage the features of sharing data between systems in Windows Server 2022.

In the first recipe, *Managing NTFS file and folder permissions*, you use the NTFS Security third-party module to set ACLs and ACL inheritance for files held in NTFS from `SRV1`. In the following recipe, *Setting up and securing an SMB file server*, you deploy a hardened SMB file server. You run that recipe on `SRV2`.

iSCSI is a popular **Storage Area Networking** (**SAN**) technology. Many SAN vendors provide iSCSI as a way to access data stored in a SAN. There are two aspects to iSCSI: the server (the iSCSI target) and the client (the iSCSI initiator). In the *Creating an iSCSI target* recipe, you create an iSCSI target on the SS1 server, while in the *Using an iSCSI target* recipe, you make use of that shared iSCSI disk from `SRV1` and `SRV2`.

File System Resource Manager (**FSRM**) is a Windows Server feature designed to help you manage file servers. You can use FSRM to set user quotas for folders, set file screens, and create rich reports.

Several servers are involved in the recipes in this chapter – each recipe describes the specific server(s) you use for that recipe. As with other chapters, all the servers are members of the `Reskit.Org` domain, on which you have loaded PowerShell 7 and VS Code. You can install them by using the `Reskit.Org` setup scripts on GitHub.

Managing NTFS file and folder permissions

Every file and folder in an NTFS filesystem has an **Access Control List** (**ACL**). The ACL contains a set of **Access Control Entries** (**ACEs**). Each ACE defines permission to a file or folder for an account. For example, you could give the Sales AD global group full control of a file.

NTFS also allows a file or folder to inherit permission from its parent folder. If you create a new folder and then create a file within that new folder, the new file inherits the parent folder's permissions. You can manage the ACL to add or remove permissions, and you can modify inheritance.

There's limited PowerShell support for managing NTFS permissions. PowerShell does have the `Get-ACL` and `Set-ACL` cmdlets, but creating the individual ACEs and managing inheritance requires using the .NET Framework (by default). A more straightforward approach is to use a third-party module, `NTFSSecurity`, which makes managing ACEs and ACLs, including dealing with inheritance, a lot easier.

Getting ready

This recipe uses SRV2, a domain-joined host in the Reskit.Org domain, on which you have installed PowerShell 7 and VS Code. You also need DC1 online. In *Chapter 9*, *Managing Storage*, you added several virtual disks to SRV1 and SRV2 and set up a Storage Replica. You created an F: drive on both servers. Storage Replication should be replicating from disk 1 (the F: partition) in SRV2 to disk 1 on SRV1 (effectively in a hidden partition). If your Hyper-V host is low on physical resources, consider removing the replication partnership.

How to do it...

1. Downloading NTFSSecurity module from PSGallery

```
Install-Module NTFSSecurity -Force
```

2. Getting commands in the module

```
Get-Command -Module NTFSSecurity
```

3. Creating a new folder and a file in the folder

```
New-Item -Path F:\Secure1 -ItemType Directory |
  Out-Null
"Secure" | Out-File -FilePath F:\Secure1\Secure.Txt
Get-ChildItem -Path F:\Secure1
```

4. Viewing ACL of the folder

```
Get-NTFSAccess -Path F:\Secure1 |
  Format-Table -AutoSize
```

5. Viewing ACL of the file

```
Get-NTFSAccess F:\Secure1\Secure.Txt |
  Format-Table -AutoSize
```

6. Creating the Sales group in AD if it does not exist

```
$SB = {
  try {
    Get-ADGroup -Identity 'Sales' -ErrorAction Stop
  }
  catch {
    New-ADGroup -Name Sales -GroupScope Global |
      Out-Null
  }
}
Invoke-Command -ComputerName DC1 -ScriptBlock $SB
```

7. Displaying Sales AD Group

```
Invoke-Command -ComputerName DC1 -ScriptBlock {
                                    Get-ADGroup -Identity Sales}
```

8. Adding explicit full control for Domain Admins

```
$AHT1 = @{
  Path         = 'F:\Secure1'
  Account      = 'Reskit\Domain Admins'
  AccessRights = 'FullControl'
}
Add-NTFSAccess @AHT1
```

9. Removing `builtin\users` access from `secure.txt` file

```
$AHT2 = @{
  Path         = 'F:\Secure1\Secure.Txt'
  Account      = 'Builtin\Users'
  AccessRights = 'FullControl'
}
Remove-NTFSAccess @AHT2
```

10. Removing inherited rights for the folder

```
$IRHT1 = @{
  Path                        = 'F:\Secure1'
  RemoveInheritedAccessRules = $True
}
Disable-NTFSAccessInheritance @IRHT1
```

11. Adding Sales group access to the folder

```
$AHT3 = @{
  Path         = 'F:\Secure1\'
  Account      = 'Reskit\Sales'
  AccessRights = 'FullControl'
}
Add-NTFSAccess @AHT3
```

12. Getting ACL on path

```
Get-NTFSAccess -Path F:\Secure1 |
  Format-Table -AutoSize
```

13. Getting resulting ACL on the file

```
Get-NTFSAccess -Path F:\Secure1\Secure.Txt |
  Format-Table -AutoSize
```

How it works...

In *step 1*, you download and install the `NTFSSecurity` module from the PowerShell Gallery. This step creates no console output. In *step 2*, you take a look at the commands provided by the `NTFSSecurity` module, with output like this:

```
PS C:\Foo> # 2. Getting commands in the module
PS C:\Foo> Get-Command -Module NTFSSecurity

CommandType  Name                              Version  Source
-----------  ----                              -------  ------
Cmdlet       Add-NTFSAccess                    4.2.6    NTFSSecurity
Cmdlet       Add-NTFSAudit                     4.2.6    NTFSSecurity
Cmdlet       Clear-NTFSAccess                  4.2.6    NTFSSecurity
Cmdlet       Clear-NTFSAudit                   4.2.6    NTFSSecurity
Cmdlet       Copy-Item2                        4.2.6    NTFSSecurity
Cmdlet       Disable-NTFSAccessInheritance     4.2.6    NTFSSecurity
Cmdlet       Disable-NTFSAuditInheritance      4.2.6    NTFSSecurity
Cmdlet       Disable-Privileges                4.2.6    NTFSSecurity
Cmdlet       Enable-NTFSAccessInheritance      4.2.6    NTFSSecurity
Cmdlet       Enable-NTFSAuditInheritance       4.2.6    NTFSSecurity
Cmdlet       Enable-Privileges                 4.2.6    NTFSSecurity
Cmdlet       Get-ChildItem2                    4.2.6    NTFSSecurity
Cmdlet       Get-DiskSpace                     4.2.6    NTFSSecurity
Cmdlet       Get-FileHash2                     4.2.6    NTFSSecurity
Cmdlet       Get-Item2                         4.2.6    NTFSSecurity
Cmdlet       Get-NTFSAccess                    4.2.6    NTFSSecurity
Cmdlet       Get-NTFSAudit                     4.2.6    NTFSSecurity
Cmdlet       Get-NTFSEffectiveAccess           4.2.6    NTFSSecurity
Cmdlet       Get-NTFSHardLink                  4.2.6    NTFSSecurity
Cmdlet       Get-NTFSInheritance               4.2.6    NTFSSecurity
Cmdlet       Get-NTFSOrphanedAccess            4.2.6    NTFSSecurity
Cmdlet       Get-NTFSOrphanedAudit             4.2.6    NTFSSecurity
Cmdlet       Get-NTFSOwner                     4.2.6    NTFSSecurity
Cmdlet       Get-NTFSSecurityDescriptor        4.2.6    NTFSSecurity
Cmdlet       Get-NTFSSimpleAccess              4.2.6    NTFSSecurity
Cmdlet       Get-Privileges                    4.2.6    NTFSSecurity
Cmdlet       Move-Item2                        4.2.6    NTFSSecurity
Cmdlet       New-NTFSHardLink                  4.2.6    NTFSSecurity
Cmdlet       New-NTFSSymbolicLink              4.2.6    NTFSSecurity
Cmdlet       Remove-Item2                      4.2.6    NTFSSecurity
Cmdlet       Remove-NTFSAccess                 4.2.6    NTFSSecurity
Cmdlet       Remove-NTFSAudit                  4.2.6    NTFSSecurity
Cmdlet       Set-NTFSInheritance               4.2.6    NTFSSecurity
Cmdlet       Set-NTFSOwner                     4.2.6    NTFSSecurity
Cmdlet       Set-NTFSSecurityDescriptor        4.2.6    NTFSSecurity
Cmdlet       Test-Path2                        4.2.6    NTFSSecurity
```

Figure 10.1: Viewing the commands in the NTFSSecurity module

In *step 3*, you create a folder and a file within that folder, with output like this:

```
PS C:\Foo> # 3. Creating a new folder and a file in the folder
PS C:\Foo> New-Item -Path F:\Secure1 -ItemType Directory |
             Out-Null
PS C:\Foo> "Secure" | Out-File -FilePath F:\Secure1\Secure.Txt
PS C:\Foo> Get-ChildItem -Path F:\Secure1

    Directory: F:\Secure1

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a---          26/02/2021    14:11              8 Secure.Txt
```

Figure 10.2: Creating a new folder and a file in the folder

In *step 4*, you use the `Get-NTFSAccess` cmdlet to return the ACL of the folder, with output like this:

```
PS C:\Foo> # 4. Viewing ACL of the folder
PS C:\Foo> Get-NTFSAccess -Path F:\Secure1 |
             Format-Table -AutoSize

    Path: F:\Secure1 (Inheritance enabled)

Account                Access Rights              Applies to                 Type  IsInherited InheritedFrom
-------                -------------              ----------                 ----  ----------- -------------
BUILTIN\Administrators FullControl                ThisFolderOnly             Allow False
BUILTIN\Administrators FullControl                ThisFolderSubfoldersAndFiles Allow True        F:
NT AUTHORITY\SYSTEM    FullControl                ThisFolderSubfoldersAndFiles Allow True        F:
CREATOR OWNER          GenericAll                 SubfoldersAndFilesOnly     Allow True        F:
BUILTIN\Users          ReadAndExecute, Synchronize ThisFolderSubfoldersAndFiles Allow True        F:
BUILTIN\Users          CreateDirectories          ThisFolderAndSubfolders    Allow True        F:
BUILTIN\Users          CreateFiles                ThisFolderAndSubfolders    Allow True        F:
```

Figure 10.3: Viewing the ACL of the F:\Secure1 folder

In *step 5*, you view the ACL of the `F:\Secure1\Secure.Txt` file, with output like this:

```
PS C:\Foo> # 5. Viewing ACL of the file
PS C:\Foo> Get-NTFSAccess F:\Secure1\Secure.Txt |
             Format-Table -AutoSize

    Path: F:\Secure1\Secure.Txt (Inheritance enabled)


Account                Access Rights              Applies to     Type  IsInherited InheritedFrom
-------                -------------              ----------     ----  ----------- -------------
BUILTIN\Administrators FullControl                ThisFolderOnly Allow True        F:
NT AUTHORITY\SYSTEM    FullControl                ThisFolderOnly Allow True        F:
BUILTIN\Users          ReadAndExecute, Synchronize ThisFolderOnly Allow True        F:
```

Figure 10.4: Viewing the ACL of the F:\Secure1\Secure.txt file

In *step 6*, you create a global group, `Sales`, in the `Reskit.Org` domain if it does not already exist. This step creates no output. In *step 7*, you view the Sales group from the AD, with output like this:

```
PS C:\Foo> # 7. Displaying Sales AD Group
PS C:\Foo> Invoke-Command -ComputerName DC1 -ScriptBlock {
                          Get-ADGroup -Identity Sales}

PSComputerName    : DC1
RunspaceId        : 52db7eca-3884-4dc5-97c5-a710cd960103
DistinguishedName : CN=Sales,CN=Users,DC=Reskit,DC=Org
GroupCategory     : Security
GroupScope        : Global
Name              : Sales
ObjectClass       : group
ObjectGUID        : f13a190e-ae7a-4533-a231-69bb3f7aec9a
SamAccountName    : Sales
SID               : S-1-5-21-1877631356-3085136781-2799030166-1146
```

Figure 10.5: Viewing the Sales global group

In *step 8*, you add an explicit ACE for the Domain Admins group for full control of the `F:\Secure1` folder. Next, in *step 9*, you remove the `builtin\users` group from access to the `Secure.txt` file. And then, in *step 10*, you remove all inherited access from `F:\Secure1`. Finally, in *step 11*, you add explicit access to the `F:\Secure1` folder to the Sales group. These four steps produce no output.

In *step 12*, you examine the updated ACL for the `F:\Secure` folder, with output like this:

```
PS C:\Foo> # 12. Getting ACL on path
PS C:\Foo> Get-NTFSAccess -Path F:\Secure1 |
             Format-Table -AutoSize

    Path: F:\Secure1 (Inheritance disabled)

Account                 Access Rights Applies to                  Type  IsInherited InheritedFrom
-------                 ------------- ----------                  ----  ----------- -------------
BUILTIN\Administrators FullControl   ThisFolderOnly              Allow False
RESKIT\Domain Admins   FullControl   ThisFolderSubfoldersAndFiles Allow False
RESKIT\Sales           FullControl   ThisFolderSubfoldersAndFiles Allow False
```

Figure 10.6: Viewing the updated ACL for the F:\Secure folder

In *step 13*, you look at the updated ACL for the `F:\Secure\Secure1.txt` file, with output like this:

```
PS C:\Foo> # 13. Getting resulting ACL on the file
PS C:\Foo> Get-NTFSAccess -Path F:\Secure1\Secure.Txt |
             Format-Table -AutoSize

    Path: F:\Secure1\Secure.Txt (Inheritance enabled)

Account                Access Rights Applies to     Type  IsInherited InheritedFrom
-------                ------------- -----------    ----  ----------- -------------
RESKIT\Domain Admins FullControl   ThisFolderOnly Allow True        F:\Secure1
RESKIT\Sales         FullControl   ThisFolderOnly Allow True        F:\Secure1
```

Figure 10.7: Viewing the updated ACL for the F:\Secure1\Secure.Txt file

There's more...

As you can see in *step 2*, looking at *Figure 10.1*, there are several cmdlets in the `NTFSSecurity` module. You can use these cmdlets to set up the ACL on a file or folder and the **system access control list** (**SACL**), which enables you to audit file or folder access. There are also some improved cmdlets, such as `Get-ChildItem2` and `Get-Item2`, which you may find helpful.

Setting up and securing an SMB file server

The next step in creating a file server is to install the necessary features to the server, and then harden it. You use the `Add-WindowsFeature` cmdlet to add the features that are necessary for a file server. You can then use the `Set-SmbServerConfiguration` cmdlet to improve the configuration.

Since your file server may contain sensitive information, you must take reasonable steps to avoid some of the expected attack mechanisms and adopt best security practices. Security is a good thing but, as always, be careful! By locking down your SMB file server too hard, you can lock some users out of the server.

SMB 1.0 has many weaknesses and, in general, should be removed. By default, Windows Server 2022 installs with SMB 1.0 turned off. Remember that if you disable SMB 1.0, you may find that older computers (for example, those running Windows XP) lose the ability to access shared data. Before you lock down any of the server configurations, be sure to test your changes carefully.

Getting ready

This recipe uses SRV2, a domain-joined host in the Reskit.Org domain, on which you have installed PowerShell 7 and VS Code. You also use SRV2 and should have DC1 online.

How to do it...

1. Adding file server features to SRV2

```
$Features = 'FileAndStorage-Services',
            'File-Services',
            'FS-FileServer',
            'RSAT-File-Services'
Add-WindowsFeature -Name $Features
```

2. Viewing the SMB server settings

```
Get-SmbServerConfiguration
```

3. Turning off SMB1

```
$CHT = @{
  EnableSMB1Protocol = $false
  Confirm            = $false
}
Set-SmbServerConfiguration @CHT
```

4. Turning on SMB signing and encryption

```
$SHT1 = @{
    RequireSecuritySignature = $true
    EnableSecuritySignature  = $true
    EncryptData              = $true
    Confirm                  = $false
}
Set-SmbServerConfiguration @SHT1
```

5. Turning off default server and workstation shares

```
$SHT2 = @{
    AutoShareServer       = $false
    AutoShareWorkstation  = $false
    Confirm               = $false
}
Set-SmbServerConfiguration @SHT2
```

6. Turning off server announcements

```
$SHT3 = @{
    ServerHidden    = $true
    AnnounceServer = $false
    Confirm         = $false
}
Set-SmbServerConfiguration @SHT3
```

7. Restarting SMB server service with the new configuration

```
Restart-Service lanmanserver -Force
```

How it works...

In *step 1*, you add the file server features to SRV2, with output like this:

```
PS C:\Foo> # 1. Adding file server features to SRV2
PS C:\Foo> $Features = 'FileAndStorage-Services',
                       'File-Services',
                       'FS-FileServer',
                       'RSAT-File-Services'
PS C:\Foo> Add-WindowsFeature -Name $Features

Success Restart Needed Exit Code  Feature Result
------- -------------- ---------  --------------
True    No             Success    {File and iSCSI Services, File Server, File …
```

Figure 10.8: Adding file server features to SRV2

In *step 2*, you use the `Get-SmbServerConfiguration` cmdlet to return the SMB server settings for `SRV2`, which looks like this:

```
PS C:\Foo> # 2. Viewing the SMB server settings
PS C:\Foo> Get-SmbServerConfiguration

AnnounceComment                           :
AnnounceServer                            : False
AsynchronousCredits                       : 512
AuditSmb1Access                           : False
AutoDisconnectTimeout                     : 15
AutoShareServer                           : True
AutoShareWorkstation                      : True
CachedOpenLimit                           : 10
DisableSmbEncryptionOnSecureConnection    : True
DurableHandleV2TimeoutInSeconds           : 180
EnableAuthenticateUserSharing             : False
EnableDownlevelTimewarp                   : False
EnableForcedLogoff                        : True
EnableLeasing                             : True
EnableMultiChannel                        : True
EnableOplocks                             : True
EnableSecuritySignature                   : False
EnableSMB1Protocol                        : False
EnableSMB2Protocol                        : True
EnableStrictNameChecking                  : True
EncryptData                               : False
IrpStackSize                              : 15
KeepAliveTime                             : 2
MaxChannelPerSession                      : 32
MaxMpxCount                               : 50
MaxSessionPerConnection                   : 16384
MaxThreadsPerQueue                        : 20
MaxWorkItems                              : 1
NullSessionPipes                          :
NullSessionShares                         :
OplockBreakWait                           : 35
PendingClientTimeoutInSeconds             : 120
RejectUnencryptedAccess                   : True
RequireSecuritySignature                  : False
ServerHidden                              : True
Smb2CreditsMax                            : 8192
Smb2CreditsMin                            : 512
SmbServerNameHardeningLevel               : 0
TreatHostAsStableStorage                  : False
ValidateAliasNotCircular                  : True
ValidateShareScope                        : True
ValidateShareScopeNotAliased              : True
ValidateTargetName                        : True
RestrictNamedpipeAccessViaQuic            : True
EnableSMBQUIC                             : True
```

Figure 10.9: Viewing the SMB server settings on SRV2

In *step 3,* you turn off SMB 1.0 explicitly. In *step 4*, you turn on digital signing and encrypting of all SMB-related data packets. In *step 5*, you turn off the default server and workstation shares, while in *step 6*, you turn off SMB server announcements to improve security. These four steps produce no output.

In *step 7*, which also produces no output, you restart the `LanManServer` service, which is the Windows service that provides SMB file sharing.

There's more...

In *steps 3* through *6*, you update the configuration of the SMB service to be more secure. The SMB 1.0 protocol has long been considered unsafe. By default, the Windows OS setup process never turns on version 1, but it's a good idea to ensure you turn it off. Digitally signing and encrypting all SMB packets protects against someone using a network sniffer to view data packets. SMB server announcements could provide more information to a potential network hacker about the services on your network.

In *step 7*, after making changes to the SMB service configuration, you restart the `LanManWorkstation` service. You must restart this service to implement any changes to the file server configuration.

Creating and securing SMB shares

With your file server service set up, the next step in deploying a file server is to create SMB shares and secure them. For decades, administrators have used the `net.exe` command to set up shared folders and more. This command continues to work, but you may find the SMB cmdlets easier to use, particularly if you're automating large-scale SMB server deployments.

This recipe looks at creating and securing shares on a Windows Server 2022 platform using the PowerShell `SMBServer` module. You also use cmdlets from the `NTFSSecurity` module (a third-party module you download from the PS Gallery).

You run this recipe on the file server (`SRV2`) that you set up and hardened in the *Setting up and securing an SMB file server* recipe. In this recipe, you share a folder (`C:\ITShare`) on the file server. Then, you create a file in the `C:\ITShare` folder you just shared and set the ACL for the files to be the same for the share. You use the `Set-SMBPathAcl` cmdlet to do this. You then review the ACL for both the folder and the file.

This recipe uses a universal security group, Sales, which you create in the Reskit.Org domain. See the *Managing printer security* recipe in *Chapter 11*, *Managing Printers*, for the script snippet you can use to create the groups used by this recipe. In this recipe, you use the `Get-NTFSAccess` cmdlet from `NTFSSecurity`, a third-party module that you downloaded from the PowerShell Gallery. See the *Managing NTFS file and folder permissions* recipe for more details about this module and instructions on downloading it.

Getting ready

This recipe uses SRV2, a domain-joined host in the Reskit.Org domain, on which you have installed PowerShell 7 and VS Code. You should also have DC1 online. If you have not created the Sales universal group yet, see the *Managing printer security* recipe in *Chapter 11*, *Managing Printers*.

How to do it...

1. Discovering existing shares and access rights

```
Get-SmbShare -Name * |
  Get-SmbShareAccess |
    Format-Table -GroupBy Name
```

2. Sharing a new folder

```
New-Item -Path C:\ -Name ITShare -ItemType Directory |
  Out-Null
New-SmbShare -Name ITShare -Path C:\ITShare
```

3. Updating the share to have a description

```
$CHT = @{Confirm=$False}
Set-SmbShare -Name ITShare -Description 'File Share for IT' @CHT
```

4. Setting folder enumeration mode

```
$CHT = @{Confirm = $false}
Set-SMBShare -Name ITShare -FolderEnumerationMode AccessBased @CHT
```

5. Setting encryption on for ITShare share

```
Set-SmbShare -Name ITShare -EncryptData $true @CHT
```

6. Removing all access to ITShare share for the Everyone group

```
$AHT1 = @{
  Name        =  'ITShare'
  AccountName = 'Everyone'
  Confirm     =  $false
}
Revoke-SmbShareAccess @AHT1 | Out-Null
```

7. Adding Reskit\Administrators to have read permissions

```
$AHT2 = @{
    Name         = 'ITShare'
    AccessRight  = 'Read'
    AccountName  = 'Reskit\ADMINISTRATOR'
```

```
    Confirm      =  $false
}
Grant-SmbShareAccess @AHT2 | Out-Null
```

8. Adding full access to the system account

```
$AHT3 = @{
    Name          = 'ITShare'
    AccessRight   = 'Full'
    AccountName   = 'NT Authority\SYSTEM'
    Confirm       = $False
}
Grant-SmbShareAccess  @AHT3 | Out-Null
```

9. Setting Creator/Owner to full access

```
$AHT4 = @{
    Name         = 'ITShare'
    AccessRight  = 'Full'
    AccountName  = 'CREATOR OWNER'
    Confirm      = $False
}
Grant-SmbShareAccess @AHT4  | Out-Null
```

10. Granting Sales group read access

```
$AHT5 = @{
    Name        = 'ITShare'
    AccessRight = 'Read'
    AccountName = 'Sales'
    Confirm     = $false
}
Grant-SmbShareAccess @AHT5 | Out-Null
```

11. Reviewing share access

```
Get-SmbShareAccess -Name ITShare |
  Sort-Object AccessRight
```

12. Setting file ACL to be the same as the share ACL

```
Set-SmbPathAcl -ShareName 'ITShare'
```

13. Creating a file in `C:\ITShare`

```
'File Contents' | Out-File -FilePath C:\ITShare\File.Txt
```

14. Setting the file ACL to be the same as the share ACL

```
Set-SmbPathAcl -ShareName 'ITShare'
```

15. View the file ACL

```
Get-NTFSAccess -Path  C:\ITShare\File.Txt |
  Format-Table -AutoSize
```

How it works...

In *step 1*, you use `Get-SmbShare` to discover the current SMB shares on `SRV2` and which accounts have access to those shares. The output looks like this:

```
PS C:\Foo> # 1. Discovering existing shares and access rights
PS C:\Foo> Get-SmbShare -Name * |
             Get-SmbShareAccess |
               Format-Table -GroupBy Name

   Name: IPC$

Name ScopeName AccountName              AccessControlType AccessRight
---- --------- -----------              ----------------- -----------
IPC$ *         BUILTIN\Administrators   Allow             Full
IPC$ *         BUILTIN\Backup Operators Allow             Full
IPC$ *         NT AUTHORITY\INTERACTIVE Allow             Full
```

Figure 10.10: Discovering SMB shares and access rights on SRV2

In *step 2*, you create a new folder on the `C:\` drive and share the folder as `\\SRV2\ITShare`. The output from this step looks like this:

```
PS C:\Foo> # 2. Sharing a new folder
PS C:\Foo> New-Item -Path C: -Name ITShare | Out-Null
PS C:\Foo> New-SmbShare -Name ITShare -Path C:\ITShare

Name    ScopeName Path       Description
----    --------- ----       -----------
ITShare *         C:\ITShare
```

Figure 10.11: Creating and sharing C:\ITShare as \\SRV2\ITShare

Having created the share, you next configure share access details. In *step 3*, you modify the share so that it has a description. With *step 4*, you set access-based enumeration on the share. Then, in *step 5*, you ensure Windows encrypts all data that's transferred via the share. Next, in *step 6*, you remove access to the `ITShare` for the `Everyone` group. In *step 7*, you grant the `Reskit\Administrator` group read permission on the `ITShare`. With *step 8*, you give the OS full access to the share. In *step 9*, you grant the creator or owner of any file/folder full access to the file. Finally, in *step 10*, you grant the Sales group read access. These eight configuration steps produce no output.

In *step 11*, you review the access to the share, which produces output like this:

```
PS C:\Foo> # 11. Reviewing share access
PS C:\Foo> Get-SmbShareAccess -Name ITShare |
             Sort-Object AccessRight

Name      ScopeName AccountName           AccessControlType AccessRight
----      --------- -----------           ----------------- -----------
ITShare   *         NT AUTHORITY\SYSTEM   Allow             Full
ITShare   *         CREATOR OWNER         Allow             Full
ITShare   *         RESKIT\Administrator  Allow             Read
ITShare   *         RESKIT\Sales          Allow             Read
```

Figure 10.12: Viewing access to ITShare

Now that you have access to the share configured as needed, in *step 12*, you use the `Set-SMBPathAcl` command to make the NTFS permissions match the SMB share permissions. In *step 13*, you create a new file in the folder shared as `ITShare` and then ensure, in *step 14*, that the file itself has the same ACL as the share. These three steps produce no output.

In *step 15*, you view the file, `C:\ITShare\File.txt`, which produces output like this:

```
PS C:\Foo> # 15. Viewing file ACL
PS C:\Foo> Get-NTFSAccess -Path C:\ITShare\File.Txt |
             Format-Table -AutoSize

    Path: F:\ITShare\File.Txt (Inheritance enabled)


Account                Access Rights                Applies to     Type  IsInherited InheritedFrom
-------                -------------                ----------     ----  ----------- -------------
BUILTIN\Administrators FullControl                  ThisFolderOnly Allow True        F:\ITShare
NT AUTHORITY\SYSTEM    FullControl                  ThisFolderOnly Allow True        F:\ITShare
RESKIT\Administrator   ReadAndExecute, Synchronize ThisFolderOnly Allow True        F:\ITShare
RESKIT\Sales           ReadAndExecute, Synchronize ThisFolderOnly Allow True        F:\ITShare
BUILTIN\Users          ReadAndExecute, Synchronize ThisFolderOnly Allow True        F:
```

Figure 10.13: Viewing access to the file

There's more...

In *step 1*, you examine the shares available on `SRV2`. In the *Setting up and securing an SMB file server* recipe, you configured the SMB service to remove the default shares on `SRV2`. The only share you see in *step 1* is the `IPC$` share, which Windows uses for the named pipes communication mechanism. For more details about this share, see `https://docs.microsoft.com/troubleshoot/windows-server/networking/inter-process-communication-share-null-session`.

In *step 4*, you set access-based enumeration for the `ITShare` share. This setting means that any user viewing files or folders within the share only sees objects to which they have access. This setting improves security and minimizes administrative questions such as "What is this file/folder, and why can't I have access to this file/folder?"

In *step 5*, you set encryption to on for the `ITShare` share. This step ensures that Windows performs data encryption on any data transferred across this share. You can set this by default at the server level or, as in this case, at the share level.

Accessing SMB shares

In the *Creating and securing SMB shares* recipe, you created a share on `SRV2`. Data you access using SMB file sharing acts and feels like accessing local files via Windows Explorer or the PowerShell console.

In this recipe, you access the `ITShare` share on `SRV2` from `SRV1`.

Getting ready

This recipe uses `SRV1`, a domain-joined host in the `Reskit.Org` domain, on which you have installed PowerShell 7 and VS Code. You also use `SRV2` and should have `DC1` online. You previously created SMB shares on `SRV2`, which you use in this recipe.

How to do it...

1. Examining the SMB client's configuration on `SRV1`

   ```
   Get-SmbClientConfiguration
   ```

2. Setting signing of SMB packets

   ```
   $CHT = @{Confirm=$false}
   Set-SmbClientConfiguration -RequireSecuritySignature $True @CHT
   ```

3. Examining the SMB client's network interface

   ```
   Get-SmbClientNetworkInterface |
       Format-Table
   ```

4. Examining the shares provided by `SRV2`

   ```
   net view \\SRV2
   ```

5. Creating a drive mapping, mapping `R:` to the share on server `SRV2`

   ```
   New-SmbMapping -LocalPath R: -RemotePath \\SRV2\ITShare
   ```

6. Viewing the shared folder mapping

   ```
   Get-SmbMapping
   ```

7. Viewing the shared folder contents

   ```
   Get-ChildItem -Path R:
   ```

8. Viewing existing connections

```
Get-SmbConnection
```

How it works...

In *step 1*, you examine SMB client configuration on SRV1. The output looks like this:

```
PS C:\Foo> # 1. Examining the SMB client's configuration on SRV1
PS C:\Foo> Get-SmbClientConfiguration

SkipCertificateCheck                  : False
ConnectionCountPerRssNetworkInterface : 4
DirectoryCacheEntriesMax              : 16
DirectoryCacheEntrySizeMax            : 65536
DirectoryCacheLifetime                : 10
DormantFileLimit                      : 1023
EnableBandwidthThrottling             : True
EnableByteRangeLockingOnReadOnlyFiles : True
EnableInsecureGuestLogons             : True
EnableLargeMtu                        : True
EnableLoadBalanceScaleOut             : True
EnableMultiChannel                    : True
EnableSecuritySignature               : True
ExtendedSessionTimeout                : 1000
FileInfoCacheEntriesMax               : 64
FileInfoCacheLifetime                 : 10
FileNotFoundCacheEntriesMax           : 128
FileNotFoundCacheLifetime             : 5
ForceSMBEncryptionOverQuic            : False
KeepConn                              : 600
MaxCmds                               : 50
MaximumConnectionCountPerServer       : 32
OplocksDisabled                       : False
RequireSecuritySignature              : False
SessionTimeout                        : 60
UseOpportunisticLocking               : True
WindowSizeThreshold                   : 1
```

Figure 10.14: Examining the SMB client's configuration

In *step 2*, you ensure that SRV1 requires signed SMB packets, irrespective of the settings on the SMB server (SRV2). There is no output from this step.

In *step 3*, you examine the client NIC on SRV1, with output that looks like this:

```
PS C:\Foo> # 3. Examining the SMB client's network interface
PS C:\Foo> Get-SmbClientNetworkInterface |
             Format-Table

Interface Index RSS Capable RDMA Capable Speed   IpAddresses                                   Friendly Name
--------------- ----------- ------------ -----   -----------                                   -------------
14              True        False        10 Gbps {fe80::71a9:bdb8:2ad4:410b, 10.10.10.50}      Ethernet
```

Figure 10.15: Viewing NIC on SRV1

In *step 4*, you use the `net.exe` command to view the shares provided by the `SRV2` host. The output from this step looks like this:

```
PS C:\Foo> # 4. Examining the shares provided by SRV2
PS C:\Foo> net view \\SRV2
Shared resources at \\SRV2

Share name  Type  Used as  Comment
-------------------------------------------------------------------------------
ITShare     Disk           File Share for IT
The command completed successfully.
```

Figure 10.16: Viewing the shares offered by SRV2

In *step 5*, you create a new drive mapping on `SRV1`, mapping the `R:` drive to `\\SRV2\ITShare`, which creates output that looks like this:

```
PS C:\Foo> # 5. Creating a drive mapping, mapping R: to the share on server SRV2
PS C:\Foo> New-SmbMapping -LocalPath r: -RemotePath \\SRV2\ITShare

Status Local Path Remote Path
------ ---------- -----------
OK     R:         \\SRV2\ITShare
```

Figure 10.17: Creating a new drive mapping on SRV1

In *step 6*, you view the SMB drive mappings on `SRV1`, which looks like this:

```
PS C:\Foo> # 6. Viewing the shared folder mapping
PS C:\Foo> Get-SmbMapping

Status Local Path Remote Path
------ ---------- -----------
OK     R:         \\SRV2\ITShare
```

Figure 10.18: Viewing the SMB shares provided by SRV1

In *step 7*, you view the contents of the share to reveal the file you created in *Creating and securing SMB shares*. The output looks like this:

```
PS C:\Foo> # 7. Viewing the shared folder contents
PS C:\Foo> Get-ChildItem -Path R:

    Directory: R:\

Mode          LastWriteTime   Length Name
----          -------------   ------ ----
-a---   28/02/2021    17:27       15 File.Txt
```

Figure 10.19: Viewing the contents of the shared folder

In *step 8*, you view all existing SMB connections from `SRV1`. This step produces the following output:

```
PS C:\Foo> # 8. Viewing existing connections
PS C:\Foo> Get-SmbConnection

ServerName ShareName UserName             Credential               Dialect NumOpens
---------- --------- --------             ----------               ------- --------
SRV2       ITShare   RESKIT\Administrator RESKIT.ORG\Administrator 3.1.1   1
```

Figure 10.20: Viewing existing SMV connections from SRV1

There's more...

In *step 4*, you use the `net.exe` command to view the shares offered by `SRV2` (from `SRV1`). The `SMBShare` module does not provide a cmdlet that views shares offered by a remote host.

Creating an iSCSI target

iSCSI is an industry-standard protocol that implements block storage over a TCP/IP network. With iSCSI, the server or target provides a volume shared via iSCSI to an iSCSI client, also known as the initiator.

In the original SCSI protocol, you use the term **Logical Unit Number** (**LUN**) to refer to a single physical disk attached to the SCSI bus. With iSCSI, you give each remotely shared volume an iSCSI LUN. The iSCSI client then sees the LUN as just another disk device attached to the local system. From the iSCSI client, you can manage the disk just like locally attached storage. Windows Server 2022 includes both iSCSI target (server) and iSCSI initiator (client) features.

You set up an iSCSI target on a server and then use an iSCSI initiator on another server (or client) system to access the iSCSI target. You can use both Microsoft and third-party initiators and targets, although if you mix and match, you need to carefully test that the combination works in your environment.

With iSCSI, a target is a single disk that the client accesses using the iSCSI client. An iSCSI target server hosts one or more targets, where each iSCSI target is equivalent to a LUN on a Fiber Channel SAN.

You could use iSCSI in a cluster of Hyper-V servers. The servers in the cluster can use the iSCSI initiator to access an iSCSI target. Used via the Cluster Shared Volume, the shared iSCSI target is shared between nodes in a failover cluster, which enables the VMs in that cluster to be highly available.

Getting ready

This recipe uses `SS1`, a domain-joined host in the `Reskit.Org` domain, on which you have installed PowerShell 7 and VS Code.

How to do it...

1. Installing the iSCSI target feature on `SS1`

```
Import-Module -Name ServerManager -WarningAction SilentlyContinue
Install-WindowsFeature FS-iSCSITarget-Server
```

2. Exploring iSCSI target server settings

```
Get-IscsiTargetServerSetting
```

3. Creating a folder on `SS1` to hold the iSCSI virtual disk

```
$NIHT = @{
  Path        = 'C:\iSCSI'
  ItemType    = 'Directory'
  ErrorAction = 'SilentlyContinue'
}
New-Item @NIHT | Out-Null
```

4. Creating an iSCSI virtual disk (that is, a LUN)

```
$LP = 'C:\iSCSI\ITData.Vhdx'
$LN = 'ITTarget'
$VDHT = @{
   Path        = $LP
   Description = 'LUN For IT Group'
   SizeBytes   = 500MB
 }
New-IscsiVirtualDisk @VDHT
```

5. Setting the iSCSI target, specifying who can initiate an iSCSI connection

```
$THT = @{
  TargetName   = $LN
  InitiatorIds = 'DNSNAME:SRV1.Reskit.Org',
                 'DNSNAME:SRV2.Reskit.Org'
}
New-IscsiServerTarget @THT
```

6. Creating an iSCSI disk target mapping LUN name to a local path

```
Add-IscsiVirtualDiskTargetMapping -TargetName $LN -Path $LP
```

How it works...

In *step 1*, you install the iSCSI target feature on the SS1 server, with output like this:

```
PS C:\Foo> # 1. Installing the iSCSI target feature on SS1
PS C:\Foo> Import-Module -Name ServerManager -WarningAction SilentlyContinue
PS C:\Foo> Install-WindowsFeature FS-iSCSITarget-Server


Success Restart Needed Exit Code      Feature Result
------- -------------- ---------      --------------
True    No             Success        {File and iSCSI Services, File Server, iSCSI…
```

Figure 10.21: Installing the iSCSI target feature on SS1

In *step 2*, you examine the iSCSI target server settings, with output that looks like this:

```
PS C:\Foo> # 2. Exploring iSCSI target server settings
PS C:\Foo> Get-IscsiTargetServerSetting

RunspaceId              : 8bb152ae-1139-4be5-8a38-5547b911401c
ComputerName            : SS1.Reskit.Org
IsClustered             : False
Version                 : 10.0
DisableRemoteManagement : False
Portals                 : {+10.10.10.111:3260, +[fe80::2592:2f35:d386:a902%6]:3260, -[fe80::4988:1c17:ee70:2275%10]:3260}
```

Figure 10.22: Examining the iSCSI target server settings

In *step 3*, you create a folder on SS1 to hold the iSCSI virtual disk, which creates no output. In *step 4*, you create an iSCSI virtual disk (essentially a LUN), with output that looks like this:

```
PS C:\Foo> # 4. Creating an iSCSI virtual disk (that is a LUN)
PS C:\Foo> $LP = 'C:\iSCSI\ITData.Vhdx'
PS C:\Foo> $LN = 'ITTarget'
PS C:\Foo> $VDHT = @{
             Path        = $LP
             Description = 'LUN For IT Group'
             SizeBytes   = 500MB
           }
PS C:\Foo> New-IscsiVirtualDisk @VDHT

RunspaceId         : 8bb152ae-1139-4be5-8a38-5547b911401c
ClusterGroupName   :
ComputerName       : SS1.Reskit.Org
Description        : LUN For IT Group
DiskType           : Dynamic
HostVolumeId       : {EE0C6DD3-0000-0000-0000-100000000000}
LocalMountDeviceId :
OriginalPath       :
ParentPath         :
Path               : C:\iSCSI\ITData.Vhdx
SerialNumber       : 3471BFE1-391F-42E5-A5F1-82165C29CBAC
Size               : 524288000
SnapshotIds        :
Status             : NotConnected
```

Figure 10.23: Creating an iSCSI virtual disk on SS1

In *step 5*, you specify which computers can use the virtual iSCSI target, with output like this:

```
PS C:\Foo> # 5. Set the iSCSI target, specifying who can initiate an iSCSI connection
PS C:\Foo> $THT = @{
             TargetName   = $LN
             InitiatorIds = 'DNSNAME:SRV1.Reskit.Org',
                            'DNSNAME:SRV2.Reskit.Org'
           }
PS C:\Foo> New-IscsiServerTarget @THT

RunspaceId                  : 8bb152ae-1139-4be5-8a38-5547b911401c
ChapUserName                :
ClusterGroupName            :
ComputerName                : SS1.Reskit.Org
Description                 :
EnableChap                  : False
EnableReverseChap           : False
EnforceIdleTimeoutDetection : True
FirstBurstLength            : 65536
IdleDuration                : 00:00:00
InitiatorIds                : {DnsName:SRV1.Reskit.Org, DnsName:SRV2.Reskit.Org}
LastLogin                   :
LunMappings                 : {}
MaxBurstLength              : 262144
MaxReceiveDataSegmentLength : 65536
ReceiveBufferCount          : 10
ReverseChapUserName         :
Sessions                    : {}
Status                      : NotConnected
TargetIqn                   : iqn.1991-05.com.microsoft:ss1-salestarget-target
TargetName                  : ITTarget
```

Figure 10.24: Specifying which hosts can access the iSCSI virtual disk

In the final step, *step 6*, you specify the disk target mapping, which generates no output. This step creates a mapping between an iSCSI target name (`ITTarget`) and the local path where you stored the virtual iSCSI hard disk.

There's more...

By default, Windows does not install the iSCSI target feature, but as you can see in *step 1*, you use `Install-WindowsFeature` to add the feature to this storage server.

When you create an iSCSI target, you create the target name and the target virtual hard drive separately, and then, in *step 6*, you map the iSCSI target name to the file location. In production, you would use a separate set of (fault-tolerant) disks to hold the iSCSI information, possibly using Storage Spaces to create fault-tolerant virtual disks.

Using an iSCSI target

Windows provides a built-in iSCSI client component you use the `Connect-IscsiTarget` command to connect the iSCSI client to the iSCSI server and start using the disk as if it were attached locally.

Getting ready

This recipe uses SRV1, a domain-joined host in the Reskit.Org domain, on which you have installed PowerShell 7 and VS Code. You also use SS1 and should have DC1 online. You previously created an iSCSI target (on SS1), and now you use the built-in iSCSI initiator to access the iSCSI disk. You run this recipe on SRV1.

How to do it...

1. Adjusting the iSCSI service to auto-start, then starting the service

```
Set-Service MSiSCSI -StartupType 'Automatic'
Start-Service MSiSCSI
```

2. Setting up the portal to SS1

```
$PHT = @{
  TargetPortalAddress     = 'SS1.Reskit.Org'
  TargetPortalPortNumber  = 3260
}
New-IscsiTargetPortal @PHT
```

3. Finding and viewing the ITTarget on the portal

```
$Target  = Get-IscsiTarget |
               Where-Object NodeAddress -Match 'ITTarget'
$Target
```

4. Connecting to the target on SS1

```
$CHT = @{
  TargetPortalAddress = 'SS1.Reskit.Org'
  NodeAddress         = $Target.NodeAddress
}
Connect-IscsiTarget  @CHT
```

5. Viewing the iSCSI disk from SRV1 on SS1

```
$ISD =  Get-Disk |
  Where-Object BusType -eq 'iscsi'
$ISD |
  Format-Table -AutoSize
```

6. Turning disk online and making disk R/W

```
$ISD |
  Set-Disk -IsOffline  $False
$ISD |
  Set-Disk -Isreadonly $False
```

7. Formatting the volume on SS1

```
$NVHT = @{
  FriendlyName = 'ITData'
  FileSystem   = 'NTFS'
  DriveLetter  = 'I'
}
$ISD |
  New-Volume @NVHT
```

8. Using the drive as a local drive

```
Set-Location -Path I:
New-Item -Path I:\  -Name ITData -ItemType Directory |
  Out-Null
'Testing 1-2-3' |
  Out-File -FilePath I:\ITData\Test.Txt
Get-ChildItem I:\ITData
```

How it works...

In *step 1,* you set the iSCSI service to start when SRV1 starts automatically, and then you explicitly start the iSCSI service. This step creates no console output.

In *step 2*, you set up the iSCSI portal to SS1, which looks like this:

```
PS C:\Foo> # 2. Setting up the portal to SS1
PS C:\Foo> $PHT = @{
             TargetPortalAddress     = 'SS1.Reskit.Org'
             TargetPortalPortNumber  = 3260
           }
PS C:\Foo> New-IscsiTargetPortal @PHT

RunspaceId             : ded5213d-8a72-4752-9cb1-b388db48edc6
InitiatorInstanceName  :
InitiatorPortalAddress :
IsDataDigest           : False
IsHeaderDigest         : False
TargetPortalAddress    : SS1.Reskit.Org
TargetPortalPortNumber : 3260
```

Figure 10.25: Setting up the iSCSI portal to SS1

In *step 3*, you find and view the `ITTarget` LUN from `SS1`. The output looks like this:

```
PS C:\Foo> # 3. Finding and viewing the ITTarget on the portal
PS C:\Foo> $Target  = Get-IscsiTarget |
                        Where-Object NodeAddress -Match 'ITTarget'
PS C:\Foo> $Target

RunspaceId   : ded5213d-8a72-4752-9cb1-b388db48edc6
IsConnected : False
NodeAddress : iqn.1991-05.com.microsoft:ss1-ittarget-target
```

Figure 10.26: Viewing the ITTarget LUN

In *step 4*, you connect from `SRV1` to the iSCSI target on `SS1`, which looks like this:

```
PS C:\Foo> # 4. Connecting to the target on SS1
PS C:\Foo> $CHT = @{
             TargetPortalAddress = 'SS1.Reskit.Org'
             NodeAddress         = $Target.NodeAddress
           }
PS C:\Foo> Connect-IscsiTarget  @CHT
RunspaceId                : ded5213d-8a72-4752-9cb1-b388db48edc6
AuthenticationType        : NONE
InitiatorInstanceName     : ROOT\ISCSIPRT\0000_0
InitiatorNodeAddress      : iqn.1991-05.com.microsoft:srv1.reskit.org
InitiatorPortalAddress    : 0.0.0.0
InitiatorSideIdentifier : 400001370000
IsConnected               : True
IsDataDigest              : False
IsDiscovered              : False
IsHeaderDigest            : False
IsPersistent              : False
NumberOfConnections       : 1
SessionIdentifier         : ffff828526afb010-4000013700000012
TargetNodeAddress         : iqn.1991-05.com.microsoft:ss1-ittarget-target
TargetSideIdentifier      : 0100
```

Figure 10.27: Connecting to the iSCSI target on SS1

In *step 5*, you use `Get-Disk` to view the iSCSI disk from `SRV1`, which looks like this:

```
PS C:\Foo> # 5. Viewing the iSCSI disk
PS C:\Foo> $ISD =  Get-Disk |
             Where-Object BusType -eq 'iscsi'
PS C:\Foo> $ISD |
             Format-Table -AutoSize

Number Friendly Name   Serial Number                    HealthStatus OperationalStatus Total Size Partition Style
------ -------------   -------------                    ------------ ----------------- ---------- ---------------
11     MSFT Virtual HD 9BF969AC-7228-42F9-A994-4CFDA4EBC46D Healthy      Online              500 MB RAW
```

Figure 10.28: Viewing the iSCSI disk

In *step 6*, you ensure the iSCSI disk is online and Read/Write, a step that generates no output. In *step 7*, you create a new volume on the iSCSI disk, which looks like this:

```
PS C:\Foo> # 7. Formatting the volume on SS1
PS C:\Foo> $NVHT = @{
             FriendlyName = 'ITData'
             FileSystem   = 'NTFS'
             DriveLetter  = 'I'
           }
PS C:\Foo> $ISD |
             New-Volume @NVHT

DriveLetter FriendlyName FileSystemType DriveType HealthStatus OperationalStatus SizeRemaining      Size
----------- ------------ -------------- --------- ------------ ----------------- -------------      ----
I           ITData       NTFS           Fixed     Healthy      OK                    467.73 MB 483.93 MB
```

Figure 10.29: Formatting the iSCSI disk

In the final step of this recipe, *step 8*, you create a folder in the iSCSI disk. Then, you create a file and view the file, which looks like this:

```
PS C:\Foo> # 8. Using the drive as a local drive
PS C:\Foo> Set-Location -Path I:
PS I:\> New-Item -Path I:\  -Name ITData -ItemType Directory |
          Out-Null
PS I:\> 'Testing 1-2-3' |
          Out-File -FilePath I:\ITData\Test.Txt
PS I:\> Get-ChildItem I:\ITData

    Directory: I:\ITData

Mode           LastWriteTime    Length Name
----           -------------    ------ ----
-a---    02/03/2021    08:42        15 Test.Txt
```

Figure 10.30: Using the iSCSI disk

There's more...

Using an iSCSI disk is straightforward – connect to the iSCSI target and manage the disk volume locally. Once connected, you can format it with a filesystem and then use it to store data. In production, you may not be using a Windows Server host to serve as an iSCSI target. Many **storage area network** (**SAN**) vendors add iSCSI target features to their SAN offerings. With a SAN offering, you can use the Windows iSCSI initiator to access the SAN via iSCSI, although some SAN vendors may provide an updated iSCSI initiator for you to use.

Implementing FSRM quotas

The **File Server Resource Manager** (**FSRM**) is a feature of Windows Server that assists you in managing file servers. FSRM has three key features:

- **Quota management**: With FSRM, you can set soft or hard quotas on volumes and folders. Soft quotas allow a user to exceed an allowance, while hard quotas stop a user from exceeding an allowance. You can configure a quota with thresholds and threshold actions. If a user exceeds 65% of the quota allowance, FSRM can send an email, while at 90%, you log an event in the event log or run a program. You have different actions for different quota levels. This recipe shows you how to use quotas.
- **File screening**: You can set up a file screen and stop a user from saving screened files. For example, you could screen for `.MP3` or `FLAC` files. Should a user then attempt to save a file (say, `jg75-02-28D1T1.flac`), the file screen rejects the request and doesn't allow the user to save the file.
- **Reporting**: FSRM enables you to create a wealth of storage reports, which can be highly useful for management purposes.

In this recipe, you install FSRM, perform some general configuration, and then work with soft and hard quotas.

Getting ready

This recipe uses `SRV1`, a domain-joined host in the `Reskit.Org` domain, on which you have installed PowerShell 7 and VS Code. You should have `DC1` online to provide authentication for `SRV1`.

FSRM has features that send email messages to an SMTP server. To test these features, as shown in this recipe, you need an email server so that FSRM can send emails.

You can use Internet Information Server within Windows Server to forward emails to an SMTP email server. This recipe configures FSRM to send mail to a host (`SMTP.Reskit.Org`), which then forwards the mail to a free email service at `https://www.sendgrid.com`. For more details on how to set up SendGrid to forward the email, see `https://tfl09.blogspot.com/2020/04/setting-up-smtp-relay-using-sendgrid.html`. If you have an SMTP server that accepts mail from FSRM, then change this recipe to use that server.

How to do it...

1. Installing FS Resource Manager feature on `SRV1`

```
Import-Module -Name ServerManager -WarningAction 'SilentlyContinue'
$IHT = @{
  Name                    = 'FS-Resource-Manager'
```

```
  IncludeManagementTools = $True
  WarningAction          = 'SilentlyContinue'
}
Install-WindowsFeature @IHT
```

2. Setting SMTP settings in FSRM

```
$MHT = @{
  SmtpServer        = 'SMTP.Reskit.Org'
  FromEmailAddress  = 'FSRM@Reskit.Org'
  AdminEmailAddress = 'Doctordns@Gmail.Com'
}
Set-FsrmSetting @MHT
```

3. Sending and viewing a test email to check the setup

```
$MHT = @{
  ToEmailAddress = 'DoctorDNS@gmail.com'
  Confirm        = $false
}
Send-FsrmTestEmail @MHT
```

4. Creating a new FSRM quota template for a 10 MB hard limit

```
$QHT1 = @{
  Name        = '10 MB Reskit Quota'
  Description = 'Filestore Quota (10mb)'
  Size        = 10MB
}
New-FsrmQuotaTemplate @QHT1
```

5. Viewing available FSRM quota templates

```
Get-FsrmQuotaTemplate |
  Format-Table -Property Name, Description, Size, SoftLimit
```

6. Creating a new folder on which to apply a quota

```
If (-Not (Test-Path C:\Quota)) {
  New-Item -Path C:\Quota -ItemType Directory  |
    Out-Null
}
```

7. Building an FSRM action

```
$Body = @'
User [Source Io Owner] has exceeded the [Quota Threshold]% quota
threshold for the quota on [Quota Path] on server [Server].
The quota limit is [Quota Limit MB] MB, and [Quota Used MB] MB
currently is in use ([Quota Used Percent]% of limit).
```

```
'@
$NAHT = @{
  Type      = 'Email'
  MailTo    = 'Doctordns@gmail.Com'
  Subject   = 'FSRM Over limit [Source Io Owner]'
  Body      = $Body
}
$Action1 = New-FsrmAction @NAHT
```

8. Creating an FSRM threshold

```
$Thresh = New-FsrmQuotaThreshold -Percentage 85 -Action $Action1
```

9. Building a quota for the `C:\Quota` folder

```
$NQHT1 = @{
  Path      = 'C:\Quota'
  Template  = '10 MB Reskit Quota'
  Threshold = $Thresh
}
New-FsrmQuota @NQHT1
```

10. Testing the 85% soft quota limit on `C:\Quota`

```
Get-ChildItem -Path C:\Quota -Recurse |
  Remove-Item -Force     # for testing purposes!
$S = '+'.PadRight(8MB)
# Make a first file - under the soft quota
$S | Out-File -FilePath C:\Quota\Demo1.Txt
$S2 = '+'.PadRight(.66MB)
# Now create a second file to take the user over the soft quota
$S2 | Out-File -FilePath C:\Quota\Demo2.Txt
```

11. Testing hard quota limit

```
$S | Out-File -FilePath C:\Quota\Demo3.Txt
```

12. Viewing the contents of the `C:\Quota` folder

```
Get-ChildItem -Path C:\Quota
```

How it works...

In *step 1*, you use the `Install-WindowsFeature` cmdlet to add the `FS-Resource-Manager` feature to `SRV1`, which looks like this:

```
PS C:\Foo> # 1. Installing FS Resource Manager feature on SRV1
PS C:\Foo> Import-module -Name ServerManager -WarningAction 'SilentlyContinue'
PS C:\Foo> $IHT = @{
             Name                  = 'FS-Resource-Manager'
             IncludeManagementTools = $True
             WarningAction         = 'SilentlyContinue'
           }
PS C:\Foo> Install-WindowsFeature @IHT

Success Restart Needed Exit Code   Feature Result
------- -------------- ---------   --------------
True    No             Success     {File Server Resource Manager}
```

Figure 10.31: Installing the FSRM feature for SRV1

In *step 2*, you set SMTP details, including the SMTP server name and the `From` and `Admin` addresses. This step produces no output. In *step 3*, you use the `Send-FsrmTestEmail` cmdlet to test SMTP email handling. This step produces no console output but does generate an email, which looks like this:

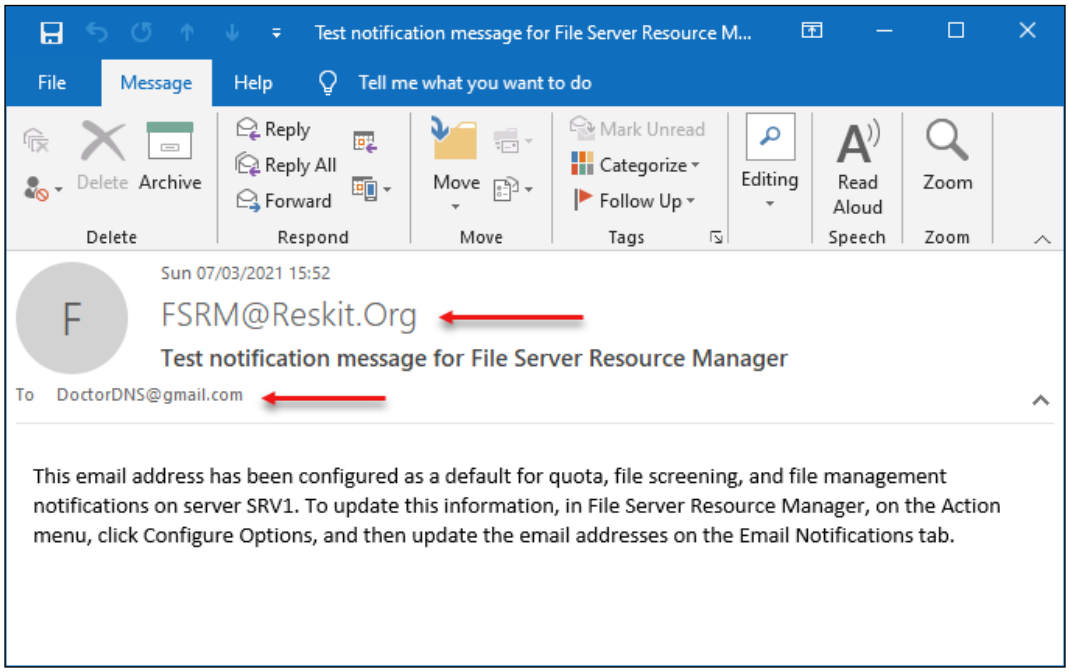


Figure 10.32: Test email received from FSRM

In *step 4*, you create a new FSRM quota template for a 10 MB hard quota limit. The output from this step looks like this:

```
PS C:\Foo> # 4. Creating a new FSRM quota template for a 10MB hard limit
PS C:\Foo> $QHT1 = @{
             Name        = '10 MB Reskit Quota'
             Description = 'Filestore Quota (10mb)'
             Size        = 10MB
           }
PS C:\Foo> New-FsrmQuotaTemplate @QHT1

Description           : Filestore Quota (10mb)
Name                  : 10 MB Reskit Quota
Size                  : 10485760
SoftLimit             : False
Threshold             :
UpdateDerived         : False
UpdateDerivedMatching : False
PSComputerName        :
```

Figure 10.33: Creating an FSRM quota template

In *step 5*, you view all available FSRM quota templates, with output like this:

```
PS C:\Foo> # 5. Viewing available FSRM quota templates
PS C:\Foo> Get-FsrmQuotaTemplate |
             Format-Table -Property Name, Description, Size, SoftLimit

Name                                Description                    Size SoftLimit
----                                -----------                    ---- ---------
100 MB Limit                                                  104857600     False
200 MB Limit Reports to User                                  209715200     False
Monitor 200 GB Volume Usage                                214748364800      True
Monitor 500 MB Share                                          524288000      True
200 MB Limit with 50 MB Extension                             209715200     False
250 MB Extended Limit                                         262144000     False
2 GB Limit                                                   2147483648     False
5 GB Limit                                                   5368709120     False
10 GB Limit                                                 10737418240     False
Monitor 3 TB Volume Usage                                 3298534883328      True
Monitor 5 TB Volume Usage                                 5497558138880      True
Monitor 10 TB Volume Usage                               10995116277760      True
10 MB Reskit Quota                  Filestore Quota (10mb)     10485760     False
```

Figure 10.34: Viewing the available FSRM quota templates

In *step 6*, you create a new folder, `C:\Quota`. In *step 7*, you build an FSRM action that sends an email whenever a user exceeds the quota. In *step 8*, you create an FSRM threshold (how much of the soft quota limit a user can use before triggering a quota violation). These three steps produce no console output.

In *step 9*, you build a quota for the `C:\Quota` folder, with output that looks like this:

```
PS C:\Foo> # 9. Building a quota for the C:\Quota folder
PS C:\Foo> $NQHT1 = @{
             Path      = 'C:\Quota'
             Template  = '10 MB Reskit Quota'
             Threshold = $Thresh
           }
PS C:\Foo> New-FsrmQuota @NQHT1

Description     :
Disabled        : False
MatchesTemplate : False
Path            : C:\Quota
PeakUsage       : 1024
Size            : 10485760
SoftLimit       : False
Template        : 10 MB Reskit Quota
Threshold       : {MSFT_FSRMQuotaThreshold}
Usage           : 1024
PSComputerName  :
```

Figure 10.35: Building a quota for the C:\Quota folder

In *step 10*, you test the 85% soft quota limit. First, you create a new file (`C:\Quota\Demo1.Txt`) that is under the soft quota limit. Then, you create a second file (`C:\Quota\Demo2.Txt`) that uses up more than the soft quota limit. There is no console output from this step, but FSRM detects you have exceeded the soft quota limit for this folder and generates an email message that looks like this:

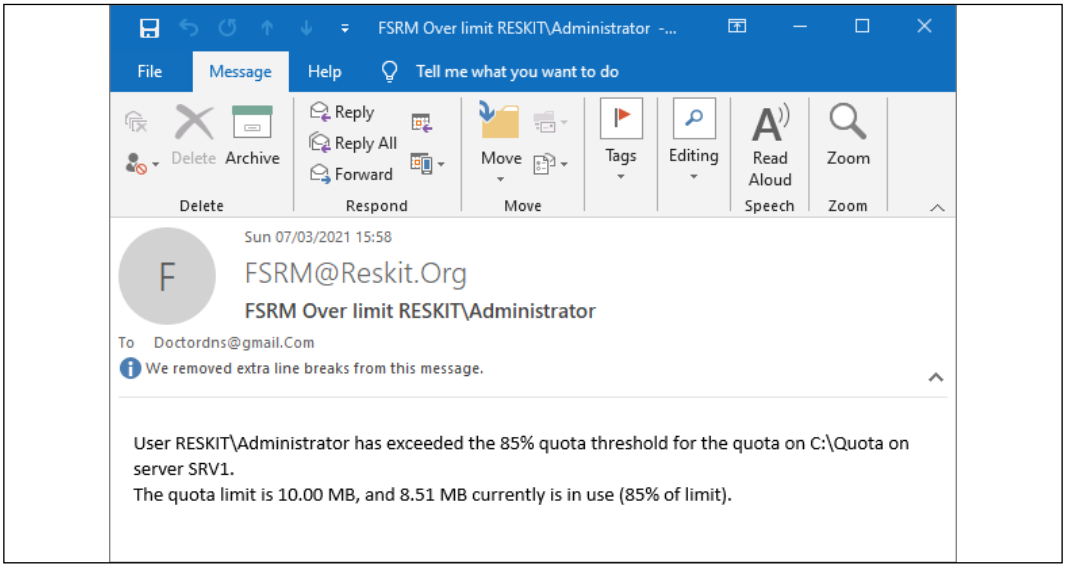


Figure 10.36: Exceeding soft quota limit email

In *step 11*, you attempt to create an additional file, `C:\Quota\Demo3.txt`, by outputting the `$S` array to a file, which results in you exceeding the hard quota limit. You should see the following output:

```
PS C:\Foo> # 11. Testing hard limit quota
PS C:\Foo> $S | Out-File -FilePath C:\Quota\Demo3.Txt
out-lineoutput: There is not enough space on the disk. : 'C:\Quota\Demo3.Txt'
```

Figure 10.37: Testing hard quota limit

In *step 12*, you examine the files in the `C:\Quota` folder, which looks like this:

```
PS C:\Foo> # 12. Viewing the contents of the C:\Quota folder
PS C:\Foo> Get-ChildItem -Path C:\Quota

    Directory: C:\Quota

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a---          07/03/2021     15:57        8388610 Demo1.Txt
-a---          07/03/2021     15:57         692062 Demo2.Txt
-a---          07/03/2021     16:23        1441792 Demo3.Txt
```

Figure 10.38: Viewing the C:\Quota folder

There's more...

In this recipe, you set up and test both a soft and a hard FSRM quota. With the soft quota, you configure FSRM to send an email to inform the recipient that they have exceeded a quota. You might want to send an email to either an administrator or a user who has exceeded the quota thresholds. For the hard quota, FSRM writes application event log entries and stops the user from saving excess data. The quotas set in this recipe are small and would probably not be of much use in production. But a change from, say, 10 MB to 10 GB would be simple to make.

In *step 1*, you install a new Windows feature to `SRV1`. From time to time, you may see the installation process just stall and not complete. In such cases, rerunning the command in a new PowerShell console or rebooting the server enables you to add features.

In *step 4*, you create a new FSRM quota template. You can see this new template in the output generated by *step 5*. Note that this quota template is for a hard quota limit, not a soft quota limit.

In *step 12*, you examine the `C:\Quota` folder. Notice that with the third file (which you attempted to create in *step 11*), Windows has not saved the entire file. If you are planning on imposing hard quotas, you must ensure your users understand the implications of exceeding any hard quota limits.

Implementing FSRM reporting

A useful and often overlooked feature of the FSRM component is reporting. FSRM defines several basic report types that you can use. These reports can either be generated immediately (also known as Interactive) or at a scheduled time. The latter causes FSRM to generate reports on a weekly or monthly basis.

FSRM produces reports with a fixed layout that you cannot change. FSRM can return the same data contained in the HTML report but as an XML document. You can then use that XML document to create a report the way you need it.

Getting ready

This recipe uses `SRV1`, a domain-joined host in the `Reskit.Org` domain, on which you have installed PowerShell 7 and VS Code. In the previous recipe, *Implementing FSRM quotas*, you installed FSRM on `SRV1`.

How to do it...

1. Creating a new FSRM storage report for large files on `C:\` on `SRV1`

```
$NRHT = @{
  Name             = 'Large Files on SRV1'
  NameSpace        = 'C:\'
  ReportType       = 'LargeFiles'
  LargeFileMinimum = 10MB
  Interactive      = $true
}
New-FsrmStorageReport @NRHT
```

2. Getting existing FSRM reports

```
Get-FsrmStorageReport * |
  Format-Table -Property Name, NameSpace,
                         ReportType, ReportFormat
```

3. Viewing Interactive reports available on `SRV1`

```
$Path = 'C:\StorageReports\Interactive'
Get-ChildItem -Path $Path
```

4. Viewing the report

```
$Rep = Get-ChildItem -Path $Path\*.html
Invoke-Item -Path $Rep
```

5. Extracting key information from the FSRM XML output

```
$XF   = Get-ChildItem -Path $Path\*.xml
$XML  = [XML] (Get-Content -Path $XF)
$Files = $XML.StorageReport.ReportData.Item
$Files | Where-Object Path -NotMatch '^Windows|^Program|^Users'|
  Format-Table -Property name, path,
    @{ Name ='Sizemb'
       Expression = {(([int]$_.size)/1mb).tostring('N2')}},
       DaysSinceLastAccessed -AutoSize
```

6. Creating a monthly task in the task scheduler

```
$Date = Get-Date '04:20'
$NTHT = @{
  Time    = $Date
  Monthly = 1
}
$Task = New-FsrmScheduledTask @NTHT
$NRHT = @{
  Name             = 'Monthly Files by files group report'
  Namespace        = 'C:\'
  Schedule         = $Task
  ReportType       = 'FilesbyFileGroup'
  FileGroupINclude = 'Text Files'
  LargeFileMinimum = 25MB
}
New-FsrmStorageReport @NRHT | Out-Null
```

7. Getting details of the task

```
Get-ScheduledTask |
  Where-Object TaskName -Match 'Monthly' |
    Format-Table -AutoSize
```

8. Running the task now

```
Get-ScheduledTask -TaskName '*Monthly*' |
  Start-ScheduledTask
Get-ScheduledTask -TaskName '*Monthly*'
```

9. Viewing the report in the `C:\StorageReports` folder

```
$Path = 'C:\StorageReports\Scheduled'
$Rep = Get-ChildItem -Path $path\*.html
$Rep
```

10. Viewing the report

```
Invoke-item -Path $Rep
```

How it works...

In *step 1*, you create a new FSRM report to discover large files (over 10 MB in size) on the `C:\` drive. The output from this step looks like this:

```
PS C:\Foo> # 1. Creating a new FSRM storage report for large files on C:\ on SRV1
PS C:\Foo> $NRHT = @{
             Name             = 'Large Files on SRV1'
             NameSpace        = 'C:\'
             ReportType       = 'LargeFiles'
             LargeFileMinimum = 10MB
             Interactive      = $true
           }
PS C:\Foo> New-FsrmStorageReport @NRHT

FileGroupIncluded        :
FileOwnerFilePattern     :
FileOwnerUser            :
FileScreenAuditDaysSince : 0
FileScreenAuditUser      :
FolderPropertyName       :
Interactive              : True
LargeFileMinimum         : 10485760
LargeFilePattern         :
LastError                :
LastReportPath           :
LastRun                  :
LeastAccessedFilePattern :
LeastAccessedMinimum     : 0
MailTo                   :
MostAccessedFilePattern  :
MostAccessedMaximum      : 0
Name                     : Large Files on SRV1
Namespace                : {C:\}
PropertyFilePattern      :
PropertyName             :
QuotaMinimumUsage        : 0
ReportFormat             : {DHtml, XML}
ReportType               : LargeFiles
Schedule                 :
Status                   : Queued
PSComputerName           :
```

Figure 10.39: Creating a new FSRM storage report

In *step 2*, you view the available FSRM reports, with output like this:

```
PS C:\Foo> # 2. Getting existing FSRM reports
PS C:\Foo> Get-FsrmStorageReport * |
             Format-Table -Property Name, NameSpace,
                                    ReportType, ReportFormat

Name                 NameSpace ReportType ReportFormat
----                 --------- ---------- ------------
Large Files on SRV1 {C:\}      LargeFiles {DHtml, XML}
```

Figure 10.40: Viewing the available FSRM storage reports

In *step 3*, you examine the reports that have completed and been output in the `C:\StorageReports` folder. The output looks like this:

```
PS C:\Foo> # 3. Viewing Interactive reports available on SRV1
PS C:\Foo> $Path = 'C:\StorageReports\Interactive'
PS C:\Foo> Get-ChildItem -Path $Path

    Directory: C:\StorageReports\Interactive

Mode           LastWriteTime   Length Name
----           -------------   ------ ----
d----    08/03/2021    11:22          LargeFiles4_2021-03-08_11-22-21_files
-a---    08/03/2021    11:22   262395 LargeFiles4_2021-03-08_11-22-21.html
-a---    08/03/2021    11:22   478068 LargeFiles4_2021-03-08_11-22-21.xml
```

Figure 10.41: Viewing the completed FSRM storage reports

In *step 4*, you examine the large file report in your default browser, which looks like this:

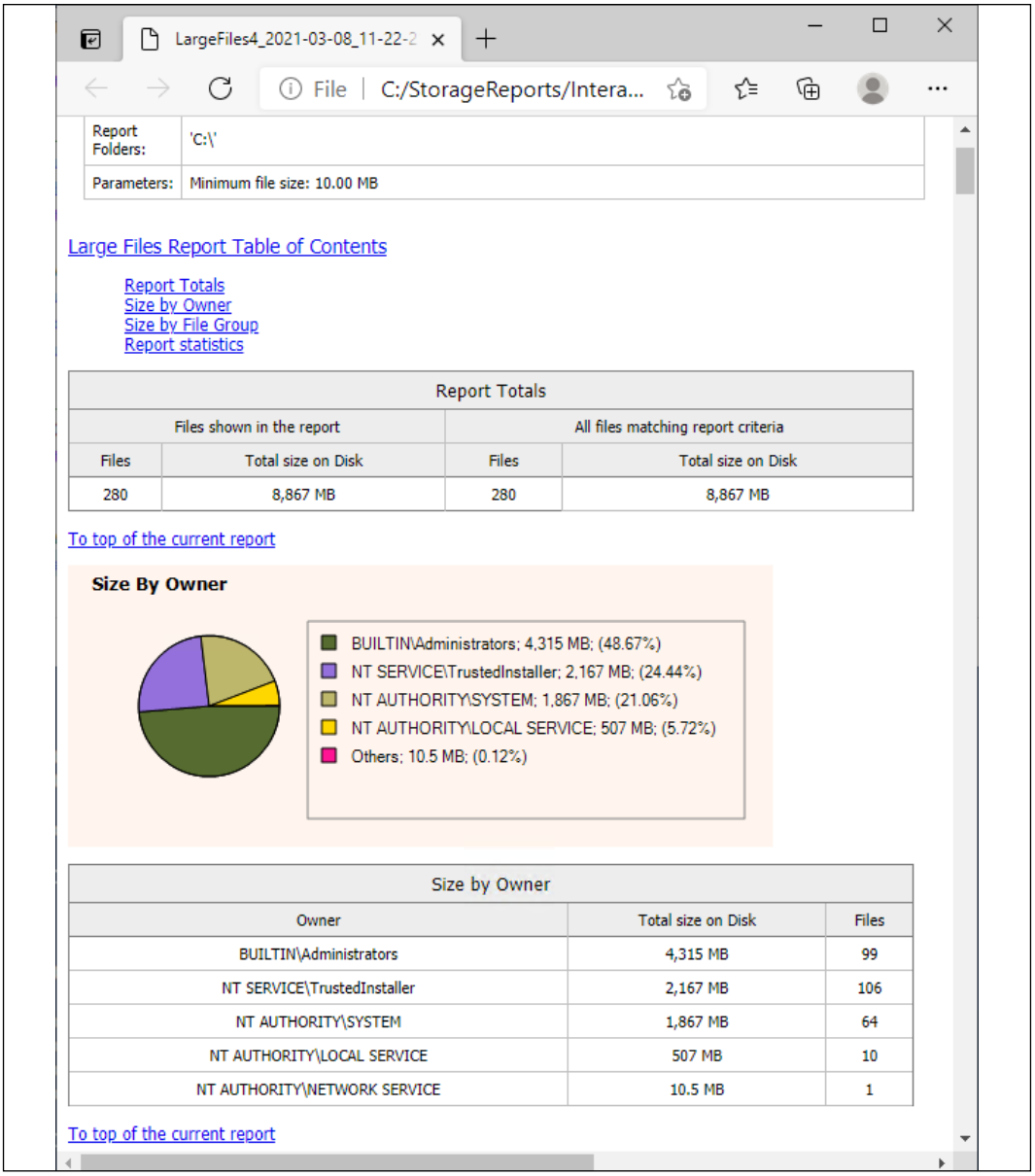


Figure 10.42: Viewing the large file report from SRV1

In *step 5*, you extract the critical information from the report XML file and output it to the console. The output looks like this:

```
PS C:\Foo> # 5. Extracting key information from the FSRM XML output
PS C:\Foo> $XF   = Get-ChildItem -Path $Path\*.xml
PS C:\Foo> $XML  = [XML] (Get-Content -Path $XF)
PS C:\Foo> $Files = $XML.StorageReport.ReportData.Item
PS C:\Foo> $Files | Where-Object Path -NotMatch '^Windows|^Program|^Users'|
            Format-Table -Property name, path,
               @{ Name ='Sizemb'
                  Expression = {([int]$_.size)/1mb).tostring('N2')}},
               DaysSinceLastAccessed -AutoSize

Name                                   Path                                Sizemb    DaysSinceLastAccessed
----                                   ----                                ------    ---------------------
pagefile.sys                                                               1,472.00 1
3{3808876b-c176-4e48-b7ae-04046e6cc752} System Volume Information          320.00   0
System.Management.Automation.dll       PSPREVIEW                           20.16    23
System.Management.Automation.dll       DAILYBUILD                          20.15    23
System.Management.Automation.dll       PSDAILYBUILD                        20.15    23
System.Management.Automation.dll       $Recycle.Bin\S-1-5-21-1877631356-...  20.01    41
System.Management.Automation.dll       $Recycle.Bin\S-1-5-21-1877631356-...  20.01    41
PresentationFramework.dll              $Recycle.Bin\S-1-5-21-1877631356-...  15.30    41
PresentationFramework.dll              $Recycle.Bin\S-1-5-21-1877631356-...  15.30    41
PresentationFramework.dll              PSPREVIEW                           15.13    23
PresentationFramework.dll              DAILYBUILD                          15.12    23
PresentationFramework.dll              PSDAILYBUILD                        15.12    23
Microsoft.CodeAnalysis.CSharp.dll      $Recycle.Bin\S-1-5-21-1877631356-...  14.93    41
Microsoft.CodeAnalysis.CSharp.dll      $Recycle.Bin\S-1-5-21-1877631356-...  14.92    41
Microsoft.CodeAnalysis.CSharp.dll      PSPREVIEW                           14.89    23
Microsoft.CodeAnalysis.CSharp.dll      DAILYBUILD                          14.88    23
Microsoft.CodeAnalysis.CSharp.dll      PSDAILYBUILD                        14.88    23
System.Windows.Forms.dll               $Recycle.Bin\S-1-5-21-1877631356-...  12.39    41
System.Windows.Forms.dll               $Recycle.Bin\S-1-5-21-1877631356- ...  12.39    41
System.Windows.Forms.dll               PSPREVIEW                           12.37    23
System.Windows.Forms.dll               DAILYBUILD                          12.36    23
System.Windows.Forms.dll               PSDAILYBUILD                        12.36    23
```

Figure 10.43: Viewing large file information

In *step 6*, you create a new scheduled task to run monthly. The task runs the `FilesbyFileGroup` report. This step creates no output.

In *step 7*, you examine the details of the scheduled task, with output like this:

```
PS C:\Foo> # 7. Getting details of the task
PS C:\Foo> Get-ScheduledTask |
             Where-Object TaskName -Match 'Monthly' |
               Format-Table -AutoSize

TaskPath                                        TaskName                                                State
--------                                        --------                                                -----
\Microsoft\Windows\File Server Resource Manager\ StorageReport-Monthly Files by files group report Ready
```

Figure 10.44: Getting details of the scheduled task

In *step 8*, you execute the scheduled task immediately, with output like this:

```
PS C:\Foo> # 8. Running the task now
PS C:\Foo> Get-ScheduledTask -TaskName '*Monthly*' |
             Start-ScheduledTask
PS C:\Foo> Get-ScheduledTask -TaskName '*Monthly*'

TaskPath                                    TaskName                       State
--------                                    --------                       -----
\Microsoft\Windows\File Server Resource Manag… StorageReport-Monthly Files by f… Ready
```

Figure 10.45: Executing the scheduled FSRM task

In *step 9*, after FSRM has finished running the report, you view the report output, which looks like this:

```
PS C:\Foo> # 9. Viewing the report in the StorageReports folder
PS C:\Foo> $Path = 'C:\StorageReports\Scheduled'
PS C:\Foo> $Rep = Get-ChildItem -Path $path\*.html
PS C:\Foo> $Rep

    Directory: C:\StorageReports\Scheduled

Mode          LastWriteTime   Length  Name
----          -------------   ------  ----
-a---   08/03/2021    11:37    93994  FilesByType5_2021-03-08_11-36-55.html
```

Figure 10.46: Viewing the report data

In the final step of this recipe, *step 10*, you view the report in the browser, which looks like this:

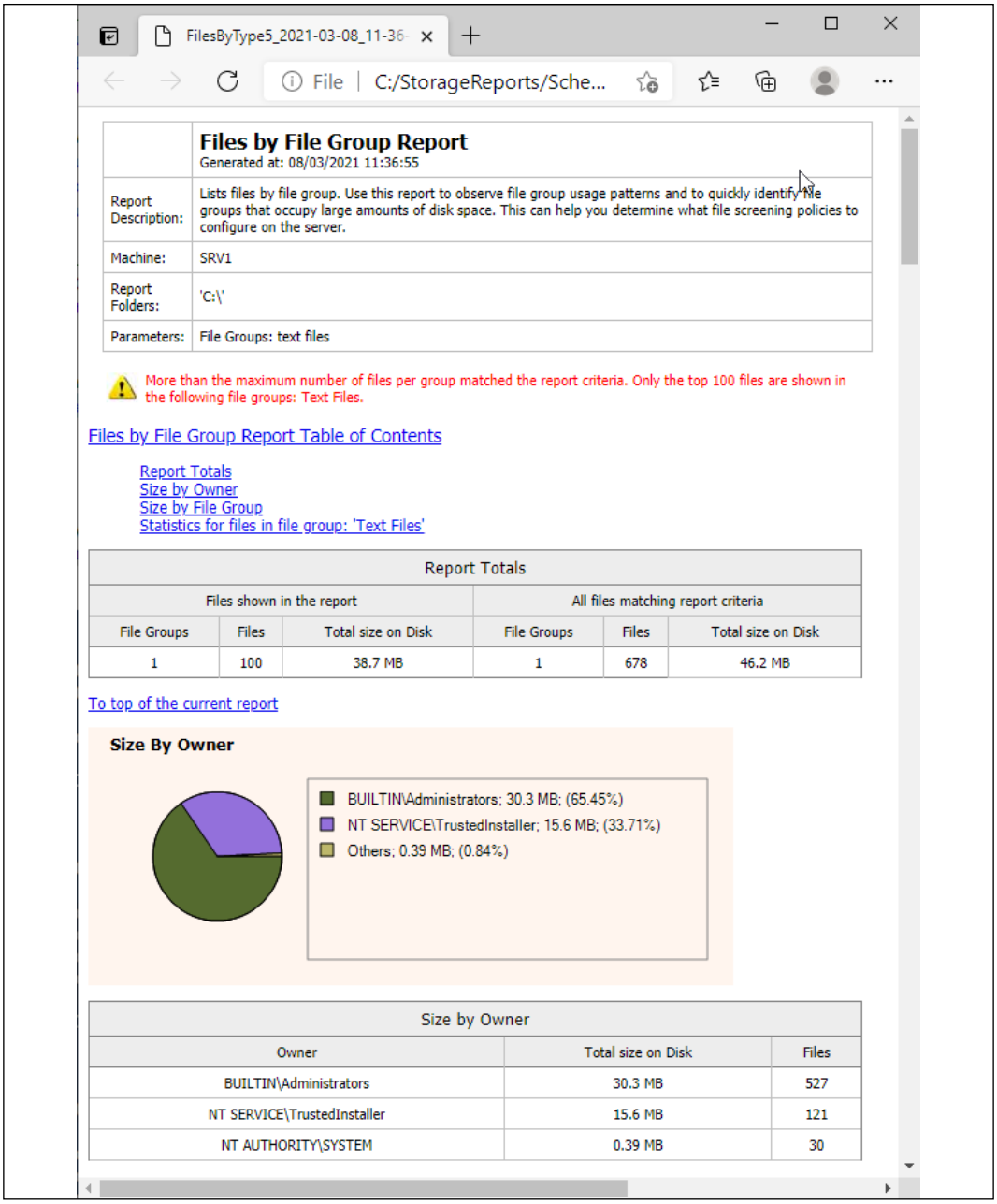


Figure 10.47: Viewing the report in the browser

There's more...

In *step 1*, you create a new FSRM interactive report. FSRM starts running this command immediately. When you view the report content folder, for example, in *step 3*, you may initially see no report output. It can take FSRM some time to produce the report, so you need to be patient.

In *step 4*, you view the HTML report created by FSRM using your default browser. Depending on the configuration of your host, you may see a prompt asking which application you wish to use to view the report.

As you can see from this recipe, FSRM creates report output in both HTML and XML format. You cannot change the HTML format, but it is probably good enough for most uses. If you want a specific format or just some of the data, you can get the same information from XML and format it to suit your needs.

Implementing FSRM file screening

FSRM has a file screening option. This feature allows you to control the types of files you allow to be stored on your file server. You could, for example, define a file screen to prohibit a user from saving music files (files with the `.MP3` or `.FLAC` extension) to your file server. With file screening, if a user attempts to save a file such as `GD71-02-18.T09.FLAC`, FSRM prevents the saving of the file.

To configure FSRM file screening, you need to specify the folder FSRM should protect and a file screen template that describes the characteristics of files that FSRM should block. FSRM comes with five built-in file screen templates. You can create additional templates to suit your requirements.

Each file screen template contains a set of file groups. Each file group defines a set of file extensions that FSRM can block. FSRM comes with 11 built-in file groups that cover common content types and can be updated and extended.

One built-in FSRM file group is audio and video files. This group includes a wide variety of audio and video file extensions, including `.AAC`, `.MP3`, `.FLAC`, and more. Interestingly, this built-in file group does not block `.SHN` (Shorten) files. You could easily add this extension to the relevant file group, should you wish.

Note that file screening works solely based on file extensions. FSRM, for example, might block you from saving a file such as `GD71-02-18.T09.FLAC`. However, if you tried to store this file as `GD71-02-18.T09.CALF`, FSRM would allow the file to be stored. FSRM file screening does not examine the file to ascertain the actual file type. In most cases, file screening stops the more obvious policy infractions.

Getting ready

This recipe uses SRV1, a domain-joined host in the Reskit.Org domain, on which you have installed PowerShell 7 and VS Code. In a previous recipe, *Implementing FSRM quotas*, you installed FSRM on SRV1.

How to do it...

1. Examining the existing file groups

```
Get-FsrmFileGroup |
  Format-Table -Property Name, IncludePattern
```

2. Examining the existing file screening templates

```
Get-FsrmFileScreenTemplate |
  Format-Table -Property Name, IncludeGroup, Active
```

3. Creating a new folder

```
$Path = 'C:\FileScreen'
If (-Not (Test-Path $Path)) {
  New-Item -Path $Path -ItemType Directory  |
    Out-Null
}
```

4. Creating a new file screen

```
$FSHT =  @{
  Path         = $Path
  Description  = 'Block Executable Files'
  IncludeGroup = 'Executable Files'
}
New-FsrmFileScreen @FSHT
```

5. Testing file screen by copying notepad.exe

```
$FSTHT = @{
  Path        = "$Env:windir\notepad.exe"
  Destination = 'C:\FileScreen\notepad.exe'
}
Copy-Item  @FSTHT
```

6. Setting up an active email notification

```
$Body =
"[Source Io Owner] attempted to save an executable program to
 [File Screen Path].

This is not allowed!
```

```
"
$FSRMA = @{
  Type             = 'Email'
  MailTo           = 'DoctorDNS@Gmail.Com'
  Subject          = 'Warning: attempted to save an executable file'
  Body             = $Body
  RunLimitInterval = 60
}
$Notification = New-FsrmAction @FSRMA
$FSFS = @{
  Path         = $Path
  Notification = $Notification
  IncludeGroup = 'Executable Files'
  Description  = 'Block any executable file'
  Active       = $true
}
Set-FsrmFileScreen @FSFS
```

7. Getting FSRM notification limits

```
Get-FsrmSetting |
  Format-List -Property "*NotificationLimit"
```

8. Changing FSRM notification limits

```
$FSRMSHT = @{
  CommandNotificationLimit = 1
  EmailNotificationLimit   = 1
  EventNotificationLimit   = 1
  ReportNotificationLimit  = 1
}
Set-FsrmSetting @FSRMSHT
```

9. Re-testing the file screen to check the action

```
Copy-Item @FSTHT
```

10. Viewing file screen email

    Use your email client to view the mail created by FSRM.

How it works...

In *step 1*, you examine the initial set of FSRM file groups. The output looks like this:

```
PS C:\Foo> # 1. Examining the existing file groups
PS C:\Foo> Get-FsrmFileGroup |
             Format-Table -Property Name, IncludePattern

Name                 IncludePattern
----                 --------------
Audio and Video Files {*.aac, *.aif, *.aiff, *.asf, *.asx, *.au, *.avi, *.flac, *.m3u, *.mid, *.midi, *.mov, *.mp1,
                      *.mp2, *.mp3, *.mp4, *.mpa, *.mpe, *.mpeg, *.mpeg2, *.mpeg3, *.mpg, *.ogg, *.qt, *.qtw, *.ram,
                      *.rm, *.rmi, *.rmvb, *.snd, *.swf, *.vob, *.wav, *.wax, *.wma, *.wmv, *.wvx}
Image Files          {*.bmp, *.dib, *.eps, *.gif, *.img, *.jfif, *.jpe, *.jpeg, *.jpg, *.pcx, *.png, *.ps, *.psd,
                      *.raw, *.rif, *.spiff, *.tif, *.tiff}
Office Files         {*.accdb, *.accde, *.accdr, *.accdt, *.adn, *.adp, *.doc, *.docm, *.docx, *.dot, *.dotm, *.dotx,
                      *.grv, *.gsa, *.gta, *.mad, *.maf, *.mda, *.mda, *.mda, *.mdb, *.mde, *.mdf, *.mdf, *.mdm, *.mdt,
                      *.mdw, *.mdw, *.mdw, *.mdz, *.mpd, *.mpp, *.mpt, *.obt, *.odb, *.one, *.onepkg, *.pot, *.potm,
                      *.potx, *.ppa, *.ppam, *.pps, *.ppsm, *.ppsx, *.ppt, *.pptm, *.pptx, *.pub, *.pwz, *.rqy, *.rtf,
                      *.rwz, *.sldm, *.sldx, *.slk, *.thmx, *.vdx, *.vsd, *.vsl, *.vss, *.vst, *.vsu, *.vsw, *.vsx,
                      *.vtx, *.wbk, *.wri, *.xla, *.xlam, *.xlb, *.xlc, *.xld, *.xlk, *.xll, *.xlm, *.xls, *.xlsb,
                      *.xlsm, *.xlsx, *.xlt, *.xltm, *.xltx, *.xlv, *.xlw, *.xsf, *.xsn}
E-mail Files         {*.eml, *.idx, *.mbox, *.mbx, *.msg, *.oft, *.ost, *.pab, *.pst}
Executable Files     {*.bat, *.cmd, *.com, *.cpl, *.exe, *.inf, *.js, *.jse, *.msh, *.msi, *.msp, *.ocx, *.pif, *.pl,
                      *.ps1, *.scr, *.vb, *.vbs, *.wsf, *.wsh}
System Files         {*.acm, *.dll, *.ocx, *.sys, *.vxd}
Compressed Files     {*.ace, *.arc, *.arj, *.bhx, *.bz2, *.cab, *.gz, *.gzip, *.hpk, *.hqx, *.jar, *.lha, *.lzh, *.lzx,
                      *.pak, *.pit, *.rar, *.sea, *.sit, *.sqz, *.tgz, *.uu, *.uue, *.z, *.zip, *.zoo}
Web Page Files       {*.asp, *.aspx, *.cgi, *.css, *.dhtml, *.hta, *.htm, *.html, *.mht, *.php, *.php3, *.shtml, *.url}
Text Files           {*.asc, *.text, *.txt}
Backup Files         {*.bak, *.bck, *.bkf, *.old}
Temporary Files      {*.temp, *.tmp, ~*}
```

Figure 10.48: Examining existing FSRM file groups

In *step 2*, you examine the built-in FSRM file screening templates. The output from this step is:

```
PS C:\Foo> # 2. Examining the existing file screening templates
PS C:\Foo> Get-FsrmFileScreenTemplate |
             Format-Table -Property Name, IncludeGroup, Active

Name                                IncludeGroup                          Active
----                                ------------                          ------
Block Audio and Video Files         {Audio and Video Files}                 True
Block Executable Files              {Executable Files}                      True
Block Image Files                   {Image Files}                           True
Block E-mail Files                  {E-mail Files}                          True
Monitor Executable and System Files {Executable Files, System Files}  False
```

Figure 10.49: Viewing file screen templates

In *step 3*, you create a new folder for testing FSRM file screening, which produces no output. In *step 4*, you create a new FSRM file screen. The output from this step looks like this:

```
PS C:\Foo> # 4. Creating a new file screen
PS C:\Foo> $FSHT =  @{
             Path         = $Path
             Description  = 'Block Executable Files'
             IncludeGroup = 'Executable Files'
           }
PS C:\Foo> New-FsrmFileScreen @FSHT

Active          : True
Description     : Block Executable Files
IncludeGroup    : {Executable Files}
MatchesTemplate : False
Notification    :
Path            : C:\FileScreen
Template        :
PSComputerName  :
```

Figure 10.50: Creating a new file screen

To test the file screen, in *step 5*, you copy `notepad.exe` from the Windows folder to the file screen folder, with output like this:

```
PS C:\Foo> # 5. Testing file screen by copying notepad.exe
PS C:\Foo> $FSTHT = @{
             Path        = "$Env:windir\notepad.exe"
             Destination = 'C:\FileScreen\notepad.exe'
           }
PS C:\Foo> Copy-Item  @FSTHT
Copy-Item:
Line |
   6 |  Copy-Item  @FSTHT
     |  ~~~~~~~~~~~~~~~~~
     | Access to the path 'C:\FileScreen\notepad.exe' is denied.
```

Figure 10.51: Testing a file screen

In *step 6*, you set up an active email notification to notify you any time a user attempts to save an executable file to the screened folder. This step creates no output.

In *step 7*, you examine the FSRM notification limits, with output like this:

```
PS C:\Foo> # 7. Getting FSRM notification limits
PS C:\Foo> Get-FsrmSetting |
             Format-List -Property "*NotificationLimit"

CommandNotificationLimit : 60
EmailNotificationLimit   : 60
EventNotificationLimit   : 60
ReportNotificationLimit  : 60
```

Figure 10.52: Examining FSRM notification limits

To speed up the creation of email notifications, in *step 8*, you reduce the email notification limits to 1 second. This step creates no console output.

In *step 9*, you test the updated file screen by re-attempting to save an executable to the screened folder. This results in the following output:

```
PS C:\Foo> # 9. Re-testing the file screen to check the action
PS C:\Foo> Copy-Item @FSTHT
Copy-Item:
Line |
   2 |  Copy-Item @FSTHT
     |  ~~~~~~~~~~~~~~~~
     | Access to the path 'C:\FileScreen\notepad.exe' is denied.
```

Figure 10.53: Re-testing the file screen

Having set up an email notification for the file screen, in *step 10*, you view the generated email, which looks like this:

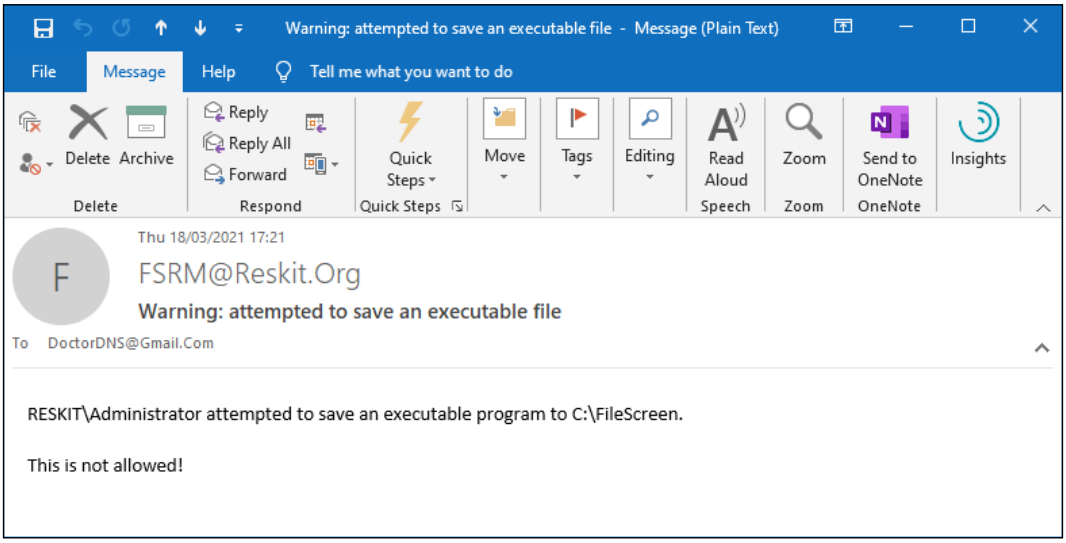


Figure 10.54: Viewing the file screen email

There's more...

In *step 1*, you look at the file extensions that FSRM recognizes by default. These cover most of the common scenarios. A small omission is that the audio and video files should include the extension SHN. SHN files are lossless audio files that use the Shorten compression algorithm. You can find a wealth of legal SHN-based concert recordings of many bands, such as The Grateful Dead. For this reason, in production, you might want to update the FSRM file group to enable FSRM to screen SHN files when you use that FSRM file group in a screening rule.

FSRM's file screening feature does a good job of stopping a user from accidentally saving files that would violate the organization's file storage policies. For example, the organization might stipulate that users may not save audio or video files to the organization's file server. If a user "accidentally" saves an MP3 file, FSRM would politely refuse. However, as noted earlier, FSRM file screening is based solely on the file's extension. Thus, if the user saves a file and changes the extension, say to 3MP, FSRM does not object. Of course, in doing so, the user is deliberately breaking an organizational policy, which could be a career-limiting move.

11

Managing Printing

In this chapter, we cover the following recipes:

- Installing and sharing printers
- Publishing a printer
- Changing the spooler directory
- Changing printer drivers
- Printing a test page
- Managing printer security
- Creating a printer pool

Introduction

Printing is a feature that Microsoft incorporated into various versions of the Windows operating system and has evolved over the years. Windows Server 2022 provides you with a printing platform for your organization. Printer configuration and management in Windows Server 2022 hasn't changed much from earlier versions.

When printing in Windows, the physical device that renders output onto paper is known as a print device. A printer is a queue for a print device. A print server can support multiple printers. Each print device has an associated printer driver that converts your documents to the printed form on a given print device. Some drivers come with Windows, others you need to obtain from the printer vendor. In some cases, these drivers are downloadable from the internet; in other cases, you may need to download and run a driver installation program to add the correct drivers to your print server.

Printers print to the print device using a printer port (such as USB, parallel, or network). For network printers, you need to define the port before you can create a Windows printer. Microsoft hasn't changed the basic print architecture with Windows Server 2022. Windows Server 2012 introduced a new driver architecture that Windows Server 2022 supports. This driver model enables you to use two different driver types: printer class drivers and model-specific drivers. The former provides a single driver for various printing device models. You use model-specific drivers for just a single printer model. Increasingly, print device manufacturers are implementing more generic drivers that can simplify the organizational rollout of shared printers.

Another change in Windows Server 2012, carried into Windows Server 2022, is that you no longer have to use the print server to distribute printer drivers (which is especially relevant for network printers). You can use tools such as the System Center Configuration Manager or Group Policy to distribute print drivers to clients in such cases.

This chapter covers installing, managing, and updating printers, print drivers, and printer ports on a Windows Server 2022 server. You may find that some of the administration tools used in this chapter aren't available on Windows Server Core systems. To enable full management, you need to have the full GUI (including Desktop Experience) for any GUI utilities.

In the *Installing and sharing printers* recipe, you install a printer and share it for others to use. In the *Publishing a printer* recipe, you'll publish the printer to **Active Directory** (**AD**), enabling users in the domain to search AD to find the printer. When you create a print server (by adding printer ports and printers), the default spool folder (Windows uses a folder, by default, that is underneath `C:\Windows`) may not be in an ideal location. In the *Changing the spooler directory* recipe, you change the default location for the printer spool.

Sometimes, a printer can have an associated print device swapped for a different printer model. In the *Changing printer drivers* recipe, you change the driver for the printer you created earlier. A useful troubleshooting step when working with printers is to print a test page, as you can see in the *Printing a test page* recipe.

Printers, like files, can have **Access Control Lists** (**ACLs**) to specify who can use the printer. You can modify the ACL, as shown in the *Managing printer security* recipe. In many organizations, print devices are a shared resource.

In Windows, a **printer pool** is a printer that has two or more associated printing devices. This means having two or more physical printers (print devices on separate ports) that users see as just a single printer. Printer pools are useful in situations where users create large numbers of printed documents, and multiple physical printers can be deployed. In the *Creating a printer pool* recipe, you see how you can automate the creation of a printer pool using `rundll32.exe`.

Installing and sharing printers

The first step in creating a print server for your organization involves installing the print server feature, printer drivers, and printer ports. With those installed, you can create and share a printer for others to access.

In this recipe, you download and install two Xerox printer drivers. You use one of the drivers in this recipe; the other you use in the *Changing printer drivers* recipe. This download comes as a ZIP archive that you need to extract before using the drivers.

Note: if you're using this recipe to support other printer makes and models, you may need to make some changes. In some cases, such as with some Hewlett Packard printers, the manufacturer designed the printer drivers to be installed via a downloadable executable. You would need to run the downloaded executable, which you execute on your print server to add the drivers. Thus, this recipe may not apply to all printing devices.

Getting ready

This recipe uses a Windows Server host `PSRV`, a domain-joined Windows Server 2022 host on which you have installed PowerShell 7 and VS Code.

How to do it...

1. Installing the `Print-Server` feature on `PSRV`

```
Install-WindowsFeature -Name Print-Server, RSAT-Print-Services
```

2. Creating a folder for the Xerox printer drivers

```
$NIHT = @{
  Path        = 'C:\Foo\Xerox'
  ItemType    = 'Directory'
  Force       = $true
  ErrorAction = "Silentlycontinue"
}
New-Item @NIHT | Out-Null
```

3. Downloading printer drivers for Xerox printers

```
$URL='http://download.support.xerox.com/pub/drivers/6510/'+
     'drivers/win10x64/ar/6510_5.617.7.0_PCL6_x64.zip'
$Target='C:\Foo\Xerox\Xdrivers.zip'
Start-BitsTransfer -Source $URL -Destination $Target
```

4. Expanding the ZIP file

```
$Drivers = 'C:\Foo\Xerox\Drivers'
Expand-Archive -Path $Target -DestinationPath $Drivers
```

5. Installing the drivers

```
$M1 = 'Xerox Phaser 6510 PCL6'
$P =  'C:\Foo\Xerox\Drivers\6510_5.617.7.0_PCL6_x64_Driver.inf\'+
      'x3NSURX.inf'
rundll32.exe printui.dll,PrintUIEntry /ia /m "$M1"  /f "$P"
$M2 = 'Xerox WorkCentre 6515 PCL6'
rundll32.exe printui.dll,PrintUIEntry /ia /m "$M2"  /f "$P"
```

6. Adding a `PrinterPort` for a new printer

```
$PPHT = @{
  Name               = 'SalesPP'
  PrinterHostAddress = '10.10.10.61'
}
Add-PrinterPort @PPHT
```

7. Adding the printer to `PSRV`

```
$PRHT = @{
  Name = 'SalesPrinter1'
  DriverName = $m1
  PortName   = 'SalesPP'
}
Add-Printer @PRHT
```

8. Sharing the printer

```
Set-Printer -Name SalesPrinter1 -Shared $True
```

9. Reviewing what you have done

```
Get-PrinterPort -Name SalesPP |
    Format-Table -Autosize -Property Name, Description,
                           PrinterHostAddress, PortNumber
Get-PrinterDriver -Name xerox* |
    Format-Table -Property Name, Manufacturer,
                           DriverVersion, PrinterEnvironment
Get-Printer -ComputerName PSRV -Name SalesPrinter1 |
    Format-Table -Property Name, ComputerName,
                           Type, PortName, Location, Shared
```

How it works...

In *step 1*, you install both the `Print-Server` Windows feature and the printing **Remote Server Administration Tools** (**RSAT**), with output like this:

```
PS C:\Foo> # 1. Installing the Print-Server feature on PSRV
PS C:\Foo> Install-WindowsFeature -Name Print-Server, RSAT-Print-Services

Success Restart Needed Exit Code      Feature Result
------- -------------- ---------      --------------
True    No             Success        {Print Server, Print and Document Services, …
```

Figure 11.1: Installing the Print-Server feature on PSRV

In *step 2*, you create a new folder that you use to hold the printer driver download. In *step 3*, you download the drivers (as a compressed ZIP file) into the folder you just created. In *step 4*, you expand the ZIP files, and in *step 5*, you install the printer drivers. In *step 6*, you add a new printer port to `PSRV`, and in *step 7,* you add a new printer using the printer port you just created and making use of the Xerox printer drivers you have downloaded. Finally, in *step 8*, you share the printer so other users can print to it. These seven steps produce no output.

In *step 5*, you use the `rundll32.exe` console application to install the printer drivers in Windows. This command can, on occasion, generate "Operation could not be completed" error dialogs. The resolution is to wait a few seconds, then try again.

In *step 9*, you examine the printer ports, printer drivers, and printers available on `PSRV`, with output like this:

```
PS C:\Foo> # 9. Reviewing what you have done
PS C:\Foo> Get-PrinterPort -Name SalesPP |
             Format-Table -Autosize -Property Name, Description,
                                            PrinterHostAddress, PortNumber

Name    Description          PrinterHostAddress PortNumber
----    -----------          ------------------ ----------
SalesPP Standard TCP/IP Port 10.10.10.61              9100

PS C:\Foo> Get-PrinterDriver -Name xerox* |
             Format-Table -Property Name, Manufacturer,
                                    DriverVersion, PrinterEnvironment

Name                       Manufacturer       DriverVersion PrinterEnvironment
----                       ------------       ------------- -------------------
Xerox WorkCentre 6515 PCL6 Xerox        1581047950660861952 Windows x64
Xerox Phaser 6510 PCL6     Xerox        1581047950660861952 Windows x64

PS C:\Foo> Get-Printer -ComputerName PSRV -Name SalesPrinter1 |
             Format-Table -Property Name, ComputerName,
                                    Type, PortName, Location, Shared

Name          ComputerName  Type PortName Location Shared
----          ------------  ---- -------- -------- ------
SalesPrinter1 PSRV          Local SalesPP             True
```

Figure 11.2: Viewing printer components on PSRV

There's more...

In this recipe, you create a printer on `PSRV` based on Xerox printers. The fact that you may not have this model of printer in your environment means you can't physically print to such a print device, but you can set up the printer as shown in the recipe.

In *step 5*, you use `printui.dll` and `rundll32.exe`. Now, `printui.dll` is a library of printer management functionalities. If you use the Printer Management GUI tool to manage printers, the GUI calls this DLL to perform your chosen action. Since Microsoft designed and built this DLL to support the Windows printer GUI, the DLL can create additional dialog boxes, which are not useful for automation. You can rely on the help information generated to resolve any problems.

In practice, `printui.dll` is a little flakey and can generate error messages with little supporting detail to help you. The solution is to retry the operation after waiting some seconds, or if the issue continues, reboot the server.

You can get helpful information on the syntax needed by `printui.dll` by opening a PowerShell console window and running the following command:

```
rundll32 printui.dll PrintUIEntry
```

In this recipe, you downloaded and installed two drivers. You use one driver in this recipe to create the `SalesPrinter1` printer. You use the second driver in the *Changing printer drivers* recipe later in this chapter.

Publishing a printer

After you create and share a printer (as shown in the previous recipe), you can also publish it to Active Directory. When you publish a printer, you can also specify a physical location for the printer. Your users can then search for published printers based on location, as well as on capabilities (such as color printing). In this recipe, you publish the printer you created in the previous recipe and examine the results.

Getting ready

Before running this recipe, you need to have the `PSRV` printer server set up (you did this in the *Installing and sharing printers* recipe). Additionally, you need `SalesPrinter1` created.

How to do it...

1. Getting the printer to publish and store the returned object in `$Printer`

```
$Printer = Get-Printer -Name SalesPrinter1
```

2. Viewing the printer details

```
$Printer | Format-Table -Property Name, Published
```

3. Publishing and sharing the printer to AD

```
$Printer | Set-Printer -Location '10th floor 10E4'
$Printer | Set-Printer -Shared $true -Published $true
```

4. Viewing the updated publication status

```
Get-Printer -Name SalesPrinter1 |
  Format-Table -Property Name, Location, DriverName, Published
```

How it works...

In *step 1*, you obtain details of the printer you wish to share, and you store this into the variable `$Printer`, producing no output. In *step 2*, you examine the printer details contained in `$Printer`, with output like this:

```
PS C:\Foo> # 2. Viewing the printer details
PS C:\Foo> $Printer | Format-Table -Property Name, Published

Name          Published
----          ---------
SalesPrinter1     False
```

Figure 11.3: Viewing the SalesPrinter1 printer on PSRV

In *step 3*, you publish the printer to the Reskit.Org AD, and you share the printer explicitly. This step produces no output. In *step 4*, you use `Get-Printer` to review the publication status of the printer, with output like this:

```
PS C:\Foo> # 4. Viewing the updated publication status
PS C:\Foo> Get-Printer -Name SalesPrinter1 |
             Format-Table -Property Name, Location, Drivername, Published

Name          Location        Drivername            Published
----          --------        ----------            ---------
SalesPrinter1 10th floor 10E4 Xerox Phaser 6510 PCL6      True
```

Figure 11.4: Viewing the SalesPrinter1 printer publication status

There's more...

Publishing a printer to AD allows users to locate printers near them using the **Add Printer** dialog to search for published printers. For example, if you log onto any computer in the domain, you can get to this dialog box by clicking **Start | Settings | Devices | Printers & scanners** to bring up the **Add printers & scanners** dialog. From this dialog box, click **Add a printer or scanner**. Wait until the search is complete, then click on **The printer that I want isn't listed**, which brings up the **Add Printer** dialog, like this:

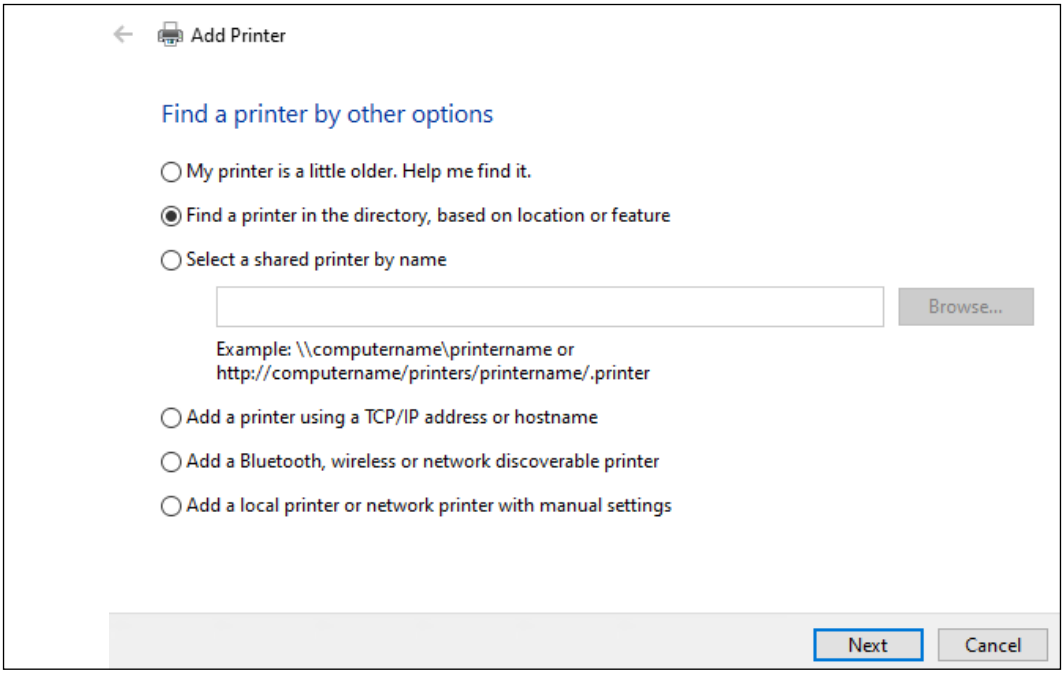


Figure 11.5: Using the Add Printer dialog

From this dialog box, click on **Next** to bring up the **Find Printers** dialog, which looks like this:

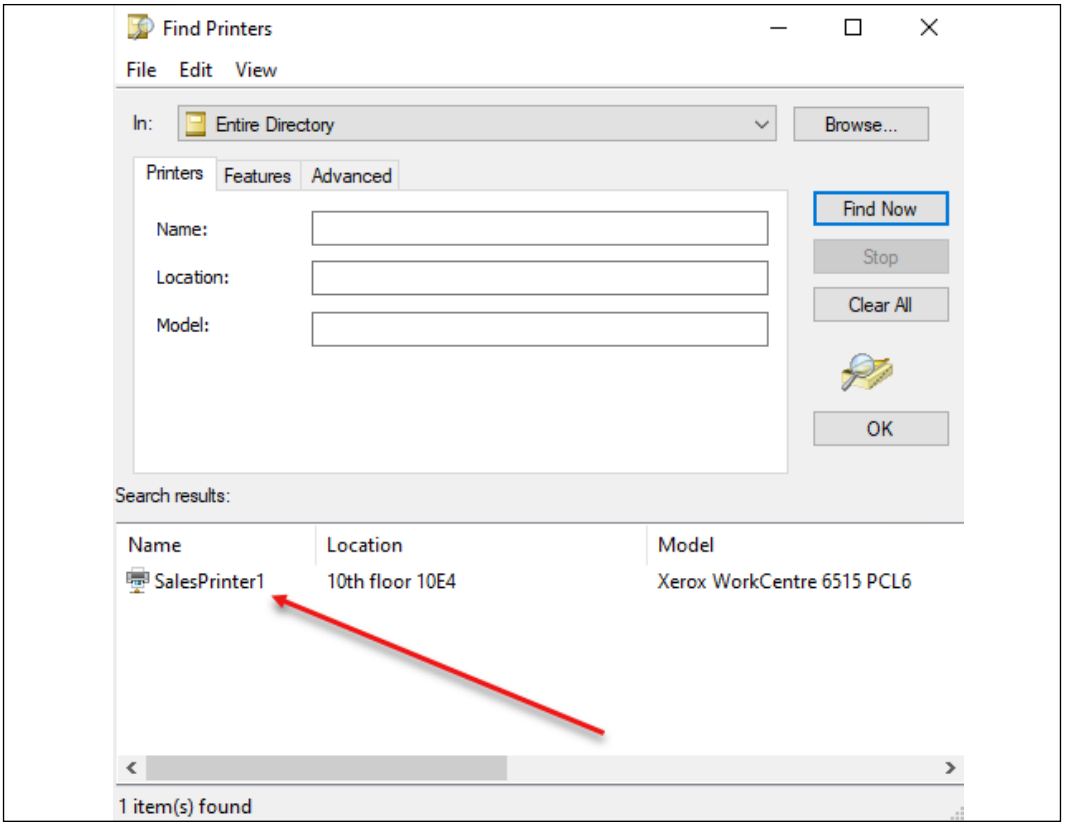


Figure 11.6: Using the Find Printers dialog

In larger organizations, publishing printers to AD can be very useful in helping users to find the corporate printers available to them.

Changing the spooler directory

During the printing process, the Windows printer spooler in Windows uses an on-disk folder to hold the temporary files the printing process creates. If multiple users each print very large documents to a single printer, the print queue and the temporary folder can get quite large. By default, this folder is `C:\Windows\System32\spool\PRINTERS`. For a busy print server with multiple printers, you may wish to change the default spool folder.

Getting ready

This recipe uses the PSRV printer server set up as per the *Installing and sharing printers* recipe.

How to do it...

1. Loading the `System.Printing` namespace and classes

```
Add-Type -AssemblyName System.Printing
```

2. Defining the required `permissions`

```
$Permissions =
   [System.Printing.PrintSystemDesiredAccess]::AdministrateServer
```

3. Creating a `PrintServer` object with the required permissions

```
$NOHT = @{
  TypeName     = 'System.Printing.PrintServer'
  ArgumentList = $Permissions
}
$PS = New-Object @NOHT
```

4. Observing the default spool folder

```
"The default spool folder is: [{0}]" -f $PS.DefaultSpoolDirectory
```

5. Creating a new spool folder

```
$NIHT = @{
  Path        = 'C:\SpoolPath'
  ItemType    = 'Directory'
  Force       = $true
  ErrorAction = 'SilentlyContinue'
}
New-Item @NIHT | Out-Null
```

6. Updating the default spool folder path

```
$Newpath = 'C:\SpoolPath'
$PS.DefaultSpoolDirectory = $Newpath
```

7. Committing the change

```
$Ps.Commit()
```

8. Restarting the `Spooler` to accept the new folder

```
Restart-Service -Name Spooler
```

9. Verifying the new spooler folder

```
New-Object -TypeName System.Printing.PrintServer |
  Format-Table -Property Name,
                DefaultSpoolDirectory
```

10. Stopping the `Spooler` service

```
Stop-Service -Name Spooler
```

11. Creating a new spool directory

```
$SPL = 'C:\SpoolViaRegistry'
$NIHT2 = @{
  Path        = $SPL
  Itemtype    = 'Directory'
  ErrorAction = 'SilentlyContinue'
}
New-Item  @NIHT2 | Out-Null
```

12. Creating the spooler folder and configuring it in the registry

```
$RPath = 'HKLM:\SYSTEM\CurrentControlSet\Control\' +
         'Print\Printers'
$IP = @{
  Path    = $RPath
  Name    = 'DefaultSpoolDirectory'
  Value   = $SPL
}
Set-ItemProperty @IP
```

13. Restarting the `Spooler`

```
Start-Service -Name Spooler
```

14. Viewing the newly updated spool folder

```
New-Object -TypeName System.Printing.PrintServer |
  Format-Table -Property Name, DefaultSpoolDirectory
```

How it works...

In *step 1*, you load the `System.Printing` namespace, which produces no output. In *step 2*, you create a variable holding the desired access to the printer. In *step 3*, you create a `PrintServer` object with the appropriate permissions. Then, in *step 4*, you examine the default spool folder for a newly installed Windows Server 2022 host, which produces output like this:

```
PS C:\Foo> # 4. Observing the default spool folder
PS C:\Foo> "The default spool folder is: [{0}]" -f $PS.DefaultSpoolDirectory
The default spool folder is: [C:\Windows\system32\spool\PRINTERS]
```

Figure 11.7: Examining the default spool folder

In *step 5*, you create a new folder on `PSRV` to serve as your print server's spool folder. In *step 6*, you update the printer spool folder path. In *step 7*, you commit this change, and in *step 8*, you restart the spooler service. These four steps produce no console output.

In *step 9*, you re-view the printer details with output like this:

```
PS C:\Foo> # 9. Verifying the new spooler folder
PS C:\Foo> New-Object -TypeName System.Printing.PrintServer |
             Format-Table -Property Name,
                                    DefaultSpoolDirectory

Name    DefaultSpoolDirectory
----    ---------------------
\\PSRV C:\SpoolPath
```

Figure 11.8: Verifying the new spooler folder

In *step 1* through *step 9*, you set and validate the spool folder using a .NET object.

In *step 10*, you stop the spooler service. In *step 11*, you create a new folder, and in *step 12*, you configure the necessary registry settings to contain the path to the new spool folder. Then in *step 13*, you restart the spooler. These four steps produce no console output. With the changes made to the spool folder within the registry, in *step 14*, you can view the updated spool folder with output like this:

```
PS C:\Foo> # 14. Viewing the newly updated spool folder
PS C:\Foo> New-Object -TypeName System.Printing.PrintServer |
             Format-Table -Property Name, DefaultSpoolDirectory

Name    DefaultSpoolDirectory
----    ---------------------
\\PSRV C:\SpoolViaRegistry
```

Figure 11.9: Examining the updated spool folder

There's more...

For most organizations with a larger number of shared printers, configuring the print server to use another folder for spooling is a good idea. If possible, use a separate disk drive if you can to avoid the risk of the spool folder filling up.

In this recipe, you used two different mechanisms to change the spooler folder. One uses a .NET object (which isn't loaded by default), while the other involves directly editing the registry. Needless to say, if you're rolling out printers using scripts, particularly ones that edit the registry, careful testing is vital.

Many of the steps in this recipe produce no output. This lack of output is normal when dealing directly with .NET classes and methods and editing the registry.

Changing printer drivers

On occasion, it may be necessary to change the printer driver for a printer. For example, you might be replacing an existing print device with a new or different model. In this case, you want the printer name to remain the same, but you need to update the printer's print driver. In the *Installing and sharing printers* recipe, you downloaded and installed two Xerox printer drivers. You used the first driver, `Xerox Phaser 6510 PCL6`, when you defined the `SalesPrinter1` printer.

In this recipe, you change the driver for the printer and use the other previously installed driver, `Xerox Phaser 6515 PCL6`.

This recipe assumes that the printer name and printer port (including the printer's IP address and port number) do not change, only the driver.

Getting ready

Run this recipe on the `PSRV` host, set up as per the *Installing and sharing printers* recipe.

How to do it...

1. Adding the print driver for the new printing device

```
$M2 = 'Xerox WorkCentre 6515 PCL6'
Add-PrinterDriver -Name $M2
```

2. Getting the Sales group printer object and storing it in `$Printer`

```
$Printern = 'SalesPrinter1'
$Printer  = Get-Printer -Name $Printern
```

3. Updating the driver using the `Set-Printer` cmdlet

```
$Printer | Set-Printer -DriverName $M2
```

4. Observing the updated printer driver

```
Get-Printer -Name $Printern |
  Format-Table -Property Name, DriverName, PortName,
                Published, Shared
```

How it works...

In *step 1*, you use the `Add-PrinterDriver` cmdlet to add the printer driver for the printer. In *step 2*, you obtain the printer details, and in *step 3*, you update the printer to use the updated driver. These steps produce no console output.

In *step 4*, you use `Get-Printer` to observe that you have installed the updated driver for the `SalesPrinter1` printer, which looks like this:

```
PS C:\Foo> # 4. Observing the updated printer driver
PS C:\Foo> Get-Printer -Name $Printern |
             Format-Table -Property Name, DriverName, PortName,
                                    Published, Shared

Name          DriverName                PortName Published Shared
----          ----------                -------- --------- ------
SalesPrinter1 Xerox WorkCentre 6515 PCL6 SalesPP       True   True
```

Figure 11.10: Viewing the updated printer driver for SalesPrinter1

There's more...

As you see in this recipe, changing a printer driver is straightforward – after you install a new printer driver, you use `Set-Printer` to inform Windows which driver it should use when printing to the printer.

Printing a test page

There are occasions when you may wish to print a test page on a printer; for example, after changing the toner or printer ink on a physical printer or after changing the print driver (as shown in the *Changing printer drivers* recipe). In those cases, the test page helps you to ensure that the printer is working properly.

Getting ready

This recipe uses the PSRV print server that you set up in the *Installing and sharing printers* recipe.

How to do it...

1. Getting the printer objects from WMI

```
$Printers = Get-CimInstance -ClassName Win32_Printer
```

2. Displaying the number of printers defined on PSRV

```
'{0} Printers defined on this system' -f $Printers.Count
```

3. Getting the Sales group printer WMI object

```
$Printer = $Printers |
  Where-Object Name -eq 'SalesPrinter1'
```

4. Displaying the printer's details

```
$Printer | Format-Table -AutoSize
```

5. Printing a test page

```
Invoke-CimMethod -InputObject $Printer -MethodName PrintTestPage
```

6. Checking on print job

```
Get-PrintJob -PrinterName SalesPrinter1
```

How it works...

In *step 1*, you get details of the printers installed on PSRV using WMI and store them in the variable `$Printers`, producing no output. In *step 2*, you display how many printers you have defined on your printer server, with output like this:

```
PS C:\Foo> # 2. Displaying the number of printers defined on PSRV
PS C:\Foo> '{0} Printers defined on this system' -f $Printers.Count
7 Printers defined on this system
```

Figure 11.11: Viewing the number of printers on PSRV

In *step 3*, you get the specific WMI instance for the `SalesPrinter1` printer, creating no output. In *step 4*, you view the details of this printer, with output like this:

```
PS C:\Foo> # 4. Displaying the printer's details
PS C:\Foo> $Printer | Format-Table -AutoSize

Name          ShareName     SystemName PrinterState PrinterStatus Location
----          ---------     ---------- ------------ ------------- --------
SalesPrinter1 SalesPrinter1 PSRV       0            3             10th floor 10E4
```

Figure 11.12: Viewing printers on PSRV

In *step 5*, you use `Invoke-CimMethod` to run the `PrintTestPage` method on the printer. This step generates a simple printer test page, with console output like this:

```
PS C:\Foo> # 5. Printing a test page
PS C:\Foo> Invoke-CimMethod -InputObject $Printer -MethodName PrintTestPage


ReturnValue PSComputerName
----------- --------------
          0
```

Figure 11.13: Printing a test page

In the final step in this recipe, you view the print jobs on the `SalesPrinter1` printer, where you can see your test page output generated in the previous step. The output from this step looks like this:

```
PS C:\Foo> # 6. Checking on print job
PS C:\Foo> Get-PrintJob -PrinterName SalesPrinter1


Id    ComputerName    PrinterName    DocumentName    SubmittedTime        JobStatus
--    ------------    -----------    ------------    -------------        ---------
1                     SalesPrinter1  Test Page       20/03/2021 15:44:12  Normal
```

Figure 11.14: Viewing printer jobs to observe the test page

There's more...

In this recipe, you used WMI to create a test page on the `SalesPrinter1` printer. As you saw in *step 6,* the printer's queue has the print job. In theory, the document should appear on the print device pretty much straightaway. If you do not get any physical output, you need to carry out some routine printer troubleshooting: is the printer turned on, and is the network cable plugged in and working?

Managing printer security

Every Windows printer has a discretionary **access control list** (**ACL**). The ACL contains one or more **access control entries** (**ACEs**). Each ACE defines a specific permission for some specific group or user. You could define a group (such as `SalesAdmins`) and give that group the permission to manage documents, while giving another group (such as `Sales`) access to print to the printer.

By default, when you create a printer, Windows adds some ACEs to the printer's ACL. This includes giving the `Everyone` group the permission to print to the printer. For some printers, this may not be appropriate. For this reason, you may need to adjust the ACL, as shown in this recipe.

The `PrintManagement` module contains several cmdlets that help you manage the printers. However, there are no cmdlets for managing ACLs on printers. You can always use .NET directly to manage the ACL or use third-party scripts that do the job for you, but the code for this can be complex (and easy to mess up). Make sure you test any recipes that modify printer ACLs very carefully. Always have a way to reset any ACL back to defaults should you make a mistake and need to start again to define the printer ACL. And as ever, you can always manage the ACL using the GUI if you need to!

Getting ready

Run this recipe on the `PSRV` printer server VM after you have installed and configured the `SalesPrinter1` printer. This recipe uses the `SalesPrinter1` printer and creates two new groups, `Sales` and `SalesAdmin`, in AD.

How to do it...

1. Setting up AD for this recipe

```
$SB = {
 # 1.1 Creating Sales OU
 $OUHT = @{
   Name = 'Sales'
   Path = 'DC=Reskit,DC=Org'
 }
 New-ADOrganizationalUnit @OUHT
 # 1.2 Creating Sales Group
 $G1HT = @{
   Name       = 'SalesGroup'
   GroupScope = 'Universal'
   Path       = 'OU=Sales,DC=Reskit,DC=Org'
 }
```

```
 New-ADGroup @G1HT
 # 1.3 Creating SalesAdmin Group
  $G2HT = @{
    Name       = 'SalesAdmins'
    GroupScope = 'Universal'
    Path       = 'OU=Sales,DC=Reskit,DC=Org'
  }
  New-ADGroup @G2HT
}
# 1.4 Running Script block on DC1
Invoke-Command -ComputerName DC1 -ScriptBlock $SB
```

2. Creating an `NTAccount` object

```
$GHT1 = @{
    Typename     = 'Security.Principal.NTAccount'
    Argumentlist = 'SalesGroup'
}
$SalesGroup = New-Object @GHT1
$GHT2 = @{
    Typename     = 'Security.Principal.NTAccount'
    Argumentlist = 'SalesAdmins'
}
$SalesAdminGroup = New-Object @GHT2
```

3. Getting the group SIDs

```
$SalesGroupSid =
   $SalesGroup.Translate([Security.Principal.Securityidentifier]).Value
 $SalesAdminGroupSid =
   $SalesAdminGroup.Translate(
     [Security.Principal.Securityidentifier]).Value
```

4. Defining the SDDL for this printer

```
$SDDL = 'O:BAG:DUD:PAI(A;OICI;FA;;;DA)' +
        "(A;OICI;0x3D8F8;;;$SalesGroupSid)" +
        "(A;;LCSWSDRCWDWO;;;$SalesAdminGroupSid)"
```

5. Getting the Sales group's printer object

```
$SGPrinter = Get-Printer -Name SalesPrinter1 -Full
```

6. Setting the permissions

```
$SGPrinter | Set-Printer -Permission $SDDL
```

7. Viewing the permissions from the GUI

   Open the Windows Settings applet, click on **Devices**, and then click on **Printers & scanners**. Next, click on the **SalesPrinter1** printer, then click on **Manage**. Finally, click on **Printer properties**.

How it works...

In *step 1*, you create a new OU in the Reskit.Org domain (`Sales`), then create two new Universal security groups (`SalesGroup` and `SalesAdmins`).

In *step 2*, you create two `Security.Principal.NTAccount` objects with the properties of the two security groups. In *step 3*, you use these two objects to retrieve the **Security IDs** (**SIDs**) for each of the groups. In *step 4*, you create a **Security Descriptor Description Language** (**SDDL**) permission set.

In *step 5*, you use `Get-Printer` to return a WMI object that describes the printer. Then in *step 6*, you use `Set-Printer` to set the ACL for your printer. These first six steps produce no console output.

In *step 7*, you observe the ACL by using the Windows Printer GUI. The output looks like this:

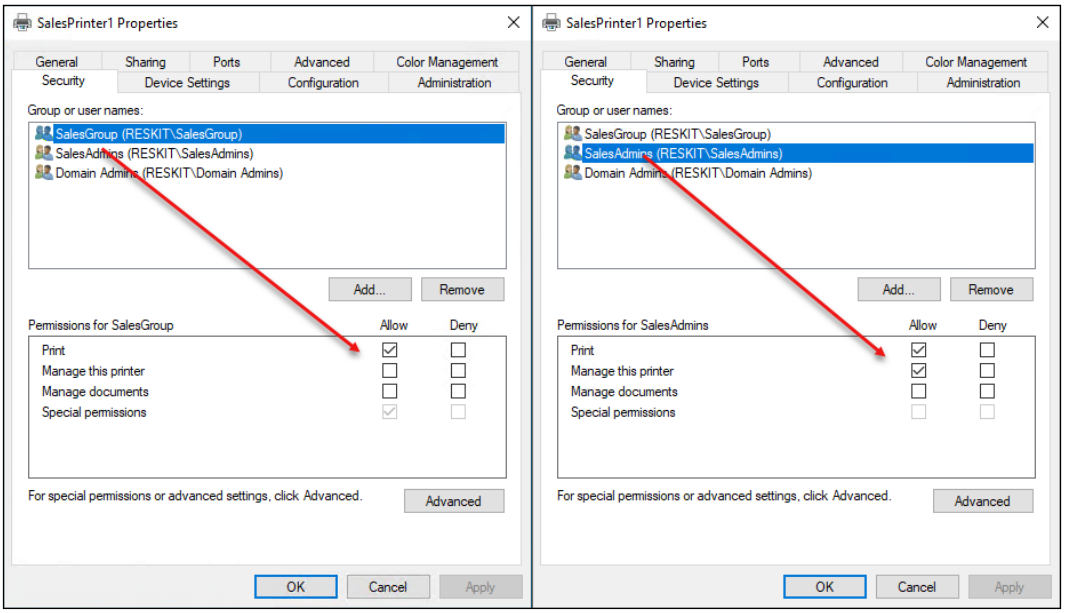


Figure 11.15: Viewing printer security

There's more...

In this recipe, you create an ACL for the Sales group printer `SalesPrinter1`. The recipe makes use of a .NET object to obtain the SIDs for two security groups. Then you hand-construct the SDDL and apply it to the printer.

Unlike NTFS, there are no third-party printer ACL management tools readily available to simplify the setting of ACLs. SDDL is the default mechanism, but it is not always all that straightforward. For some details on SDDL, see `https://itconnect.uw.edu/wares/msinf/other-help/understanding-sddl-syntax/`.

Creating a printer pool

Windows allows you to create a **printer pool**, a printer with two or more print devices (each with a separate printer port). With a printer pool, Windows sends a given print job to any of the pool's printers. This feature is helpful in environments where users print large numbers of documents and need the speed that additional printers can provide without asking the user to choose the specific print device to use.

There are no PowerShell cmdlets to enable you to create a printer pool. Also, WMI does not provide a mechanism to create a printer pool. As with other recipes in this chapter, you use `printui.dll` and `rundll32` to deploy your printer pool. This recipe is another example of utilizing older console applications to achieve your objective.

Getting ready

You run this recipe on `PSRV`, on which you have set up a new printer, `SalesPrinter1`.

How to do it...

1. Adding a port for the printer

```
$P = 'SalesPP2'       # new printer port name
Add-PrinterPort -Name $P -PrinterHostAddress 10.10.10.62
```

2. Creating the printer pool for `SalesPrinter1`

```
$Printer = 'SalesPrinter1'
$P1      = 'SalesPP'   # First printer port
$P2      = 'SalesPP2'  # Second printer port
rundll32.exe printui.dll,PrintUIEntry /Xs /n $Printer Portname "$P1,$P2"
```

3. Viewing the printer pool

```
Get-Printer $Printer |
   Format-Table -Property Name, Type, DriverName, PortName
```

How it works...

In *step 1*, you add a new printer port, `SalesPP2`. In *step 2,* you create a new printer pool by using `printui.dll` and setting two printer ports. In the final step, *step 3*, you view the output to confirm you have set up a printer pool of two print devices/ports. The output of this final step looks like this:

```
PS C:\Foo> # 3. Viewing the printer pool
PS C:\Foo> Get-Printer $Printer |
             Format-Table -Property Name, Type, DriverName, PortName -AutoSize

Name           Type DriverName                 PortName
----           ---- ----------                 --------
SalesPrinter1 Local Xerox WorkCentre 6515 PCL6 SalesPP2,SalesPP
```

Figure 11.16: Viewing the printer pool

There's more...

In *step 3*, you use the `Get-Printer` command to retrieve details about the printer to verify you have set up a printer pool. You can also view this pool using the printer GUI, which looks like this:

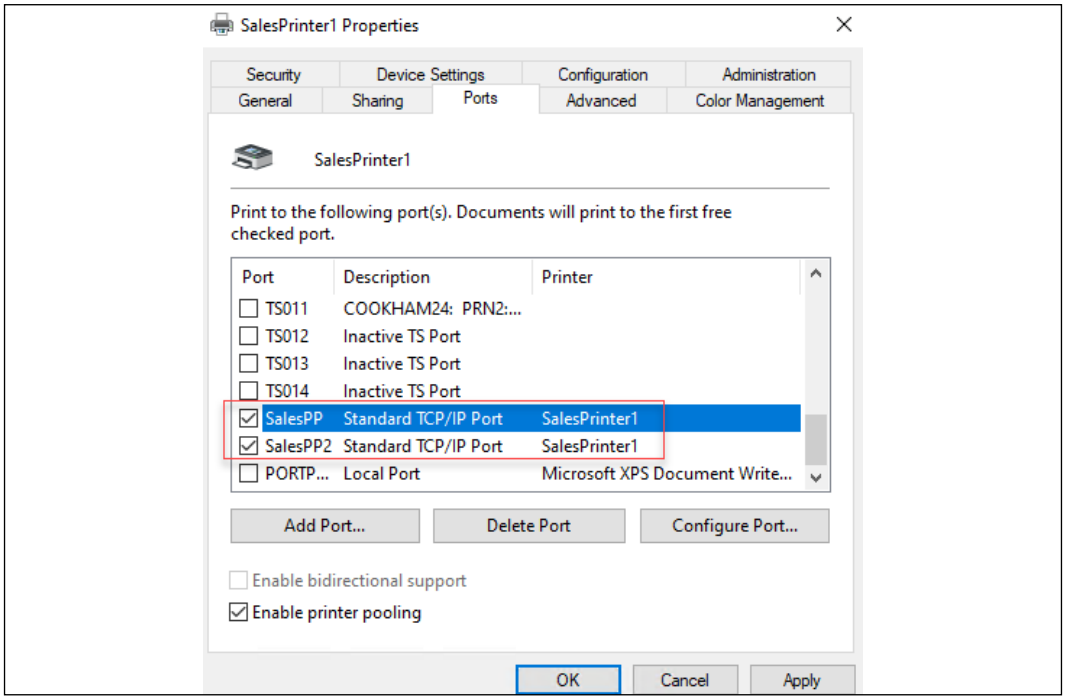


Figure 11.17: Viewing the printer pool from the GUI

12
Managing Hyper-V

In this chapter, we cover the following recipes:

- Installing Hyper-V inside Windows Server
- Creating a Hyper-V VM
- Using PowerShell Direct
- Using Hyper-V VM groups
- Configuring VM hardware
- Configuring VM networking
- Implementing nested virtualization
- Managing VM state
- Managing VM and storage movement
- Managing VM replication
- Managing VM checkpoints

Introduction

Hyper-V is Microsoft's **virtual machine** (**VM**) hypervisor. Both Windows Server 2022 and Windows 10 include Hyper-V as an option you can install, on all versions of Windows Server 2022, and the Enterprise, Professional, and Education editions of Windows 10.

Microsoft first released Hyper-V with Server 2008 and has improved it significantly with each successive version of Windows Server. Improvements include additional features, support for the latest hardware, and significantly increased scalability.

Hyper-V supports nested virtualization, the ability to run Hyper-V inside a Hyper-V VM. Nested virtualization has some good use cases, such as in training – for instance, giving each student a VM on a large blade containing the VMs needed for the course labs. Nested virtualization provides an additional layer of security that might be useful in multi-tenant scenarios.

Microsoft also ships a free version of Hyper-V, the Microsoft Hyper-V Server. The Hyper-V Server runs VMs with no GUI. You configure and manage remotely using recipes like the ones in this chapter.

This chapter focuses solely on Hyper-V inside Windows Server 2022, although you can manage Hyper-V Server using the tools used in this chapter's recipes. References to your Hyper-V servers refer to your Windows 2022 servers that have the Hyper-V feature added. Hyper-V's management tools enable you to configure and manage both the Hyper-V service and the VMs running on your Hyper-V servers.

This chapter starts with installing and configuring the Hyper-V feature. After installing Hyper-V, you go on to create a VM, `PSDirect`, which requires you to download an ISO image of Windows Server from the internet.

After you create the `PSDirect` VM, you use the VM. You use PowerShell Direct to use a remoting session into the VM without using a network connection. You also configure the VM's hardware and networking capability. You then use PowerShell cmdlets to manage the state of the VM.

Hyper-V allows you to move a VM and/or a VM's storage between Hyper-V hosts. For disaster recovery, you can replicate a running VM (and use that replica should the primary VM fail).

You can take snapshots, or checkpoints, of a VM to save a VM's state and restore your VM to that point, as you can see in the *Managing VM checkpoints* recipe.

Installing Hyper-V inside Windows Server

Installing Hyper-V on Windows Server 2022 is straightforward, as this recipe demonstrates. Once you have installed it, you can configure the feature.

Getting ready

This recipe uses `SRV2`, a recently added workgroup host. By default, this host is a DHCP client. You also need two Windows Server instances, `HV1` and `HV2`, online and must have at least one domain controller in the domain up and running. This recipe demonstrates the remote installation and configuration of your Hyper-V servers.

How to do it...

1. Installing the Hyper-V feature on HV1, HV2

```
$SB = {
  Install-WindowsFeature -Name Hyper-V -IncludeManagementTools
}
Invoke-Command -ComputerName HV1, HV2 -ScriptBlock $Sb
```

2. Rebooting the servers to complete the installation

```
Restart-Computer -ComputerName HV2 -Force
Restart-Computer -ComputerName HV1 -Force
```

3. Creating a PSSession with both HV servers (after reboot)

```
$S = New-PSSession HV1, HV2
```

4. Creating and setting the location for VMs and VHDs on HV1 and HV2

```
$SB = {
    New-Item -Path C:\VM -ItemType Directory -Force |
        Out-Null
    New-Item -Path C:\VM\VHDs -ItemType Directory -Force |
        Out-Null
    New-Item -Path C:\VM\VMs -ItemType Directory -force |
        Out-Null}
Invoke-Command -ScriptBlock $SB -Session $S | Out-Null
```

5. Setting the default paths for Hyper-V VM disk/config information

```
$SB = {
  $VMs  = 'C:\VM\VMs'
  $VHDs = 'C:\VM\VHDs'
  Set-VMHost -ComputerName Localhost -VirtualHardDiskPath $VHDs
  Set-VMHost -ComputerName Localhost -VirtualMachinePath $VHDs
}
Invoke-Command -ScriptBlock $SB -Session $S
```

6. Setting NUMA spanning

```
$SB = {
  Set-VMHost -NumaSpanningEnabled $true
}
Invoke-Command -ScriptBlock $SB -Session $S
```

7. Setting EnhancedSessionMode

```
$SB = {
 Set-VMHost -EnableEnhancedSessionMode $true
}
Invoke-Command -ScriptBlock $SB -Session $S
```

8. Setting host resource metering on HV1, HV2

```
$SB = {
 $RMInterval = New-TimeSpan -Hours 0 -Minutes 15
 Set-VMHost -ResourceMeteringSaveInterval $RMInterval
}
Invoke-Command -ScriptBlock $SB -Session $S
```

9. Reviewing key VM host settings

```
$SB = {
  Get-VMHost
}
$P = 'Name', 'V*Path','Numasp*', 'Ena*','RES*'
Invoke-Command -Scriptblock $SB -Session $S |
  Format-Table -Property $P
```

How it works...

In *step 1*, you install the Hyper-V feature to both HV1 and HV2. The output from this step looks like this:

```
PS C:\Foo> # 1. Installing the Hyper-V feature on HV1, HV2
PS C:\Foo> $SB = {
             Install-WindowsFeature -Name Hyper-V -IncludeManagementTools
           }
PS C:\Foo> Invoke-Command -ComputerName HV1, HV2 -ScriptBlock $SB


Success Restart Needed Exit Code      Feature Result                              PSComputerName
------- -------------- ---------      --------------                              --------------
True    Yes            SuccessRestar… {Hyper-V, Hyper-V Module for Windows PowerSh… HV2
WARNING: You must restart this server to finish the installation process.
True    Yes            SuccessRestar… {Hyper-V, Hyper-V Module for Windows PowerSh… HV1
WARNING: You must restart this server to finish the installation process.
```

Figure 12.1: Installing the Hyper-V feature on HV1 and HV2

To complete the installation of Hyper-V, in *step 2,* you reboot both HV1 and HV2. This step creates no output but does reboot both hosts.

After HV1 and HV2 have rebooted, you log back in to the host using `Reskit/Administrator`. In *step 3*, you create two new PowerShell remoting sessions to HV1 and HV2. In *step 4*, you use the remoting sessions and create new folders on HV1 and HV2 to hold Hyper-V VMs and Hyper-V virtual disks. With *step 5*, you configure Hyper-V on both hosts to use these new locations to store VMs and virtual drives, and in *step 6,* you specify that the host should support NUMA spanning. Then, in *step 7*, you set enhanced session mode to improve VM connections. Finally, in *step 8*, you set the two Hyper-V hosts to make use of resource metering. These six steps produce no console output.

In *step 9*, you review the key Hyper-V host settings, with output like this:

```
PS C:\Foo> # 9. Reviewing key VM host settings
PS C:\Foo> $SB = {
             Get-VMHost
           }
PS C:\Foo> $P = 'Name', 'V*Path','Numasp*', 'Ena*','RES*'
PS C:\Foo> Invoke-Command -Scriptblock $SB -Session $S |
             Format-Table -Property $P

Name VirtualHardDiskPath VirtualMachinePath NumaSpanningEnabled EnableEnhancedSessionMode ResourceMeteringSaveInterval
---- ------------------- ------------------ ------------------- ------------------------- ----------------------------
HV1  C:\VM\VHDS          C:\VM\VMS                         True                      True 01:00:00
HV2  C:\VM\VHDS          C:\VM\VMS                         True                      True 01:00:00
```

Figure 12.2: Viewing Hyper-V settings

There's more...

In *step 1*, you install the Hyper-V feature on two servers. You can only do this successfully if the host you are using supports the necessary virtualization capabilities and you have enabled them in your system's BIOS. To ensure that your system is capable of supporting Hyper-V, see this link: `https://mikefrobbins.com/2012/09/06/use-powershell-to-check-for-processor-cpu-second-level-address-translation-slat-support/`. Additionally, ensure you double check that you have enabled virtualization in the VM host's BIOS before running this step.

In *step 2*, you restart both servers. You could have allowed `Install-WindowsFeature` (used in *step 1*) to restart the servers automatically by using the `-Restart` switch. In automation terms, this could have meant that the system started rebooting before the remote script had completed, which could cause `Invoke-Command` to error out. The recipe avoids this by not rebooting after installing the Hyper-V features. You then reboot the host in a controlled way in a later step. Once the restart has completed, your script can carry on managing the servers.

In *step 4* through *step 8,* you set up one aspect of the VM hosts in each step. You could have combined these steps and just called `Set-VMHost` once with all Hyper-V server properties specified.

See also

You can find more information on some of the Hyper-V features used in this recipe (details of which are outside the scope of this book), as follows:

Features	Links for more information
Connecting to a VM, including enhanced session mode	`https://docs.microsoft.com/windows-server/virtualization/hyper-v/learn-more/use-local-resources-on-hyper-v-virtual-machine-with-vmconnect`

Understanding the hard disk options	https://www.altaro.com/hyper-v/understanding-working-vhdx-files/
Hyper-V and NUMA	https://blogs.technet.microsoft.com/pracheta/2014/01/22/numa-understand-it-its-usefulness-with-windows-server-2012/
Configuring Hyper-V resource metering	https://redmondmag.com/articles/2013/08/15/hyper-v-resource-metering.aspx

Creating a Hyper-V VM

Creating a Hyper-V VM is relatively straightforward and consists of a few simple steps. First, you need to create the VM itself inside Hyper-V. Then, you create the VM's virtual hard drive and add it to the VM. You may also wish to adjust the number of processors and memory for the VM and set the VM's DVD drive contents. Once you have created your VM, you need to install the VM's **operating system** (**OS**). You have numerous options regarding how you deploy Windows (or Linux) in a Hyper-V VM.

The Windows Assessment and Deployment Kit, a free product from Microsoft, contains various tools to assist in the automation of deploying Windows. These include **Deployment Image Servicing and Management** (**DISM**), **Windows Imaging and Configuration Designer** (**Windows ICD**), **Windows System Image Manager** (**Windows SIM**), **User State Migration Tool** (**USMT**), and a lot more. For more information on the tools and deploying Windows, see `https://docs.microsoft.com/windows/deployment/windows-deployment-scenarios-and-tools`.

Another way to install the OS onto a VM is to create the VM (either with PowerShell or the Hyper-V Manager) and attach the OS's ISO image to the VM's DVD drive. After starting the VM, you do a manual installation, and once you have completed the OS installation, you can use the recipes in this book to configure the server to your needs.

In this recipe, you create a VM, `PSDirect`. Initially, this has a Windows-assigned hostname that you later change to Wolf. In building the VM, you assign the Windows Server 2022 DVD to the VM's DVD drive. This step ensures that when you start the VM, Windows commences the GUI setup process. The result should be a fully installed OS inside the VM. The details of performing the actual installation are outside the scope of this recipe.

Two minor issues with using the GUI to install Windows Server 2022 are that the Windows setup randomly generates a machine name, and the VM is set up as a workgroup computer and not joined to the domain. You can easily script both renaming the server and joining the domain. The scripts used to generate the VM farm used in this book are examples of deploying Windows Server 2022 in a more automated fashion using a `SETUP.XML` file that specifies the installation's details. The scripts that create the VMs used are available online on GitHub. See `https://github.com/doctordns/ReskitBuildScripts` for the scripts and documentation on them.

Getting ready

You run this recipe on the VM host HV1 that you created in the *Installing Hyper-V inside Windows Server* recipe. You also need the Windows Server ISO image. For testing purposes, this could be an evaluation version or a complete retail edition – and for this chapter, you could use an image of Windows Server 2022 or even Windows Server 2019. To get an evaluation version of Windows Server, visit the Microsoft Evaluation Center at `https://www.microsoft.com/evalcenter/evaluate-windows-server`.

You can also download an ISO of the latest preview copy of Windows Server and use ISO when installing the VM. See `https://www.microsoft.com/en-us/software-download/windowsinsiderpreviewserver` for preview build downloads, and `https://techcommunity.microsoft.com/t5/windows-server-insiders/bd-p/WindowsServerInsiders` for more information about Windows Insiders builds. For this, you need to be a member of the Windows Insiders program – membership is free. For more details on getting started with the Windows Insiders program, see `https://docs.microsoft.com/windows-insider/get-started`.

How to do it...

1. Setting up the VM name and paths for this recipe

```
$VMname      = 'PSDirect'
$VMLocation  = 'C:\VM\VMs'
$VHDlocation = 'C:\VM\VHDs'
$VhdPath     = "$VHDlocation\PSDirect.Vhdx"
$ISOPath     = 'C:\builds\en_windows_server_x64.iso'
If ( -not (Test-Path -Path $ISOPath -PathType Leaf)) {
  Throw "Windows Server ISO DOES NOT EXIST"
}
```

2. Creating a new VM

```
New-VM -Name $VMname -Path $VMLocation -MemoryStartupBytes 1GB
```

3. Creating a virtual disk file for the VM

```
New-VHD -Path $VhdPath -SizeBytes 128GB -Dynamic | Out-Null
```

4. Adding the virtual hard drive to the VM

```
Add-VMHardDiskDrive -VMName $VMname -Path $VhdPath
```

5. Setting the ISO image in the VM's DVD drive

```
$IHT = @{
  VMName           = $VMName
  ControllerNumber = 1
  Path             = $ISOPath
```

```
}
Set-VMDvdDrive @IHT
```

6. Starting the VM

```
Start-VM -VMname $VMname
```

7. Viewing the VM

```
Get-VM -Name $VMname
```

How it works...

In *step 1*, you specify VMs and VM hard drives' locations and assign them to variables. You also check to ensure that the Windows Server ISO is in the correct place. Assuming the ISO exists and you have it named correctly, then this step produces no output. If the step fails to find the file, it aborts.

In *step 2*, you create a new VM using the `New-VM` cmdlet, with output that looks like this:

```
PS C:\Foo> # 2.  Creating a new VM
PS C:\Foo> New-VM -Name $VMname -Path $VMLocation -MemoryStartupBytes 1GB

Name     State CPUUsage(%) MemoryAssigned(M) Uptime   Status             Version
----     ----- ----------- ----------------- ------   ------             -------
PSDirect Off   0           0                 00:00:00 Operating normally 10.0
```

Figure 12.3: Creating a new Hyper-V VM

In *step 3*, you create a virtual disk file for the VM, and you add this VHDX to the `PSDirect` VM in *step 4*. In *step 5,* you add the ISO image to the `PSDirect` VM and then, in *step 6*, you start the VM. These four steps create no output.

In the final step in this recipe, you use `Get-VM` to view the VM details, producing output like this:

```
PS C:\Foo> # 7. Viewing the VM
PS C:\Foo> Get-VM -Name $VMname

Name     State   CPUUsage(%) MemoryAssigned(M) Uptime           Status             Version
----     -----   ----------- ----------------- ------           ------             -------
PSDirect Running 10          1024              00:00:33.0140000 Operating normally 10.0
```

Figure 12.4: Viewing the new Hyper-V VM

There's more...

In *step 1*, you specify the name of the ISO image for Windows Server. Depending on which image you are using, the filename of the ISO you download may be different. Either adjust the filename to match this step or change it to match the ISO file's name. For released versions of Windows Server, you can find ISO images for evaluation versions at the Microsoft Evaluation Center. If you like living in the fast lane, consider using a Windows Insiders preview build.

In *step 2*, you create a VM. Although that step should succeed, you cannot run this VM yet, as you need to define at least one virtual hard drive to use with the VM. You carry out these configuration steps in subsequent steps.

Once you start the VM, in *step 6*, the VM runs and completes Windows Server's installation. To complete the installation of Windows Server, you need to use the GUI. For this chapter's purposes, accept all the installation defaults and ensure that you set the password to `Pa$$w0rd`. As with all the passwords used in this book, feel free to use whatever passwords you wish – just ensure that you don't forget them later.

Using PowerShell Direct

PowerShell Direct is a Hyper-V feature that enables you to open PowerShell remoting sessions on a VM without using a network connection. This feature enables you to create a remoting session inside the VM to fix issues, such as networking misconfiguration. An administrator might be commissioning a new host and configure its host IP address nearly correctly, meaning a network connectivity Catch-22 situation. Without a working network connection to the VM, you can't fix the issue – but until you fix the issue, you cannot make a connection to the VM. With PS Direct, you can create a remoting session in the VM, as you see in this recipe.

Getting ready

This recipe uses `HV1`, a Windows Server Datacenter host on which you have installed the Hyper-V feature. You should have also created a VM of Windows Server called `PSDirect`. You should ensure that the `PSDirect` VM is running. This recipe demonstrates PowerShell Direct, so it doesn't matter which version of Windows Server you install so long as it is Windows Server 2016 or later (and you complete the installation of the OS inside the `PSDirect` VM).

You run the first part of this recipe on `HV1`. You then run the final steps of this recipe on `DC1`, which demonstrate how to connect to a VM remotely from the Hyper-V host.

How to do it...

1. Creating a credential object for `Reskit\Administrator`

```
$Admin = 'Administrator'
$PS    = 'Pa$$w0rd'
$RKP   = ConvertTo-SecureString -String $PS -AsPlainText -Force
$Cred  = [System.Management.Automation.PSCredential]::New(
          $Admin, $RKP)
```

2. Viewing the `PSDirect` VM

```
Get-VM -Name PSDirect
```

3. Invoking a command on the VM specifying VM name

```
$SBHT = @{
  VMName      = 'PSDirect'
  Credential  = $Cred
  ScriptBlock = {hostname}
}
Invoke-Command @SBHT
```

4. Invoking a command based on VMID

```
$VMID = (Get-VM -VMName PSDirect).VMId.Guid
Invoke-Command -VMid $VMID -Credential $Cred  -ScriptBlock {ipconfig}
```

5. Entering a PS remoting session with the `PSDirect` VM

```
Enter-PSSession -VMName PSDirect -Credential $Cred
Get-CimInstance -Class Win32_ComputerSystem
Exit-PSSession
```

You now run the rest of this recipe from `DC1`. Ensure that you log in as a domain administrator.

6. Creating credential for PSDirect inside `HV1`

```
$Admin = 'Administrator'
$PS    = 'Pa$$w0rd'
$RKP   = ConvertTo-SecureString -String $PS -AsPlainText -Force
$Cred  = [System.Management.Automation.PSCredential]::New(
          $Admin, $RKP)
```

7. Creating a remoting session to HV1 (Hyper-V host)

```
$RS = New-PSSession -ComputerName HV1 -Credential $Cred
```

8. Entering an interactive session with `HV1`

```
Enter-PSSession $RS
```

9. Creating credential for `PSDirect` inside `HV1`

```
$Admin = 'Administrator'
$PS    = 'Pa$$w0rd'
$RKP   = ConvertTo-SecureString -String $PS -AsPlainText -Force
$Cred  = [System.Management.Automation.PSCredential]::New(
$Admin, $RKP)
```

10. Entering and using the remoting session inside `PSDirect`

```
$PSDRS = New-PSSession -VMName PSDirect -Credential $Cred
Enter-PSSession -Session $PSDRS
HOSTNAME.EXE
```

11. Closing sessions

```
Exit-PSSession  # exits from session to PSDirect
Exit-PSSession  # exits from session to HV1
```

How it works...

In *step 1*, you create a PowerShell credential object. You use this object later in this recipe to enable connection to the `PSDirect` VM. This step creates no output.

In *step 2*, you use the `Get-VM` cmdlet to view the `PSDirect` VM, with output like this:

```
PS C:\Foo> # 2. Viewing the PSDirect VM
PS C:\Foo> Get-VM -Name PSDirect

Name     State   CPUUsage(%) MemoryAssigned(M) Uptime           Status             Version
----     -----   ----------- ----------------- ------           ------             -------
PSDirect Running 0           1024              00:26:54.8550000 Operating normally 10.0
```

Figure 12.5: Viewing the PSDirect Hyper-V VM

In *step 3*, you use `Invoke-Command` to run the hostname command in the `PSDirect` VM, with output like this:

```
PS C:\Foo> # 3. Invoking a command on the VM specifying VM name
PS C:\Foo> $SBHT = @{
             VMName      = 'PSDirect'
             Credential  = $RKCred
             ScriptBlock = {hostname}
           }
PS C:\Foo> Invoke-Command @SBHT
WIN-0TKMU3D2DFM
```

Figure 12.6: Checking the hostname of the PSDirect VM

In *step 4*, you invoke a command inside the `PSDirect` VM, but using the VM's GUID, with output like this:

```
PS C:\Foo> # 4. Invoking a command based on VMID
PS C:\Foo> $VMID = (Get-VM -VMName PSDirect).VMId.Guid
PS C:\Foo> Invoke-Command -VMid $VMID -Credential $RKCred  -ScriptBlock {ipconfig}

Windows IP Configuration

Ethernet adapter Ethernet:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :
```

Figure 12.7: Checking the hostname of the PSDirect VM by VM GUID

In *step 5*, you use the `Enter-PSSession` command to enter an interactive session with the `PSDirect` VM. Inside the remoting session, you run `Get-CimInstance` to return details of the computer system of the VM. The output should look like this:

```
PS C:\Foo> # 5. Entering a PS remoting session with the PSDirect VM
PS C:\Foo> Enter-PSSession -VMName PSDirect -Credential $RKCred
[PSDirect]: PS C:\Users\Administrator\Documents> Get-CimInstance -Class Win32_ComputerSystem

Name            PrimaryOwnerName  Domain     TotalPhysicalMemory  Model            Manufacturer
----            ----------------  ------     -------------------  -----            ------------
WIN-0TKMU3D2DFM  Windows User      WORKGROUP  1073270784           Virtual Machine  Microsoft Corporation

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

Figure 12.8: Entering a PS remoting session with the PSDirect VM

Now that you have created a working VM in `HV1`, you run the remainder of the VM setup steps remotely on `DC1`. In *step 6*, you create a PowerShell credential object you use to connect to the `PSDirect` VM inside `HV1`.

In *step 7*, you create a remoting session with `HV1`. This step creates no output. In *step 8*, you enter an interactive remoting session with `HV1`. The output of this step looks like this:

```
C:\Foo> # 8. Entering an interactive session with HV1
C:\Foo> Enter-PSSession $RS
[HV1]: PS C:\Users\administrator\Documents>
```

Figure 12.9: Entering a remoting session to HV1

In *step 9*, you create credentials for the `PSDirect` VM inside the `HV1` VM, producing no output.

In *step 10*, you enter an interactive session to `PSDirect` from the remoting session on `HV1`. Inside that session, you run the `HOSTNAME.EXE` command, which produces output like this:

```
[HV1]: PS C:\Users\administrator\Documents> # 10. Using PS Direct session to the VM
[HV1]: PS C:\Users\administrator\Documents> Enter-PSSession -VMName PSDirect -Credential $Cred
[HV1]: [PSDirect]: PS C:\Users\Administrator\Documents> HOSTNAME.EXE
WIN-0TKMU3D2DFM
```

Figure 12.10: Running HOSTNAME.EXE in session to PSDirect from HV1

In the final step in this recipe, *step 11*, you exit from the interactive sessions on the `PSDirect` and `HV1` VMs. The output from this step looks like this:

```
[HV1]: [PSDirect]: PS C:\Users\Administrator\Documents> # 11. CLosing sessions
[HV1]: [PSDirect]: PS C:\Users\Administrator\Documents> Exit-PSSession    # Exit from PSDirect
[HV1]: PS C:\Users\administrator\Documents> Exit-PSSession               # Exit from HV1
PS C:\Foo>
```

Figure 12.11: Exiting from remoting sessions

There's more...

In *step 3*, you use PowerShell Direct to enter a VM session and run a command. Assuming you installed Windows on the `PSDirect` VM using a supported version of Windows, the setup process generates a random machine name, in this case, `WIN-0TKMU3D2DFM`. The machine name of your VM is likely to be different.

In *step 4*, you use the PowerShell Direct feature to run a command inside the `PSDirect` VM. As you can see from the output in *Figure 12.7*, the NIC for this VM is disconnected. This step shows how you can use PowerShell Direct to enter a VM and view and restore networking.

Using Hyper-V VM groups

Hyper-V VM groups allow you to group VMs for automation. There are two types of VM groups you can create, `VMCollectionType` and `ManagementCollectionType`:

- A `VMCollectionType` VM group contains VMs
- A `ManagementCollectionType` VM group contains `VMCollectionType` VM groups

With VM groups, you could have two `VMCollectionType` VM groups, `SQLAccVMG` (which contains the `SQLAcct1`, `SQLAcct2`, and `SQLAcct3` VMs) and `SQLMfgVMG`, which contains the `SQLMfg1` and `SQLMfg2` VMs. You could then create a `ManagementCollectionType` VM group, `VM-All`, containing the two `VMCollectionType` VM groups.

The VM groups feature feels incomplete. For example, there are no `-VMGroup` parameters on any of the Hyper-V cmdlets enabling you to configure a VM group, and having two types of groups seems over the top. The feature could be useful for very large VM hosts running hundreds of VMs, if only from an organizational perspective, but all in all, this feature could be improved. In production, you might just consider using simple arrays of VM names (for example, `$ACCTSQVMS` to hold the names of the accounting VMs).

Getting ready

You run this recipe on `HV2`, a Windows Server 2022 server with the Hyper-V feature added. You configured this host in the *Installing Hyper-V inside Windows Server* recipe.

How to do it...

1. Creating VMs on HV2

```
$VMLocation  = 'C:\VM\VMs'   # Created in earlier recipe
# Create SQLAcct1
$VMN1        = 'SQLAcct1'
New-VM -Name $VMN1 -Path "$VMLocation\$VMN1"
# Create SQLAcct2
$VMN2        = 'SQLAcct2'
New-VM -Name $VMN2 -Path "$VMLocation\$VMN2"
 # Create SQLAcct3
$VMN3        = 'SQLAcct3'
New-VM -Name $VMN3 -Path "$VMLocation\$VMN3"
# Create SQLMfg1
```

```
$VMN4        = 'SQLMfg1'
New-VM -Name $VMN4 -Path "$VMLocation\$VMN4"
# Create SQLMfg2
$VMN5        = 'SQLMfg2'
New-VM -Name $VMN5 -Path "$VMLocation\$VMN5"
```

2. Viewing SQL VMs

```
Get-VM -Name SQL*
```

3. Creating Hyper-V VM groups

```
$VHGHT1 = @{
  Name      = 'SQLAccVMG'
  GroupType = 'VMCollectionType'
}
$VMGroupACC = New-VMGroup @VHGHT1
$VHGHT2 = @{
  Name      = 'SQLMfgVMG'
  GroupType = 'VMCollectionType'
}
$VMGroupMFG = New-VMGroup @VHGHT2
```

4. Displaying the VM groups on HV2

```
Get-VMGroup |
  Format-Table -Property Name, *Members, ComputerName
```

5. Creating arrays of group member VM names

```
$ACCVMs = 'SQLAcct1', 'SQLAcct2','SQLAcct3'
$MFGVms = 'SQLMfg1', 'SQLMfg2'
```

6. Adding members to the Accounting SQL VM group

```
Foreach ($Server in $ACCVMs) {
    $VM = Get-VM -Name $Server
    Add-VMGroupMember -Name SQLAccVMG -VM $VM
}
```

7. Adding members to the Manufacturing SQL VM group

```
Foreach ($Server in $MfgVMs) {
    $VM = Get-VM -Name $Server
    Add-VMGroupMember -Name  SQLMfgVMG -VM $VM
}
```

8. Viewing VM groups on HV2

```
Get-VMGroup |
 Format-Table -Property Name, *Members, ComputerName
```

9. Creating a `ManagementCollectionType` VM group

```
$VMGHT = @{
  Name      = 'VMMGSQL'
  GroupType = 'ManagementCollectionType'
}
$VMMGSQL = New-VMGroup  @VMGHT
```

10. Adding the two `VMCollectionType` groups to the `ManagementCollectionType` group

```
Add-VMGroupMember -Name VMMGSQL -VMGroupMember $VMGroupACC,
                                               $VMGroupMFG
```

11. Setting `FormatEnumerationLimit` to `99`

```
$FormatEnumerationLimit = 99
```

12. Viewing VM groups by type

```
Get-VMGroup | Sort-Object -Property GroupType |
  Format-Table -Property Name, GroupType, VMGroupMembers,
                         VMMembers
```

13. Stopping all the SQL VMs

```
Foreach ($VM in ((Get-VMGroup VMMGSQL).VMGroupMembers.vmmembers)) {
  Stop-VM -Name $vm.name -WarningAction SilentlyContinue
}
```

14. Setting the CPU count in all SQL VMs to `4`

```
Foreach ($VM in ((Get-VMGroup VMMGSQL).VMGroupMembers.VMMembers)) {
  Set-VMProcessor -VMName $VM.name -Count 4
}
```

15. Setting Accounting SQL VMs to have six processors

```
Foreach ($VM in ((Get-VMGroup SQLAccVMG).VMMembers)) {
  Set-VMProcessor -VMName $VM.name -Count 6
}
```

16. Checking processor counts for all VMs sorted by CPU count

```
$VMS = (Get-VMGroup -Name VMMGSQL).VMGroupMembers.VMMembers
Get-VMProcessor -VMName $VMS.Name |
  Sort-Object -Property Count -Descending |
    Format-Table -Property VMName, Count
```

17. Removing VMs from VM groups

```
$VMs = (Get-VMGroup -Name SQLAccVMG).VMMEMBERS
Foreach ($VM in $VMS)  {
  $X = Get-VM -vmname $VM.name
  Remove-VMGroupMember -Name SQLAccVMG -VM $x
  }
$VMs = (Get-VMGroup -Name SQLMFGVMG).VMMEMBERS
Foreach ($VM in $VMS)  {
  $X = Get-VM -vmname $VM.name
  Remove-VMGroupMember -Name SQLmfgvMG -VM $x
}
```

18. Removing VM groups from VM management groups

```
$VMGS = (Get-VMGroup -Name VMMGSQL).VMMembers
Foreach ($VMG in $VMGS)  {
  $X = Get-VMGroup -vmname $VMG.name
  Remove-VMGroupMember -Name VMMGSQL -VMGroupName $x
}
```

19. Removing all VM groups

```
Remove-VMGroup -Name SQLACCVMG -Force
Remove-VMGroup -Name SQLMFGVMG -Force
Remove-VMGroup -Name VMMGSQL   -Force
```

How it works...

In *step 1*, you create several VMs using the `New-VM` cmdlet, with the output of this step looking like this:

```
PS C:\Foo> # 1. Creating VMs on HV2
PS C:\Foo> $VMLocation  = 'C:\Vm\VMs'   # Created in earlier recipe
PS C:\Foo> # Create SQLAcct1
PS C:\Foo> $VMN1        = 'SQLAcct1'
PS C:\Foo> New-VM -Name $VMN1 -Path "$VMLocation\$VMN1"

Name     State CPUUsage(%) MemoryAssigned(M) Uptime   Status             Version
----     ----- ----------- ----------------- ------   ------             -------
SQLAcct1 Off   0           0                 00:00:00 Operating normally 10.0

PS C:\Foo> # Create SQLAcct2
PS C:\Foo> $VMN2        = 'SQLAcct2'
PS C:\Foo> New-VM -Name $VMN2 -Path "$VMLocation\$VMN2"

Name     State CPUUsage(%) MemoryAssigned(M) Uptime   Status             Version
----     ----- ----------- ----------------- ------   ------             -------
SQLAcct2 Off   0           0                 00:00:00 Operating normally 10.0

PS C:\Foo> $VMN3        = 'SQLAcct3'
PS C:\Foo> New-VM -Name $VMN3 -Path "$VMLocation\$VMN3"

Name     State CPUUsage(%) MemoryAssigned(M) Uptime   Status             Version
----     ----- ----------- ----------------- ------   ------             -------
SQLAcct3 Off   0           0                 00:00:00 Operating normally 10.0

PS C:\Foo> # Create SQLMfg1
PS C:\Foo> $VMN4        = 'SQLMfg1'
PS C:\Foo> New-VM -Name $VMN4 -Path "$VMLocation\$VMN4"

Name    State CPUUsage(%) MemoryAssigned(M) Uptime   Status             Version
----    ----- ----------- ----------------- ------   ------             -------
SQLMfg1 Off   0           0                 00:00:00 Operating normally 10.0

PS C:\Foo> # Create SQLMfg2
PS C:\Foo> $VMN5        = 'SQLMfg2'
PS C:\Foo> New-VM -Name $VMN5 -Path "$VMLocation\$VMN5"

Name    State CPUUsage(%) MemoryAssigned(M) Uptime   Status             Version
----    ----- ----------- ----------------- ------   ------             -------
SQLMfg2 Off   0           0                 00:00:00 Operating normally 10.0
```

Figure 12.12: Creating VMs for this recipe

In *step 2*, you use the `Get-VM` cmdlet to look at the six VMs you just created, with output like this:

```
PS C:\Foo> # 2. Viewing SQL VMs
PS C:\Foo> Get-VM -Name SQL*

Name     State CPUUsage(%) MemoryAssigned(M) Uptime   Status             Version
----     ----- ----------- ----------------- ------   ------             -------
SQLMfg1  Off   0           0                 00:00:00 Operating normally 10.0
SQLMfg2  Off   0           0                 00:00:00 Operating normally 10.0
SQLAcct1 Off   0           0                 00:00:00 Operating normally 10.0
SQLAcct3 Off   0           0                 00:00:00 Operating normally 10.0
SQLAcct2 Off   0           0                 00:00:00 Operating normally 10.0
```

Figure 12.13: Viewing SQL VMs

In *step 3*, you create several Hyper-V VM collection type VM groups, creating no output. In *step 4*, you examine the existing VM groups on `HV2`, with output like this:

```
PS C:\Foo> # 4. Displaying the VM groups
PS C:\Foo> Get-VMGroup |
             Format-Table -Property Name, *Members, ComputerName

Name      VMMembers VMGroupMembers ComputerName
----      --------- -------------- ------------
SQLAccVMG {}                       HV2
SQLMfgVMG {}                       HV2
```

Figure 12.14: Viewing VM groups on HV2

To simplify VM collection groups' creation, in *step 5*, you create arrays of the VM names. In *step 6*, you add VMs to the `SQLAccVMG` VM group, while in *step 7*, you add VMs to the `SQLMfgVMG` VM group. These three steps produce no console output.

Then, in *step 8*, you view the VM groups again, with output like this:

```
PS C:\Foo> # 8. Viewing VM groups on HV2
PS C:\Foo> Get-VMGroup |
             Format-Table -Property Name, *Members, ComputerName

Name      VMMembers                       VMGroupMembers ComputerName
----      ---------                       -------------- ------------
SQLAccVMG {SQLAcct1, SQLAcct3, SQLAcct2}                 HV2
SQLMfgVMG {SQLMfg1, SQLMfg2}                             HV2
```

Figure 12.15: Viewing VM groups on HV2

In *step 9*, you create a VM management collection group, and in *step 10*, you populate the VM management collection. To simplify the output, in *step 11*, you set the `$FormatEnumerationLimit` to `99`. These three steps create no output.

In *step 12*, you view all the fully populated VM groups, sorted by VM group type, with output like this:

```
PS C:\Foo> # 12. Viewing VM groups by type
PS C:\Foo> Get-VMGroup | Sort-Object -Property GroupType |
             Format-Table -Property Name, GroupType, VMGroupMembers,
                                    VMMembers

Name                    GroupType VMGroupMembers         VMMembers
----                    --------- --------------         ---------
SQLAccVMG         VMCollectionType                       {SQLAcct1, SQLAcct3, SQLAcct2}
SQLMfgVMG         VMCollectionType                       {SQLMfg1, SQLMfg2}
VMMGSQL   ManagementCollectionType {SQLAccVMG, SQLMfgVMG}
```

Figure 12.16: Viewing fully populated VM groups on HV2

In *step 13*, you use the VM groups you have created to stop, explicitly, all the SQL VMs. In *step 14*, you set the VMs in the `VMMGSQL` VM management collection group to have four virtual processors, and in *step 15*, you set the VMs in the `SQLAccVMG` VM collection group to each have six virtual processors. These three steps produce no output to the console.

In *step 16*, you review the virtual processors assigned to each SQL VM, like this:

```
PS C:\Foo> # 16. Checking processor counts for all VMs sorted by CPU count
PS C:\Foo> $VMS = (Get-VMGroup -Name VMMGSQL).VMGroupMembers.VMMembers
PS C:\Foo> Get-VMProcessor -VMName $VMS.Name |
             Sort-Object -Property Count -Descending |
               Format-Table -Property VMName, Count

VMName   Count
------   -----
SQLAcct1     6
SQLAcct3     6
SQLAcct2     6
SQLMfg1      4
SQLMfg2      4
```

Figure 12.17: Viewing virtual processors assigned to each SQL VM

In *step 17*, you remove all the VMs from the VM collection groups, and in *step 18*, you remove all the members from the VM management VM groups. Finally, in *step 19*, you remove all the VM groups. These three steps produce no output.

There's more...

In *step 11*, you set a value for the `$FormatEnumerationLimit` automatic variable. This variable controls how many repeating groups the `Format-*` cmdlets display. By default, PowerShell initializes this variable with a value of `4`. Using this default, in *step 11*, you would see at most 4 members of each of the VM groups, with PowerShell displaying "..." to indicate that you have more than 4 members. With this step, PowerShell can display up to 99 VM members for each VM group.

Configuring VM hardware

Configuring hardware in your VM is very much like configuring a physical computer, just without the need for a screwdriver. With a physical computer, you can adjust the CPUs and BIOS settings and adjust physical RAM, network interfaces, disk interfaces, disk devices, DVD drives (with/without a loaded DVD), and more. You can find each of these components within a Hyper-V VM. As you can see in the recipe, the PowerShell cmdlets make it simple to configure the virtual hardware available to any Hyper-V VM.

In this recipe, you adjust the VM's BIOS, CPU count, and memory and then add an SCSI controller. With the controller in place, you create a new virtual disk and assign it to the SCSI controller. Then, you view the results.

Like most physical servers, you cannot change all of these virtual components while your virtual server is up and running. You run this recipe from `HV1` and turn the `PSDirect` VM off before configuring the virtual hardware.

This recipe does not cover the VM's virtual NIC. By default, VMs (such as those created in the *Creating a Hyper-V VM* recipe) contain a single virtual NIC. You can add additional NICs to any VM should you wish to. You configure VM networking in the *Configuring VM networking* recipe later in this chapter.

Getting ready

This recipe uses `HV1`, a Windows Server Datacenter host on which you have installed the Hyper-V feature. You should also have created a VM of Windows Server called `PSDirect`.

How to do it...

1. Turning off the PSDirect VM

```
Stop-VM -VMName PSDirect
Get-VM -VMName PSDirect
```

2. Setting the `StartupOrder` in the VM's BIOS

```
$Order = 'IDE','CD','LegacyNetworkAdapter','Floppy'
Set-VMBios -VmName PSDirect -StartupOrder $Order
Get-VMBios PSDirect
```

3. Setting and viewing CPU count for `PSDirect`

```
Set-VMProcessor -VMName PSDirect -Count 2
Get-VMProcessor -VMName PSDirect |
  Format-Table VMName, Count
```

4. Setting and viewing `PSDirect` memory

```
$VMHT = [ordered] @{
  VMName               = 'PSDirect'
  DynamicMemoryEnabled = $true
  MinimumBytes         = 512MB
  StartupBytes         = 1GB
  MaximumBytes         = 2GB
}
Set-VMMemory @VMHT
Get-VMMemory -VMName PSDirect
```

5. Adding and viewing an SCSI controller in the `PSDirect` VM

```
Add-VMScsiController -VMName PSDirect
Get-VMScsiController -VMName PSDirect
```

6. Starting the `PSDirect` VM

```
Start-VM -VMName PSDirect
Wait-VM -VMName PSDirect -For IPAddress
```

7. Creating a new VHDX file for the `PSDirect` VM

```
$VHDPath = 'C:\VM\VHDs\PSDirect-D.VHDX'
New-VHD -Path $VHDPath -SizeBytes 8GB -Dynamic
```

8. Getting controller number of the newly added SCSI controller

```
$VM    = Get-VM -VMName PSDirect
$SCSIC = Get-VMScsiController -VM $VM|
           Select-Object -Last 1
```

9. Adding the VHD to the SCSI controller

```
$VHDHT = @{
    VMName             = 'PSDirect'
    ControllerType     = $SCSIC.ControllerNumber
    ControllerNumber   =  0
    ControllerLocation = 0
    Path                = $VHDPath
}
Add-VMHardDiskDrive @VHDHT
```

10. Viewing virtual drives in the `PSDirect` VM

```
Get-VMScsiController -VMName PSDirect |
  Select-Object -ExpandProperty Drives
```

How it works...

In *step 1*, you turn off the `PSDirect` VM and check the VM's status, with output like this:

```
PS C:\Foo> # 1. Turning off the PSDirect VM
PS C:\Foo> Stop-VM -VMName PSDirect
PS C:\Foo> Get-VM -VMName PSDirect

Name     State CPUUsage(%) MemoryAssigned(M) Uptime   Status             Version
----     ----- ----------- ----------------- ------   ------             -------
PSDirect Off   0           0                 00:00:00 Operating normally 10.0
```

Figure 12.18: Stopping the PSDirect VM

In *step 2*, you set the start up order for the VM, and then view the VM's virtual BIOS. The output of this step is:

```
PS C:\Foo> # 2. Setting the StartupOrder in the VM's BIOS
PS C:\Foo> $Order = 'IDE','CD','LegacyNetworkAdapter','Floppy'
PS C:\Foo> Set-VMBios -VmName PSDirect -StartupOrder $Order
PS C:\Foo> Get-VMBios PSDirect

VMName   StartupOrder                              NumLockEnabled
------   ------------                              --------------
PSDirect {IDE, CD, LegacyNetworkAdapter, Floppy} False
```

Figure 12.19: Updating and viewing the VM's BIOS

In *step 3*, you adjust and then view the number of virtual processors for the `PSDirect` VM, with output like this:

```
PS C:\Foo> # 3. Setting and viewing CPU count for PSDirect
PS C:\Foo> Set-VMProcessor -VMName PSDirect -Count 2
PS C:\Foo> Get-VMProcessor -VMName PSDirect |
             Format-Table VMName, Count

VMName   Count
------   -----
PSDirect     2
```

Figure 12.20: Updating and viewing the VM's BIOS

In *step 4*, you set the PS Direct's virtual memory and then view the memory settings, with the following output:

```
PS C:\Foo> # 4. Setting and viewing PSDirect memory
PS C:\Foo> $VMHT = [ordered] @{
             VMName               = 'PSDirect'
             DynamicMemoryEnabled = $true
             MinimumBytes         = 512MB
             StartupBytes         = 1GB
             MaximumBytes         = 2GB
           }
PS C:\Foo> Set-VMMemory @VMHT
PS C:\Foo> Get-VMMemory -VMName PSDirect

VMName   DynamicMemoryEnabled Minimum(M) Startup(M) Maximum(M)
------   -------------------- ---------- ---------- ----------
PSDirect True                 512        1024       2048
```

Figure 12.21: Changing and viewing VM memory

In *step 5*, you add an SCSI controller to the `PSDirect` VM. When you then view the SCSI controllers inside the `PSDirect` VM, you should see output like this:

```
PS C:\Foo> # 5. Adding and viewing a ScsiController in the PSDirect VM
PS C:\Foo> Add-VMScsiController -VMName PSDirect
PS C:\Foo> Get-VMScsiController -VMName PSDirect

VMName   ControllerNumber Drives
------   ---------------- ------
PSDirect 0                {}
PSDirect 1                {}
```

Figure 12.22: Adding and viewing SCSI virtual disk controllers

Now that you have adjusted the virtual hardware for the `PSDirect` VM, in *step 6*, you restart the VM and wait for the VM to come up. This step generates no output.

In *step 7*, you create a new virtual disk file for the `PSDirect` VM, with output like this:

```
PS C:\Foo> # 7. Creating a new VHDX file for the PSDirect VM
PS C:\Foo> $VHDPath = 'C:\VM\VHDs\PSDirect-D.VHDX'
PS C:\Foo> New-VHD -Path $VHDPath -SizeBytes 8GB -Dynamic

ComputerName            : HV1
Path                    : C:\VM\VHDs\PSDirect-D.VHDX
VhdFormat               : VHDX
VhdType                 : Dynamic
FileSize                : 4194304
Size                    : 8589934592
MinimumSize             :
LogicalSectorSize       : 512
PhysicalSectorSize      : 4096
BlockSize               : 33554432
ParentPath              :
DiskIdentifier          : 3959C590-8E5E-4CC0-AF19-270A68F167B5
FragmentationPercentage : 0
Alignment               : 1
Attached                : False
DiskNumber              :
IsPMEMCompatible        : False
AddressAbstractionType  : None
Number
```

Figure 12.23: Creating a new virtual disk drive

In *step 8*, you get the SCSI controllers in the `PSDirect` VM and the controller you added in *step 5*. Then, in *step 9*, you add the newly created virtual disk drive to the newly added SCSI controller. These two steps create no console output.

In *step 10*, you get the disk drives attached to SCSI controllers in the `PSDirect` VM, with output like this:

```
PS C:\Foo> # 10. Viewing drives in the PSDirect VM
PS C:\Foo> Get-VMScsiController -VMName PSDirect |
             Select-Object -ExpandProperty Drives

VMName   ControllerType ControllerNumber ControllerLocation DiskNumber Path
------   -------------- ---------------- ------------------ ---------- ----
PSDirect SCSI           0                0                             C:\Vm\Vhds\PSDirect-D.VHDX
```

Figure 12.24: Viewing the SCSI disk drives in the PSDirect VM

There's more...

With Hyper-V, you can only update some VM hardware configuration settings when you have turned off the VM. In *step 1*, you switch the VM off to enable you to make virtual BIOS settings. Once you have made these BIOS (and possibly other) changes, you can restart the VM.

In *step 2*, you set certain the start up order for the VM and view the settings. In production, you may have scripts like this that set VM settings to a preferred corporate standard, irrespective of the default values.

Some settings, such as SCSI disk controllers and disk drives, can be added and removed from a running VM. In *step 9*, you add a newly created virtual disk drive to the running `PSDirect` VM.

Configuring VM networking

In the *Creating a Hyper-V VM* recipe, you created a VM, `PSDirect`. This VM has, by default, a single network card that Hyper-V sets to acquire IP address details from DHCP. In this recipe, you assign the NIC to a switch and configure IP address details for the virtual network adapter.

Getting ready

You run this recipe on `HV1`, which uses the `PSDirect` VM you created in the *Creating a Hyper-V VM* recipe. This recipe also makes use of a DHCP server running on `DC1`. In *Chapter 7*, *Managing Networking in the Enterprise*, you set the DHCP service up in the *Installing DHCP* recipe. Then, you configured the DHCP server in the *Configuring DHCP scopes and options* recipe.

This chapter uses the `PSDirect` VM you created earlier. When you build this machine using the normal setup routine, Windows assigns a random machine name, which you saw in a previous recipe (*Using PowerShell Direct*). In this recipe, you also change the name of the host inside to `Wolf`.

In this recipe, you set the PSDirect VM's networking card to enable MAC address spoofing. This step enables the VM to see the network and get an IP address from the DHCP server on DC1. If you are running HV1 as a VM, you must also enable MAC spoofing on the NIC(s) in this VM on the VM host of HV1. You should also enable MAC spoofing on HV2.

How to do it...

1. Setting the PSDirect VM's NIC

```
Get-VM PSDirect |
  Set-VMNetworkAdapter -MacAddressSpoofing On
```

2. Getting NIC details and any IP addresses from the PSDirect VM

```
Get-VMNetworkAdapter -VMName PSDirect
```

3. Creating a credential then getting VM networking details

```
$RKAn = 'localhost\Administrator'
$PS = 'Pa$$w0rd'
$RKP = ConvertTo-SecureString -String $PS -AsPlainText -Force
$T = 'System.Management.Automation.PSCredential'
$RKCred = New-Object -TypeName $T -ArgumentList $RKAn, $RKP
$VMHT = @{
    VMName      = 'PSDirect'
    ScriptBlock = {Get-NetIPConfiguration | Format-Table }
    Credential  = $RKCred
}
Invoke-Command @VMHT | Format-List
```

4. Creating a virtual switch on HV1

```
$VSHT = @{
    Name           = 'External'
    NetAdapterName = 'Ethernet'
    Notes          = 'Created on HV1'
}
New-VMSwitch @VSHT
```

5. Connecting the PSDirect VM's NIC to the External switch

```
Connect-VMNetworkAdapter -VMName PSDirect -SwitchName External
```

6. Viewing VM networking information

```
Get-VMNetworkAdapter -VMName PSDirect
```

7. Observing the IP address in the `PSDirect` VM

```
$NCHT = @{
    VMName      = 'PSDirect'
    ScriptBlock = {Get-NetIPConfiguration}
    Credential  = $RKCred
}
Invoke-Command @NCHT
```

8. Viewing the hostname on `PSDirect`

```
$NCHT.ScriptBlock = {hostname}
Invoke-Command @NCHT
```

9. Changing the name of the host in the `PSDirect` VM

```
$NCHT.ScriptBlock = {Rename-Computer -NewName Wolf -Force}
Invoke-Command @NCHT
```

10. Rebooting and waiting for the restarted PSDirect VM

```
Restart-VM -VMName PSDirect -Wait -For IPAddress -Force
```

11. Getting hostname of the `PSDirect` VM

```
$NCHT.ScriptBlock = {HOSTNAME}
Invoke-Command @NCHT
```

How it works...

In *step 1*, you set the `PSDirect` VM's NIC to enable MAC address spoofing. There is no output from this step.

In *step 2*, you get the NIC details for the NIC assigned to the `PSDirect` VM, with output like this:

```
PS C:\Foo> # 2. Getting NIC details and any IP addresses from the PSDirect VM
PS C:\Foo> Get-VMNetworkAdapter -VMName PSDirect

Name            IsManagementOs VMName   SwitchName MacAddress   Status IPAddresses
----            -------------- ------   ---------- ----------   ------ -----------
Network Adapter False          PSDirect            00155D0AC900 {Ok}   {169.254.200.232, fe80::fca7:6a7:acfa:c8e8}
```

Figure 12.25: Viewing the NIC details for the PSDirect VM

In *step 3*, you create a credential object for the `PSDirect` VM. Then you use that credential to run the `Get-NetIPConfiguration` information from inside the VM, with output like this:

```
PS C:\Foo> # 3. Creating a credential then getting VM networking details
PS C:\Foo> $RKAn = 'localhost\Administrator'
PS C:\Foo> $PS = 'Pa$$w0rd'
PS C:\Foo> $RKP = ConvertTo-SecureString -String $PS -AsPlainText -Force
PS C:\Foo> $T = 'System.Management.Automation.PSCredential'
PS C:\Foo> $RKCred = New-Object -TypeName $T -ArgumentList $RKAn, $RKP
PS C:\Foo> $VMHT = @{
             VMName      = 'PSDirect'
             ScriptBlock = {Get-NetIPConfiguration | Format-Table }
             Credential  = $RKCred
           }
PS C:\Foo> Invoke-Command @VMHT
ComputerName     InterfaceAlias InterfaceIndex InterfaceDescription                CompartmentId NetAdapter
------------     -------------- -------------- --------------------                ------------- ----------
WIN-0TKMU3D2DFM Ethernet                    5 Microsoft Hyper-V Network Adapter               1 MSFT_NetAdapter (CreationClassName = "MSFT_NetAdapter",
                                                                                                DeviceID = "{8EF33029-8B18-4FE6-B870-9AB1E1430AC2…
```

Figure 12.26: Viewing the IP configuration inside the PSDirect VM

In *step 4*, you create a new Hyper-V VM switch inside the `HV1` VM host, which creates the following output:

```
PS C:\Foo> # 4. Creating a virtual switch on HV1
PS C:\Foo> $VSHT = @{
             Name            = 'External'
             NetAdapterName = 'Ethernet'
             Notes           = 'Created on HV1'
           }
PS C:\Foo> New-VMSwitch @VSHT

Name     SwitchType NetAdapterInterfaceDescription
----     ---------- ------------------------------
External External   Microsoft Hyper-V Network Adapter
```

Figure 12.27: Creating a new virtual switch inside HV1

In *step 5*, you connect the NIC in the `PSDirect` VM to the VM switch, creating no output. In *step 6*, you can re-view the networking information for the `PSDirect` VM, with output like this:

```
PS C:\Foo> # 6. Viewing VM networking information
PS C:\Foo> Get-VMNetworkAdapter -VMName PSDirect

Name            IsManagementOs VMName   SwitchName MacAddress   Status IPAddresses
----            -------------- ------   ---------- ----------   ------ -----------
Network Adapter False          PSDirect External   00155D0AC900 {Ok}   {10.10.10.150, fe80::fca7:6a7:acfa:c8e8}
```

Figure 12.28: Viewing the PSDirect NIC settings

In *step 7*, you execute a script block inside the `PSDirect` VM to return the VM's network details, which produces output that looks like this:

```
PS C:\Foo> # 7. Observing the IP address in the PSDirect VM
PS C:\Foo> $NCHT = @{
             VMName      = 'PSDirect'
             ScriptBlock = {Get-NetIPConfiguration}
             Credential  = $RKCred
           }
PS C:\Foo> Invoke-Command @NCHT

InterfaceAlias       : Ethernet
InterfaceIndex       : 5
InterfaceDescription : Microsoft Hyper-V Network Adapter
NetAdapter.Status    :
NetProfile.Name      : Reskit.Org
IPv4Address          : 10.10.10.150
IPv6DefaultGateway   :
IPv4DefaultGateway   : 10.10.10.254
DNSServer            : 10.10.10.11
                       10.10.10.10
PSComputerName       : PSDirect
```

Figure 12.29: Observing the PSDirect network settings

In *step 8*, you view the existing hostname of the `PSDirect` VM, with output that looks like this:

```
PS C:\Foo> # 8. Viewing the hostname on PSDirect
PS C:\Foo> #    Reuse the hash table from step 6
PS C:\Foo> $NCHT.ScriptBlock = {hostname}
PS C:\Foo> Invoke-Command @NCHT
WIN-0TKMU3D2DFM
```

Figure 12.30: Viewing the PSDirect hostname

In *step 9*, you use the `Rename-Computer` cmdlet, running inside the `PSDirect` VM. This step changes the hostname of the VM to `Wolf` and produces the following output:

```
PS C:\Foo> # 9.  Changing the name of the host in the PSDirect VM
PS C:\Foo> $NCHT.ScriptBlock = {Rename-Computer -NewName Wolf -Force}
PS C:\Foo> Invoke-Command @NCHT
WARNING: The changes will take effect after you restart the computer WIN-0TKMU3D2DFM.
```

Figure 12.31: Changing the VM hostname

In *step 10*, you reboot the `PSDirect` VM, generating no console output. After the VM has restarted, in *step 11*, you run the `HOSTNAME` command inside the VM, with output like this:

```
PS C:\Foo> # 11. Getting hostname of the PSDirect VM
PS C:\Foo> $NCHT.ScriptBlock = {hostname}
PS C:\Foo> Invoke-Command @NCHT
Wolf
```

Figure 12.32: Viewing the updated hostname in the PSDirect VM

There's more...

In *step 1*, you set the `PSDirect` VM's NIC to enable MAC address spoofing. You can read more about MAC address spoofing with Hyper-V at `https://charbelnemnom.com/how-to-set-dynamic-mac-address-on-a-hyper-v-vm-with-powershell/`.

In *step 5*, you connect the `PSDirect` VM's NIC to the virtual switch you created on `HV1`. Since the `PSDirect` VM's NIC is (by default) set to get IP addresses via DHCP, as soon as you connect the virtual NIC to the switch, the VM should acquire the IP address configuration from the `DC1` host, which is running DHCP. You see the DHCP acquired IP configuration details in *step 6*.

Implementing nested virtualization

Nested virtualization is a feature that enables a Hyper-V VM to host VMs that also have virtualization enabled. You could, for example, take a Hyper-V host (say, `HV1`) and on that host run a VM (say, `PSDirect`). With nested virtualization, you could enable your `PSDirect` VM to host VMs. You could then create a nested VM inside the `PSDirect` VM called `NestedVM`.

Nested VMs have many uses. First, nested VMs hosted in one VM are provided with hardware isolation from nested VMs run in other VMs. In this case, nested virtualization provides a further level of security for VMs.

Nested virtualization is also helpful for testing and education/training. You could give students a single VM (on a large blade server, for example). Nested virtualization enables them to create additional VMs as part of the course lab work. And most IT pros just find it cool! You can, for example, run all the recipes in this chapter using nested virtualization.

Enabling nested Hyper-V is very simple. First, you must update the virtual CPU in the VM you want to support nesting. Therefore, in this recipe, you adjust the virtual CPU in the `PSDirect` VM to expose the virtualization extensions. Changing the BIOS to do this must be done after you turn off the VM. After you restart the VM, you install the Hyper-V feature and create the `NestedVM` nested VM. This recipe does not show the details of configuring the `NestedVM` VM. These details are an exercise for the reader.

Getting ready

This recipe uses the HV1 Hyper-V host, with an existing Hyper-V VM, PSDirect, available. The recipe assumes that you have set up PSDirect as discussed in the *Creating a Hyper-V VM* recipe earlier in this chapter.

How to do it...

1. Stopping the PSDirect VM

```
Stop-VM -VMName PSDirect
```

2. Setting the VM's processor to support virtualization

```
$VMHT = @{
  VMName                          = 'PSDirect'
  ExposeVirtualizationExtensions = $true
}
Set-VMProcessor @VMHT
Get-VMProcessor -VMName PSDirect |
  Format-Table -Property Name, Count,
                         ExposeVirtualizationExtensions
```

3. Starting the PSDirect VM

```
Start-VM -VMName PSDirect
Wait-VM  -VMName PSDirect -For Heartbeat
Get-VM   -VMName PSDirect
```

4. Creating credentials for PSDirect

```
$User = 'Wolf\Administrator'
$PHT = @{
  String      = 'Pa$$w0rd'
  AsPlainText = $true
  Force       = $true
}
$PSS  = ConvertTo-SecureString @PHT
$Type = 'System.Management.Automation.PSCredential'
$CredRK = New-Object -TypeName $Type -ArgumentList $User, $PSS
```

5. Creating a script block for remote execution

```
$SB = {
  Install-WindowsFeature -Name Hyper-V -IncludeManagementTools
}
```

6. Creating a remoting session to PSDirect

```
$Session = New-PSSession -VMName PSDirect -Credential $CredRK
```

7. Installing Hyper-V inside `PSDirect`

```
$IHT = @{
  Session     = $Session
  ScriptBlock = $SB
}
Invoke-Command @IHT
```

8. Restarting the VM to finish adding Hyper-V to `PSDirect`

```
Stop-VM  -VMName PSDirect
Start-VM -VMName PSDirect
Wait-VM  -VMName PSDirect -For IPAddress
Get-VM   -VMName PSDirect
```

9. Creating a nested VM inside the `PSDirect` VM

```
$SB2 = {
        $VMname = 'NestedVM'
        New-VM -Name $VMname -MemoryStartupBytes 1GB | Out-Null
        Get-VM
}
$IHT2 = @{
  VMName = 'PSDirect'
  ScriptBlock = $SB2
}
Invoke-Command @IHT2 -Credential $CredRK
```

How it works...

In *step 1*, you ensure that the `PSDirect` VM has stopped, creating no console output. In *step 2*, you adjust the VM's virtual processor to support virtualization. Then you review the processor settings, which look like this:

```
PS C:\Foo> # 2. Setting the VM's processor to support virtualization
PS C:\Foo> $VMHT = @{
             VMName                       = 'PSDirect'
             ExposeVirtualizationExtensions = $true
           }
PS C:\Foo> Set-VMProcessor @VMHT
PS C:\Foo> Get-VMProcessor -VMName PSDirect |
             Format-Table -Property Name, Count,
                                    ExposeVirtualizationExtensions

Name      Count ExposeVirtualizationExtensions
----      ----- ------------------------------
Processor     2                           True
```

Figure 12.33: Viewing the virtual processor inside PSDirect

In *step 3*, you start the `PSDirect` VM and then view it using `Get-VM`, which looks like this:

```
PS C:\Foo> # 3. Starting the PSDirect VM
PS C:\Foo> Start-VM -VMName PSDirect
PS C:\Foo> Wait-VM  -VMName PSDirect -For Heartbeat
PS C:\Foo> Get-VM   -VMName PSDirect

Name     State   CPUUsage(%) MemoryAssigned(M) Uptime           Status             Version
----     -----   ----------- ----------------- ------           ------             -------
PSDirect Running 0           722               00:00:48.9830000 Operating normally 10.0
```

Figure 12.34: Starting and viewing PSDirect

In *step 4*, you create a PowerShell credential object for the `PSDirect` VM. In *step 5,* you create a script block that installs the Hyper-V feature. Then in *step 6*, you create a PowerShell remoting session with the `PSDirect` VM. These three steps produce no output to the console.

In *step 7*, you use the remoting session to install the Hyper-V feature inside the `PSDirect` VM, producing output like this:

```
PS C:\Foo> # 7. Installing Hyper-V inside PSDirect
PS C:\Foo> $IHT = @{
             Session     = $Session
             ScriptBlock = $SB
           }
PS C:\Foo> Invoke-Command @IHT

PSComputerName : PSDirect
RunspaceId     : 6565ef1f-c33a-4b8b-94d5-603ed98781be
Success        : True
RestartNeeded  : Yes
FeatureResult  : {Hyper-V, Hyper-V Module for Windows PowerShell,
                 Hyper-V GUI Management Tools, Remote Server Administration Tools,
                 Hyper-V Management Tools, Role Administration Tools}
ExitCode       : SuccessRestartRequired

WARNING: You must restart this server to finish the installation process
```

Figure 12.35: Installing Hyper-V inside the PSDirect VM

As you can see from the output, after installing the Hyper-V feature inside `PSDirect`, you need to reboot the VM to complete the installation process. In *step 8*, you restart `PSDirect`, wait for it to start, and view the VM details, with output like this:

```
PS C:\Foo> # 8. Restarting the VM to finish adding Hyper-V to PSDirect
PS C:\Foo> Stop-VM  -VMName PSDirect
PS C:\Foo> Wait-VM  -VMName PSDirect -For IPAddress
PS C:\Foo> Get-VM   -VMName PSDirect

Name     State   CPUUsage(%) MemoryAssigned(M) Uptime           Status             Version
----     -----   ----------- ----------------- ------           ------             -------
PSDirect Running 18          1024              00:00:57.6530000 Operating normally 10.0
```

Figure 12.36: Restarting the PSDirect VM

In the final step in this recipe, *step 9*, you create a new nested VM, `NestedVM`, inside the `PSDirect` VM, which produces console output like this:

```
PS C:\Foo> # 9. Creating a nested VM inside the PSDirect VM
PS C:\Foo> $SB2 = {
               $VMname = 'NestedVM'
               New-VM -Name $VMname -MemoryStartupBytes 1GB
               Get-VM
           }
PS C:\Foo> $IHT2 = @{
                 VMName = 'PSDirect'
                 ScriptBlock = $SB2
           }
PS C:\Foo> Invoke-Command @IHT2 -Credential $CredRK

Name     State CPUUsage(%) MemoryAssigned(M) Uptime   Status             Version PSComputerName
----     ----- ----------- ----------------- ------   ------             ------- --------------
NestedVM Off   0           0                 00:00:00 Operating normally 9.0     PSDirect
```

Figure 12.37: Creating a nested VM inside PSDirect

There's more...

In *step 7*, you install the Hyper-V feature in the nested VM (`NestedVM`) inside the `PSDirect` VM. The installation of Hyper-V inside the `PSDirect` VM is successful, as you can see in this step's output in *Figure 12.35*. Had the `PSDirect` VM not supported virtualization, the `Install-WindowsFeature` cmdlet would have thrown an error. Likewise, had you not installed Hyper-V in `PSDirect`, *step 9* would have failed without creating the VM. If you choose, you can adjust the steps from the *Creating a Hyper-V VM* recipe to install and configure an OS in the `NestedVM`.

Managing VM state

Hyper-V provides you with the ability to start, stop, and pause a Hyper-V VM. You can also save and restore a VM. You use the Hyper-V cmdlets to manage your VMs either locally (that is, on the Hyper-V host in a remote desktop (RDP) or PowerShell remoting session) or use RSAT tools to manage the state of VMs on remote Hyper-V hosts.

You can start and stop VMs either directly or via the task scheduler. You might want to start up a few VMs every working morning and stop them each evening. If you have provisioned your Hyper-V host with spinning disks, starting multiple VMs at once stresses the storage subsystem, especially if you are using any form of RAID on the disk drives you use to hold your virtual disks. Depending on your hardware, you can sometimes hear the IO Blender effect starting up a small VM farm. Even with solid-state disks, starting several VMs at once puts a considerable load on the Windows storage system. In such cases, you might pause a VM and let others start faster.

Saving a VM can be helpful in avoiding a long start up process. If you have created multiple VMs to test out the recipes in this book, you might save VMs and restart them as you move through the chapters.

Getting ready

You run this recipe on the `HV1` host after you have created the `PSDirect` VM. You created that VM in the *Creating a Hyper-V VM* recipe.

How to do it...

1. Getting the VM's state to check if it is off

```
Stop-VM -Name PSDirect -WarningAction SilentlyContinue
Get-VM -Name PSDirect
```

2. Starting the VM

```
Start-VM -VMName PSDirect
Wait-VM -VMName PSDirect -For IPAddress
Get-VM -VMName PSDirect
```

3. Suspending and viewing the `PSDirect` VM

```
Suspend-VM -VMName PSDirect
Get-VM -VMName PSDirect
```

4. Resuming the `PSDirect` VM

```
Resume-VM -VMName PSDirect
Get-VM -VMName PSDirect
```

5. Saving the VM

```
Save-VM -VMName PSDirect
Get-VM -VMName PSDirect
```

6. Resuming the saved VM and viewing the status

```
Start-VM -VMName PSDirect
Get-VM -VMName PSDirect
```

7. Restarting the `PSDirect` VM

```
Restart-VM -VMName PSDirect -Force
Get-VM     -VMName PSDirect
```

8. Waiting for the `PSDirect` VM to get an IP address

```
Wait-VM    -VMName PSDirect -For IPaddress
Get-VM     -VMName PSDirect
```

9. Performing a hard power off on the `PSDirect` VM

```
Stop-VM -VMName PSDirect -TurnOff
Get-VM  -VMname PSDirect
```

How it works...

In *step 1*, you stop the `PSDirect` VM and check its status. The output looks like this:

```
PS C:\Foo> # 1. Getting the VM's state to check if it is off
PS C:\Foo> Stop-VM -Name PSDirect -WarningAction SilentlyContinue
PS C:\Foo> Get-VM -Name PSDirect

Name     State CPUUsage(%) MemoryAssigned(M) Uptime   Status             Version
----     ----- ----------- ----------------- ------   ------             -------
PSDirect Off   0           0                 00:00:00 Operating normally 10.0
```

Figure 12.38: Stopping the PSDirect VM

In *step 2*, you use `Start-VM` to start the `PSDirect` VM, and then use `Wait-VM` to wait for the VM to start up and get an IP address. The output from this step looks like this:

```
PS C:\Foo> # 2. Starting the VM
PS C:\Foo> Start-VM -VMName PSDirect
PS C:\Foo> Wait-VM -VMName PSDirect -For IPAddress
PS C:\Foo> Get-VM -VMName PSDirect

Name     State   CPUUsage(%) MemoryAssigned(M) Uptime           Status             Version
----     -----   ----------- ----------------- ------           ------             -------
PSDirect Running 21          1024              00:00:48.8350000 Operating normally 10.0
```

Figure 12.39: Starting the PSDirect VM

In *step 3*, you use `Suspend-VM` to pause the `PSDirect` VM and observe its status, with the following output:

```
PS C:\Foo> # 3. Suspending and viewing the PSDirect VM
PS C:\Foo> Suspend-VM -VMName PSDirect
PS C:\Foo> Get-VM -VMName PSDirect

Name     State  CPUUsage(%) MemoryAssigned(M) Uptime           Status             Version
----     -----  ----------- ----------------- ------           ------             -------
PSDirect Paused 0           1024              00:01:17.9540000 Operating normally 10.0
```

Figure 12.40: Pausing the PSDirect VM

Having paused the `PSDirect` VM, in *step 4*, you use the `Resume-VM` cmdlet to unpause the VM, with the following output:

```
PS C:\Foo> # 4. Resuming the VM
PS C:\Foo> Resume-VM -VMName PSDirect
PS C:\Foo> Get-VM -VMName PSDirect

Name     State   CPUUsage(%) MemoryAssigned(M) Uptime           Status             Version
----     -----   ----------- ----------------- ------           ------             -------
PSDirect Running 14          1024              00:01:20.6640000 Operating normally 10.0
```

Figure 12.41: Resuming the PSDirect VM

In *step 5*, you save the `PSDirect` VM, with output like this:

```
PS C:\Foo> # 5. Saving the VM
PS C:\Foo> Save-VM -VMName PSDirect
PS C:\Foo> Get-VM -VMName PSDirect

Name     State CPUUsage(%) MemoryAssigned(M) Uptime   Status             Version
----     ----- ----------- ----------------- ------   ------             -------
PSDirect Saved 0           0                 00:00:00 Operating normally 10.0
```

Figure 12.42: Saving the PSDirect VM

You can resume a saved VM by using the `Start-VM` command, as you see in *step 6*, which creates output like this:

```
PS C:\Foo> # 6. Resuming the saved VM and viewing the status
PS C:\Foo> Start-VM -VMName PSDirect
PS C:\Foo> Get-Vm -VMName PSDirect

Name     State   CPUUsage(%) MemoryAssigned(M) Uptime           Status             Version
----     -----   ----------- ----------------- ------           ------             -------
PSDirect Running 0           1024              00:00:00.1520000 Operating normally 10.0
```

Figure 12.43: Resuming the PSDirect VM (after you have saved it)

Step 7 demonstrates using the `Restart-VM` cmdlet to restart the `PSDirect` VM. The output from this step looks like this:

```
PS C:\Foo> # 7. Restarting a VM
PS C:\Foo> Restart-VM -VMName PSDirect -Force
PS C:\Foo> Get-VM     -VMName PSDirect

Name     State   CPUUsage(%) MemoryAssigned(M) Uptime           Status             Version
----     -----   ----------- ----------------- ------           ------             -------
PSDirect Running 0           1024              00:00:00.3960000 Operating normally 10.0
```

Figure 12.44: Restarting the PSDirect VM

In the previous step, you restarted the VM, which can take some time, depending on your VM host and VM size. In *step 8*, you wait for the VM to restart, and then use `Get-VM` to observe the status of the `PSDirect` VM, like this:

```
PS C:\Foo> # 8. Waiting for the PSDirect VM to get an IP address
PS C:\Foo> Wait-VM    -VMName PSDirect -For IPaddress
PS C:\Foo> Get-VM     -VMName PSDirect

Name     State   CPUUsage(%) MemoryAssigned(M) Uptime           Status             Version
----     -----   ----------- ----------------- ------           ------             -------
PSDirect Running 23          1024              00:00:50.8910000 Operating normally 10.0
```

Figure 12.45: Waiting for the PSDirect VM to restart

In the final step in this recipe, *step 9*, you perform a hard power off of the `PSDirect` VM, with output like this:

```
PS C:\Vm\VMs> # 9. Performing a hard power off on the PSDirect VM
PS C:\Vm\VMs> Stop-VM -VMName PSDirect -TurnOff
PS C:\Vm\VMs> Get-VM  -VMname PSDirect

Name     State CPUUsage(%) MemoryAssigned(M) Uptime   Status             Version
----     ----- ----------- ----------------- ------   ------             -------
PSDirect Off   0           0                 00:00:00 Operating normally 10.0
```

Figure 12.46: Turning off the PSDirect VM

There's more...

Managing the state of VMs is straightforward with the cmdlets in the Hyper-V module. You can start and stop a VM just like you do with physical machines. The Hyper-V cmdlets allow you to start a VM on a Hyper-V host, and wait until it is far enough along in the start up process to enable you to interact with the VM. This is an important feature for automation.

In some scenarios, saving a VM is very useful. If you have VMs that you use infrequently, you can save them rather than shutting them down, thereby improving the VM start up time. You might also save all the VMs when you need to reboot the VM host.

Managing VM and storage movement

Hyper-V enables you to move a VM to a new VM host and move a VM's storage to a new location. Moving a VM and moving a VM's storage are two important features you can use to manage your Hyper-V hosts.

With live migration, you can move a Hyper-V VM to a different VM host with no downtime. Storage migration works best when the VM is held on shared storage (via a fiber channel SAN, iSCSI, or SMB). You can also move a VM's storage (any VHD/VHDX associated with the VM) to a different location. You can combine these and move a VM supported by local storage to another Hyper-V host, moving both the VM and the underlying storage.

In this recipe, you first move the storage for the `PSDirect` VM. You created this VM in the *Creating a Hyper-V VM* recipe and stored the VM configuration and the VM's VHD on the `H:` drive. To move the storage, you create a new SMB share and then move the VM's storage to the new SMB share.

In the second part of this recipe, you do a live migration of the `PSDirect` VM from `HV1` to `HV2`. This VM movement is known as live migration because you are migrating a live VM that stays up and running during migration.

Getting ready

In this recipe, you use the `HV1` and `HV2` systems (Windows 2022 Server with Hyper-V loaded) as set up in the *Installing Hyper-V inside Windows Server* recipe and the `PSDirect` VM created in the *Creating a Hyper-V VM* recipe.

In the first part of this recipe, you move the storage for the `PSDirect` VM. You created this VM in the *Creating a Hyper-V VM* recipe. You should ensure that this VM is up and running.

You run the first part of this recipe on `HV1` to move the `PSDirect` VM from `HV1` to `HV2`. Once you have moved the VM to `HV2`, you run the final part of this recipe on `HV2` to move the VM back to `HV1`. If you attempted to run the last part of this recipe remotely, for example, for a client machine or from `HV1` (suitably wrapped up in a script block you run with `Invoke-Command`), you would see a Kerberos error due to the Kerberos double hop problem. To overcome that issue, you could have deployed Windows **Credential Security System Provider** (**CredSSP**).

How to do it...

1. Viewing the PSDirect VM on HV1 and verifying that it is turned on and running

   ```
   Get-VM -Name PSDirect -Computer HV1
   ```

2. Getting the VM configuration location

   ```
   (Get-VM -Name PSDirect).ConfigurationLocation
   ```

3. Getting virtual hard drive locations

   ```
   Get-VMHardDiskDrive -VMName PSDirect |
     Format-Table -Property VMName, ControllerType, Path
   ```

4. Moving the VM's storage to the `C:\PSDirectNew` folder

```
$MHT = @{
  Name                  = 'PSDirect'
  DestinationStoragePath = 'C:\PSDirectNew'
}
Move-VMStorage @MHT
```

5. Viewing the configuration details after moving the VM's storage

```
(Get-VM -Name PSDirect).ConfigurationLocation
Get-VMHardDiskDrive -VMName PSDirect |
  Format-Table -Property VMName, ControllerType, Path
```

6. Getting the VM details for VMs from HV2

```
Get-VM -ComputerName HV2
```

7. Creating `External` virtual switch on HV2

```
$SB = {
  $NSHT = @{
    Name           = 'External'
    NetAdapterName = 'Ethernet'
    ALLOWmAnagementOS = $true
  }
  New-VMSwitch @NSHT
}
Invoke-Command -ScriptBlock $SB -ComputerName HV2
```

8. Enabling VM migration from both HV1 and HV2

```
Enable-VMMigration -ComputerName HV1, HV2
```

9. Configuring VM migration on both hosts

```
$SVHT = @{
  UseAnyNetworkForMigration                = $true
  ComputerName                             = 'HV1', 'HV2'
  VirtualMachineMigrationAuthenticationType =  'Kerberos'
  VirtualMachineMigrationPerformanceOption  = 'Compression'
}
Set-VMHost @SVHT
```

10. Moving the `PSDirect` VM to HV2

```
$Start = Get-Date
$VMHT = @{
  Name                  = 'PSDirect'
  ComputerName          = 'HV1'
```

```
  DestinationHost        = 'HV2'
  IncludeStorage         =  $true
  DestinationStoragePath = 'C:\PSDirect' # on HV2
}
Move-VM @VMHT
$Finish = Get-Date
($Finish - $Start)
```

11. Displaying the time taken to migrate

```
$OS = "Migration took: [{0:n2}] minutes"
($OS -f ($($Finish-$Start).TotalMinutes))
```

12. Checking the VMs on HV1

```
Get-VM -ComputerName HV1
```

13. Checking the VMs on HV2

```
Get-VM -ComputerName HV2
```

14. Looking at the details of the PSDirect VM on HV2

```
((Get-VM -Name PSDirect -Computer HV2).ConfigurationLocation)
Get-VMHardDiskDrive -VMName PSDirect -Computer HV2  |
  Format-Table -Property VMName, Path
```

Run the remainder of this recipe on HV2 directly.

15. Moving the PSDirect VM back to HV1

```
$Start2 = Get-Date
$VMHT2 = @{
    Name                  = 'PSDirect'
    ComputerName           = 'HV2'
    DestinationHost        = 'HV1'
    IncludeStorage         =  $true
    DestinationStoragePath = 'C:\VM\VHDs\PSDirect' # on HV1
}
Move-VM @VMHT2
$Finish2 = Get-Date
```

16. Displaying the time taken to migrate back to HV1

```
$OS = "Migration back to HV1 took: [{0:n2}] minutes"
($OS -f ($($Finish-$Start).TotalMinutes))
```

How it works...

In *step 1*, you check the status of the `PSDirect` VM to confirm that it is up and running. The output should look like this:

```
PS C:\Foo> # 1. Viewing the PSDirect VM on HV1 and verifying that it is turned off and not saved
PS C:\Foo> Get-VM -Name PSDirect -Computer HV1

Name     State   CPUUsage(%) MemoryAssigned(M) Uptime          Status             Version
----     -----   ----------- ----------------- ------          ------             -------
PSDirect Running 6           1024              00:00:14.0820000 Operating normally 10.0
```

Figure 12.47: Checking the status of the PSDirect VM

In *step 2*, you determine the location that Hyper-V is using to store the VM configuration for the `PSDirect` VM, with console output like this:

```
PS C:\Foo> # 2. Getting the VM configuration location
PS C:\Foo> (Get-VM -Name PSDirect).ConfigurationLocation
C:\Vm\VMs\PSDirect
```

Figure 12.48: Checking the status of the PSDirect VM

In *step 3*, you view the locations that Hyper-V is using to store the virtual hard disks for the `PSDirect` VM, with output like this:

```
PS C:\Foo> # 3. Getting the virtual hard drive locations
PS C:\Foo> Get-VMHardDiskDrive -VMName PSDirect |
             Format-Table -Property VMName, ControllerType, Path

VMName   ControllerType Path
------   -------------- ----
PSDirect            IDE C:\Vm\Vhds\PSDirect.Vhdx
PSDirect           SCSI C:\Vm\Vhds\PSDirect-D.VHDX
```

Figure 12.49: Viewing PSDirect's virtual hard drives

In *step 4*, you move the storage of the running `PSDirect` VM. This step generates no output. After moving the VM's storage, in *step 5*, you re-view the VM configuration location and the details of the hard drives for the `PSDirect` VM. The output of this step looks like this:

```
PS C:\Foo> # 5. Viewing the configuration details after moving the VM's storage
PS C:\Foo> (Get-VM -Name PSDirect).ConfigurationLocation
C:\PSDirectNew
PS C:\Foo> Get-VMHardDiskDrive -VMName PSDirect |
             Format-Table -Property VMName, ControllerType, Path

VMName   ControllerType Path
------   -------------- ----
PSDirect            IDE C:\PSDirectNew\Virtual Hard Disks\PSDirect.Vhdx
PSDirect           SCSI C:\PSDirectNew\Virtual Hard Disks\PSDirect-D.VHDX
```

Figure 12.50: Viewing PSDirect's virtual hard drives

In *step 6*, you check to see which VMs are available on `HV2`, with output like this:

```
PS C:\Foo> # 6. Getting the VM details for VMs from HV2
PS C:\Foo> Get-VM -ComputerName HV2

Name     State CPUUsage(%) MemoryAssigned(M) Uptime   Status             Version
----     ----- ----------- ----------------- ------   ------             -------
SQLAcct1 Off   0           0                 00:00:00 Operating normally 10.0
SQLAcct2 Off   0           0                 00:00:00 Operating normally 10.0
SQLAcct3 Off   0           0                 00:00:00 Operating normally 10.0
SQLMfg1  Off   0           0                 00:00:00 Operating normally 10.0
SQLMfg2  Off   0           0                 00:00:00 Operating normally 10.0
```

Figure 12.51: Viewing VMs on HV2

In *step 7*, you create a new `External` switch on `HV2` to enable networking on `HV2`. The output you see is as follows:

```
PS C:\Foo> # 7. Creating External virtual switch on HV2
PS C:\Foo> $SB = {
             $NSHT = @{
             Name              = 'External'
             NetAdapterName    = 'Ethernet'
             AllowManagementOS = $true
           }
           New-VMSwitch @NSHT
           }
PS C:\Foo> Invoke-Command -ScriptBlock $SB -ComputerName HV2

WARNING: The network connection to HV2 has been interrupted. Attempting to reconnect for up to 4 minutes...
WARNING: Attempting to reconnect to HV2 ...
WARNING: The network connection to HV2 has been restored.

Name     SwitchType NetAdapterInterfaceDescription    PSComputerName
----     ---------- ------------------------------    --------------
External External   Microsoft Hyper-V Network Adapter HV2
```

Figure 12.52: Creating the External virtual switch on HV2

In *step 8*, you enable VM migration on both `HV1` and `HV2`. In *step 9*, you configure Hyper-V VM migration on both servers. Next, in *step 10*, you perform a live migration of the `PSDirect` VM to `HV2`. These three steps produce no output.

In *step 11*, you display how long it took Hyper-V to migrate the `PSDirect` VM, with output like this:

```
PS C:\Foo> # 11. Displaying the time taken to migrate
PS C:\Foo> $OS = "Migration took: [{0:n2}] minutes"
PS C:\Foo> ($OS -f ($($Finish-$Start).TotalMinutes))
Migration took: [2.89] minutes
```

Figure 12.53: Displaying the VM migration time

In *step 12*, you use the `Get-VM` cmdlet to view the VMs on `HV1`. Since you migrated the `PSDirect` VM to `HV2`, there are no VMs on `HV2`. Thus, there is no console output from this step. Then, in *step 13*, you view the VMs on `HV2`, which produces output like this:

```
PS C:\Foo> # 13. Checking the VMs on HV2
PS C:\Foo> Get-VM -ComputerName HV2

Name     State   CPUUsage(%) MemoryAssigned(M) Uptime           Status             Version
----     -----   ----------- ----------------- ------           ------             -------
PSDirect Running 1           1024              00:02:45.8910000 Operating normally 10.0
SQLAcct1 Off     0           0                 00:00:00         Operating normally 10.0
SQLAcct2 Off     0           0                 00:00:00         Operating normally 10.0
SQLAcct3 Off     0           0                 00:00:00         Operating normally 10.0
SQLMfg1  Off     0           0                 00:00:00         Operating normally 10.0
SQLMfg2  Off     0           0                 00:00:00         Operating normally 10.0
```

Figure 12.54: Displaying VMs on HV2

In *step 14*, you examine the details of where Hyper-V has stored the VM configuration and virtual hard drives for the `PSDirect` VM. The output of this step looks like this:

```
PS C:\Foo> # 14. Looking at the details of the PSDirect VM on HV2
PS C:\Foo> ((Get-VM -Name PSDirect -Computer HV2).ConfigurationLocation)
C:\PSDirect
PS C:\Foo> Get-VMHardDiskDrive -VMName PSDirect -Computer HV2  |
             Format-Table -Property VMName, Path

VMName   Path
------   ----
PSDirect C:\PSDirect\Virtual Hard Disks\PSDirect.Vhdx
PSDirect C:\PSDirect\Virtual Hard Disks\PSDirect-D.VHDX
```

Figure 12.55: Viewing PSDirect VM details on HV2

In *step 15* (which you run from HV2), you migrate the PSDirect VM back to HV1, producing no console output. In the final step in this recipe, *step 16*, you view how long it took Hyper-V to migrate the VM back to HV1, with output like this:

```
PS C:\Foo> # 16. Displaying the time taken to migrate back to HV1
PS C:\Foo> $OS = "Migration back to HV1 took: [{0:n2}] minutes"
PS C:\Foo> ($OS -f ($($Finish-$Start).TotalMinutes))
Migration back to HV1 took: [2.76] minutes
```

Figure 12.56: Viewing PSDirect migration time

There's more...

In *step 1*, you check the status of the PSDirect VM to ensure it is running. If the output you see shows that the VM is not running, you should use the Start-VM cmdlet to start it before proceeding with this recipe.

In *step 6*, you used Get-VM to view the VMs defined on HV2. The output shows the VMs created in the earlier recipe, *Using Hyper-V VM groups*.

In *step 13*, you view the VMs on HV2, which now shows the PSDirect VM. Note that the VM continues to be running since you performed a live migration. Consider opening a remote desktop connection using mstsc.exe, or use the VM Connect feature of the Hyper-V management console and open a session to the PSDirect VM. With either method, you can use and view the VM as you carry out the migration. When the migration is complete, you may need to log in to your VM session again.

Managing VM replication

VM replication is a disaster recovery feature within Hyper-V. Hyper-V creates a VM replica on a remote Hyper-V server and keeps that replica up to date as the original VM changes. The VM on the remote host is not active during the normal operation of the original VM. You can make the replica active should the VM's host fail for some reason.

With Hyper-V replication, the source VM host bundles up the changes in a running VM's VHD file(s) and regularly sends them to the replica server. The replica server then applies those changes to the dormant replica.

Once you have a VM replica established, you can test the replica to ensure that it can start should you need it. Also, you can failover to the replica, bringing the replicated VM up based on the most recently replicated data. If the source VM host becomes inoperable before it can replicate changes on the source VM, there is a risk of the replication process losing those changes.

In this recipe, you create and use a replica of the PSDirect VM, which you have used in previous recipes in this chapter.

Getting ready

This recipe creates a Hyper-V VM replica of the PSDirect VM running in HV1 to HV2. You created this VM in the *Creating a Hyper-V VM* recipe. You also use this VM in the *Managing VM and storage movement* recipe. You should have loaded the Hyper-V management tools on DC1. If you have not already done so, do the following on DC1:

```
Install-WindowsFeature -Name RSAT-Hyper-V-Tools,
                       Hyper-V-PowerShell        |
  Out-Null
```

Additionally, remember that these recipes assume you have turned off any firewalls on your VMs (and host). This simplifies the environment and enables the recipe to focus on, in this case, VM replication.

How to do it...

1. Configuring HV1 and HV2 to be trusted for delegation in AD on DC1

```
$SB = {
  Set-ADComputer -Identity HV1 -TrustedForDelegation $True
  Set-ADComputer -Identity HV2 -TrustedForDelegation $True
}
Invoke-Command -ComputerName DC1 -ScriptBlock $SB
```

2. Rebooting the HV1 and HV2 hosts

```
Restart-Computer -ComputerName HV2 -Force
Restart-Computer -ComputerName HV1 -Force
```

3. Configuring Hyper-V replication on HV1 and HV2

```
$VMRHT = @{
  ReplicationEnabled              = $true
  AllowedAuthenticationType       = 'Kerberos'
  KerberosAuthenticationPort      = 42000
  DefaultStorageLocation          = 'C:\Replicas'
  ReplicationAllowedFromAnyServer = $true
  ComputerName                    = 'HV1.Reskit.Org',
                                    'HV2.Reskit.Org'
}
Set-VMReplicationServer @VMRHT
```

4. Enabling PSDirect on HV1 to be a replica source

```
$VMRHT = @{
  VMName             = 'PSDirect'
  Computer           = 'HV1'
  ReplicaServerName  = 'HV2'
  ReplicaServerPort  = 42000
  AuthenticationType = 'Kerberos'
  CompressionEnabled = $true
  RecoveryHistory    = 5
}
Enable-VMReplication  @VMRHT
```

5. Viewing the replication status of HV1

```
Get-VMReplicationServer -ComputerName HV1
```

6. Checking PSDirect on Hyper-V hosts

```
Get-VM -ComputerName HV1 -VMName PSDirect
Get-VM -ComputerName HV2 -VMName PSDirect
```

7. Starting the initial replication

```
Start-VMInitialReplication -VMName PSDirect -ComputerName HV1
```

8. Examining the initial replication state on HV1 just after you start the initial replication

```
Measure-VMReplication -ComputerName HV1
```

9. Examining the replication status on HV1 after replication completes

```
Measure-VMReplication -ComputerName HV1
```

10. Testing PSDirect failover to HV2

```
$SB = {
  $VM = Start-VMFailover -AsTest -VMName PSDirect -Confirm:$false
  Start-VM $VM
}
Invoke-Command -ComputerName HV2 -ScriptBlock $SB
```

11. Viewing the status of PSDirect VMs on HV2

```
Get-VM -ComputerName HV2 -VMName PSDirect*
```

12. Stopping the failover test

```
$SB = {
  Stop-VMFailover -VMName PSDirect
}
Invoke-Command -ComputerName HV2 -ScriptBlock $SB
```

13. Viewing the status of VMs on HV1 and HV2 after failover stopped

```
Get-VM -ComputerName HV1
Get-VM -ComputerName HV2
```

14. Stopping PSDirect on HV1 before performing a planned failover

```
Stop-VM PSDirect -ComputerName HV1
```

15. Starting VM failover from HV1 to HV2

```
Start-VMFailover -VMName PSDirect -ComputerName HV2 -Confirm:$false
```

16. Completing the failover

```
$CHT = @{
  VMName       = 'PSDirect'
  ComputerName = 'HV2'
  Confirm      = $false
}
Complete-VMFailover @CHT
```

17. Starting the replicated VM on HV2

```
Start-VM -VMname PSDirect -ComputerName HV2
Set-VMReplication -VMname PSDirect -reverse -ComputerName HV2
```

18. Checking the PSDirect VM on HV1 and HV2 after the planned failover

```
Get-VM -ComputerName HV1 -Name PSDirect
Get-VM -ComputerName HV2 -Name PSDirect
```

19. Removing the VM replication on HV2

```
Remove-VMReplication -VMName PSDirect -ComputerName HV2
```

20. Removing the PSDirect VM replica on HV1

```
Remove-VM -Name PSDirect -ComputerName HV1 -Confirm:$false -Force
```

Run the remainder of this recipe on HV2.

21. Moving the PSDirect VM back to HV1

```
$VMHT2 = @{
    Name                  = 'PSDirect'
    ComputerName           = 'HV2'
    DestinationHost        = 'HV1'
    IncludeStorage         =  $true
```

```
        DestinationStoragePath = 'C:\VM\VHDs\PSDirect' # on HV1
    }
    Move-VM @VMHT2
```

How it works...

In *step 1*, you configure `HV1` and `HV2` to be trusted for delegation as required to support VM replication, and then in *step 2*, you reboot both hosts. These two steps produce no console output.

After rebooting both hosts, in *step 3*, you configure both Hyper-V hosts to support VM replication. In *step 4*, you set up a Hyper-V replication partnership between `HV1` and `HV2`. These two steps create no output.

In *step 5*, you review the replication status of the `HV1` Hyper-V server, which creates output like this:

```
PS C:\Foo> # 5. Viewing the replication status of HV1
PS C:\Foo> Get-VMReplicationServer -ComputerName HV1

RepEnabled AuthType KerbAuthPort CertAuthPort AllowAnyServer
---------- -------- ------------ ------------ ---------------
True       Kerb     42000        443          True
```

Figure 12.57: Viewing the replication status of HV1

Next, in *step 6*, you check the replication status of the `PSDirect` VM, which you do on both Hyper-V servers. The output of this step looks like this:

```
PS C:\Foo> # 6. Checking PSDirect on Hyper-V hosts
PS C:\Foo> Get-VM -ComputerName HV1 -VMName PSDirect

Name     State   CPUUsage(%) MemoryAssigned(M) Uptime           Status             Version
----     -----   ----------- ----------------- ------           ------             -------
PSDirect Running 0           2048              00:03:41.3630000 Operating normally 10.0

PS C:\Foo> Get-VM -ComputerName HV2 -VMName PSDirect

Name     State CPUUsage(%) MemoryAssigned(M) Uptime   Status             Version
----     ----- ----------- ----------------- ------   ------             -------
PSDirect Off   0           0                 00:00:00 Operating normally 10.0
```

Figure 12.58: Checking the PSDirect VM status on HV1 and HV2

In *step 7*, you start the VM replication process. This step involves Hyper-V creating a duplicate VM on `HV2`, essentially a full copy of the `PSDirect` VM, producing no output.

If you run *step 8* immediately after completing *step 7*, you can observe the initial replication process. You should see output like this:

```
PS C:\Foo> # 8. Examining the initial replication state on HV1 just after
PS C:\Foo> #    you start the initial replication
PS C:\Foo> Measure-VMReplication -ComputerName HV1

VMName   State                      Health LReplTime PReplSize(M) AvgLatency AvgReplSize(M) Relationship
------   -----                      ------ --------- ------------ ---------- -------------- ------------
PSDirect InitialReplicationInProgress Normal          8,420.61               0.00          Simple
```

Figure 12.59: Viewing the replication process

The initial replication process should take a few minutes. However, the replication speed also depends on the Hyper-V host hardware components, VM memory and virtual processors, and the underlying network and disk subsystem speeds. The disk sizes of any virtual disks are also a significant factor in the initial replication speed. After Hyper-V completes the initial replication of `PSDirect`, you can view the replication status, as shown in *step 9*. The output of this step looks like this:

```
PS C:\Foo> # 9. Examining the replication status on HV1 after replication completes
PS C:\Foo> Measure-VMReplication -ComputerName HV1

VMName   State       Health LReplTime           PReplSize(M) AvgLatency AvgReplSize(M) Relationship
------   -----       ------ ---------           ------------ ---------- -------------- ------------
PSDirect Replicating Normal 01/04/2021 00:39:20 0.0078       00:03:04   1,896.00       Simple
```

Figure 12.60: Viewing the replication process once initial replication is complete

Now that you have the `PSDirect` replication invoked and Hyper-V has completed an initial replication, you can test the failover capability. In *step 10*, you run a test failover of the `PSDirect` VM from `HV1` to `HV2`. This step generates no console output.

In *step 11*, you view the status of the `PSDirect` VMs on `HV2`, with output like this:

```
PS C:\Foo> # 11. Viewing the status of PSDirect VMs on HV2
PS C:\Foo> Get-VM -ComputerName HV2 -VMName PSDirect*

Name            State   CPUUsage(%) MemoryAssigned(M) Uptime           Status             Version
----            -----   ----------- ----------------- ------           ------             -------
PSDirect - Test Running 0           1024              00:01:29.6580000 Operating normally 10.0
PSDirect        Off     0           0                 00:00:00         Operating normally 10.0
```

Figure 12.61: Viewing PSDirect VMs on HV2

In *step 12*, you stop the failover test, which generates no output. In *step 13*, you view the status of VMs on HV1 and HV2 after failover has stopped, which produces output like this:

```
PS C:\Foo> # 13. Viewing the status of VMs on HV1 and HV2 after failover stopped
PS C:\Foo> Get-VM -ComputerName HV1 -Name PSDDirect

Name     State   CPUUsage(%) MemoryAssigned(M) Uptime           Status             Version
----     -----   ----------- ----------------- ------           ------             -------
PSDirect Running 1           1024              01:39:17.6950000 Operating normally 10.0

PS C:\Foo> Get-VM -ComputerName HV2 -Name PSDDirect

Name     State CPUUsage(%) MemoryAssigned(M) Uptime   Status             Version
----     ----- ----------- ----------------- ------   ------             -------
PSDirect Off   0           0                 00:00:00 Operating normally 10.0
```

Figure 12.62: Viewing the status of PSDirect VMs on HV1 and HV2 after failover has stopped

In *step 14*, you stop the `PSDirect` VM before carrying out a planned failover. Next, in *step 15*, you perform the planned failover of `PSDirect` from `HV1` to `HV2`. Once the planned failover has completed (that is, once the cmdlet completes), in *step 16*, you complete the failover process. In *step 17*, you start the `PSDirect` VM on `HV2`. These four steps produce no console output.

In *step 18*, you check the status of the `PSDirect` VM on both Hyper-V hosts. This step generates the following output:

```
PS C:\Foo> # 18. Checking the PSDirect VM on HV1 and HV2 after the planned failover
PS C:\Foo> Get-VM -ComputerName HV1 -Name PSDirect

Name     State CPUUsage(%) MemoryAssigned(M) Uptime   Status             Version
----     ----- ----------- ----------------- ------   ------             -------
PSDirect Off   0           0                 00:00:00 Operating normally 10.0

PS C:\Foo> Get-VM -ComputerName HV2 -Name PSDirect

Name     State   CPUUsage(%) MemoryAssigned(M) Uptime           Status             Version
----     -----   ----------- ----------------- ------           ------             -------
PSDirect Running 1           2048              00:01:37.0010000 Operating normally 10.0
```

Figure 12.63: Checking the status of PSDirect VMs on HV1 and HV2 after the planned failover

In *step 19*, you remove the replication of the `PSDirect` VM from `HV2`. This step leaves the `PSDirect` VM running on `HV1`, although no longer replicating to another server. In *step 20*, you remove the `PSDirect` VM from `HV1`. Finally, in *step 21*, you move the `PSDirect` VM back to `HV1` to complete this recipe. These last three steps generate no console output.

There's more...

In *step 11*, you view the two `PSDirect` VMs on `HV2`. The first is the VM that you are replicating from `HV1`. During this step, `PSDirect` is up and running on `HV1` and replicating any changes to `HV2`. You see a separate test VM showing that the replica VM is up and running on `HV2`. It is always good to run a test failover after setting up replication to ensure that the replication and failover work as you wish.

In *step 18*, you view the `PSDirect` VM status on both your Hyper-V hosts. As you can see in the output from this step, *Figure 12.63*, the `PSDirect` VM is up and running on `HV2`. On `HV1`, you still have an (older) copy of the VM. With the VM failed over (and running) on `HV2`, you can now reverse the replication (that is, begin replication back to `HV1` from `HV2` for this VM). Or, as you do here, you can remove the VM from `HV1`, which allows you to move this VM back to `HV1` as you do in *step 21*, which completes this recipe.

Managing VM checkpoints

With Hyper-V in Server 2022, a checkpoint captures the state of a VM into a restore point. Hyper-V then enables you to roll back a VM to a checkpoint. Windows Server 2008's version of Hyper-V provided this feature. With Server 2008, these restore points were called snapshots.

With Server 2012, Microsoft changed the name to **checkpoint**. This change of terminology was then consistent with System Center and avoided confusion with respect to the **Volume Shadow Copy Service** (**VSS**) snapshots used by many backup systems. While the Hyper-V team did change the terminology, some of the cmdlet names remain unchanged. For instance, to restore a VM to a checkpoint, you use the `Restore-VMSnapshot` cmdlet.

When you create a checkpoint, Hyper-V temporarily pauses the VM. Hyper-V creates a new differencing disk (AVHD). Hyper-V then resumes the VM, which writes all data to the differencing disk. You can create a variety of checkpoints for a VM.

Checkpoints are excellent for a variety of scenarios. They can be helpful in troubleshooting. You can get the VM to the point where some bug is triggered and take a checkpoint. Then you can try a fix—if it doesn't work, you can just roll the VM back to the checkpoint and try some other fix. Checkpoints are also helpful for training. You could create a VM in which you perform all the lab exercise steps and create a checkpoint after each successful lab. That way, the student can make a mistake in a lab exercise and skip forward to a later checkpoint to carry on.

Using checkpoints in production is a different matter. In general, you should avoid using checkpoints on your production systems for several reasons. If your servers use any type of replication or transaction-based applications, the impact of rolling back a VM to an earlier time can lead to issues. Since checkpoints rely on differencing disks that feature constantly growing physical disk files, using checkpoints can also result in poor performance.

In this recipe, you create a checkpoint of the PSDirect VM, and then you create a file inside the VM. You take a further checkpoint and create a second file. Then you revert to the first checkpoint, observing that there are no files created. You roll forward to the second checkpoint to see that the first file is there but not the second (because you created the second file after taking the checkpoint). Then you remove all the checkpoints. After each checkpoint operation, you observe the VHDX and AVHD files that Hyper-V uses in the PSDirect VM.

Getting ready

You run this recipe on HV1. This recipe makes use of the PSDirect VM that you created and used earlier in this chapter. Depending on which other recipes you have run from this chapter, the virtual disks may be in different folders, but the recipe copes with the disk files in any folder known to Hyper-V.

How to do it...

1. Creating credentials for PSDirect

```
$RKAn = 'Wolf\Administrator'
$PS   = 'Pa$$w0rd'
$RKP  = ConvertTo-SecureString -String $PS -AsPlainText -Force
$T = 'System.Management.Automation.PSCredential'
$RKCred = New-Object -TypeName $T -ArgumentList $RKAn,$RKP
```

2. Examining the C:\ drive in the PSDirect VM before we start

```
$SB = { Get-ChildItem -Path C:\ | Format-Table}
$ICHT = @{
  VMName      = 'PSDirect'
  ScriptBlock = $SB
  Credential  = $RKCred
}
Invoke-Command @ICHT
```

3. Creating a checkpoint of PSDirect on HV1

```
$CPHT = @{
  VMName       = 'PSDirect'
  ComputerName = 'HV1'
  SnapshotName = 'Snapshot1'
}
Checkpoint-VM @CPHT
```

4. Examining the files created to support the checkpoints

```
$Parent = Split-Path -Parent (Get-VM -Name PSdirect |
            Select-Object -ExpandProperty HardDrives).Path |
              Select-Object -First 1
Get-ChildItem -Path $Parent
```

5. Creating some content in a file on `PSDirect` and displaying it

```
$SB = {
  $FileName1 = 'C:\File_After_Checkpoint_1'
  Get-Date | Out-File -FilePath $FileName1
  Get-Content -Path $FileName1
}
$ICHT = @{
  VMName      = 'PSDirect'
  ScriptBlock = $SB
  Credential  = $RKCred
}
Invoke-Command @ICHT
```

6. Taking a second checkpoint

```
$SNHT = @{
  VMName       = 'PSDirect'
  ComputerName = 'HV1'
  SnapshotName = 'Snapshot2'
}
Checkpoint-VM @SNHT
```

7. Viewing the VM checkpoint details for `PSDirect`

```
Get-VMSnapshot -VMName PSDirect
```

8. Looking at the files supporting the two checkpoints

```
Get-ChildItem -Path $Parent
```

9. Creating and displaying another file in `PSDirect`
(i.e. after you have taken `Snapshot2`)

```
$SB = {
  $FileName2 = 'C:\File_After_Checkpoint_2'
  Get-Date | Out-File -FilePath $FileName2
  Get-ChildItem -Path C:\ -File | Format-Table
}
$ICHT = @{
  VMName    = 'PSDirect'
  ScriptBlock = $SB
  Credential  = $RKCred
```

```
}
Invoke-Command @ICHT
```

10. Restoring the `PSDirect` VM back to the checkpoint named `Snapshot1`

```
$Snap1 = Get-VMSnapshot -VMName PSDirect -Name Snapshot1
Restore-VMSnapshot -VMSnapshot $Snap1 -Confirm:$false
Start-VM -Name PSDirect
Wait-VM -For IPAddress -Name PSDirect
```

11. Seeing what files we have now on `PSDirect`

```
$SB = {
  Get-ChildItem -Path C:\ | Format-Table
}
$ICHT = @{
  VMName    = 'PSDirect'
  ScriptBlock = $SB
  Credential  = $RKCred
}
Invoke-Command @ICHT
```

12. Rolling forward to `Snapshot2`

```
$Snap2 = Get-VMSnapshot -VMName PSdirect -Name Snapshot2
Restore-VMSnapshot -VMSnapshot $Snap2 -Confirm:$false
Start-VM -Name PSDirect
Wait-VM -For IPAddress -Name PSDirect
```

13. Observing the files you now have supporting `PSDirect`

```
$SB = {
    Get-ChildItem -Path C:\ | Format-Table
}
$ICHT = @{
  VMName      = 'PSDirect'
  ScriptBlock = $SB
  Credential  = $RKCred
}
Invoke-Command @ICHT
```

14. Restoring to `Snapshot1` again

```
$Snap1 = Get-VMSnapshot -VMName PSDirect -Name Snapshot1
Restore-VMSnapshot -VMSnapshot $Snap1 -Confirm:$false
Start-VM -Name PSDirect
Wait-VM -For IPAddress -Name PSDirect
```

15. Checking checkpoints and VM data files again

```
Get-VMSnapshot -VMName PSDirect
Get-ChildItem -Path $Parent | Format-Table
```

16. Removing all the checkpoints from HV1

```
Get-VMSnapshot -VMName PSDirect |
  Remove-VMSnapshot
```

17. Checking VM data files again

```
Get-ChildItem -Path $Parent
```

How it works...

In *step 1*, you create a Windows credential object to use with the `PSDirect` VM. This step creates no output.

In *step 2*, you examine the `C:\` drive inside the `PSDirect` VM, generating output like this:

```
PS C:\Foo> # 2. Examining the C:\ in the PSDirect VM before we start
PS C:\Foo> $SB = { Get-ChildItem -Path C:\ | Format-Table}
PS C:\Foo> $ICHT = @{
             VMName      = 'PSDirect'
             ScriptBlock = $SB
             Credential  = $RKCred
           }
PS C:\Foo> Invoke-Command @ICHT

    Directory: C:\

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d-----       15/09/2018     08:19            PerfLogs
d-r---       29/03/2021     17:52            Program Files
d-----       25/03/2021     18:13            Program Files (x86)
d-r---       25/03/2021     18:13            Users
d-----       28/03/2021     12:53            Windows
```

Figure 12.64: Examining C:\ on the PSDirect VM

In *step 3*, you create a VM checkpoint of the `PSDirect` VM, which generates no console output. After you create the VM checkpoint, in *step 4*, you examine the files that Hyper-V uses to support it. The output of this step looks like this:

```
PS C:\Foo> # 4. Examining the files created to support the checkpoints
PS C:\Foo> $Parent = Split-Path -Parent (Get-VM -Name PSdirect |
                Select-Object -ExpandProperty HardDrives).Path |
                  Select-Obhject -First 1
PS C:\Foo> Get-ChildItem -Path $Parent

    Directory: C:\Vm\Vhds\PSDirect\Virtual Hard Disks

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a---          02/04/2021    12:16       71303168 PSDirect_FE7F7810-A143-4593-892A-E4CDD061925A.avhdx
-a---          02/04/2021    12:15        4194304 PSDirect-D_79D1575C-19DC-408B-B95D-59DDDE432715.avhdx
-a---          01/04/2021    15:53        4194304 PSDirect-D.VHDX
-a---          02/04/2021    12:15    10238296064 PSDirect.Vhdx
```

Figure 12.65: Examining files supporting a checkpoint

In *step 5*, you create a new file on the `C:\` drive in the `PSDirect` VM. Then you view the contents of this file. The output looks like this:

```
PS C:\Foo> # 5. Creating some content in a file on PSDirect and displaying it
PS C:\Foo> $SB = {
             $FileName1 = 'C:\File_After_Checkpoint_1'
             Get-Content -Path $FileName1
           }
PS C:\Foo> $ICHT = @{
             VMName      = 'PSDirect'
             ScriptBlock = $SB
             Credential  = $RKCred
           }
PS C:\Foo> Invoke-Command @ICHT

02 April 2021 12:17:31
```

Figure 12.66: Creating a file in the PSDirect VM

After creating a file inside the VM, in *step 6*, you create a second checkpoint that generates no output. In *step 7*, you use the `Get-VMSnapshot` cmdlet to view the checkpoint for the `PSDirect` VM currently on `HV1`, which looks like this:

```
PS C:\Foo> # 7. Viewing the VM checkpoint details for PSDirect
PS C:\Foo> Get-VMSnapshot -VMName PSDirect

VMName   Name      SnapshotType CreationTime        ParentSnapshotName
------   ----      ------------ ------------        ------------------
PSDirect Snapshot1 Standard     02/04/2021 12:15:59
PSDirect Snapshot2 Standard     02/04/2021 12:18:53 Snapshot1
```

Figure 12.67: Viewing snapshots of PSDirect on HV1

In *step 8*, you examine the files that Hyper-V uses to store your virtual disk images and checkpoint files for the `PSDirect` VM, like this:

```
PS C:\Foo> # 8. Looking at the files supporting the two checkpoints
PS C:\Foo> Get-ChildItem -Path $Parent

    Directory: C:\Vm\Vhds\PSDirect\Virtual Hard Disks

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a---          02/04/2021    12:19       71303168 PSDirect_F4935A8F-2CE2-42C9-A198-5DC2C47F63D3.avhdx
-a---          02/04/2021    12:18      168820736 PSDirect_FE7F7810-A143-4593-892A-E4CDD061925A.avhdx
-a---          02/04/2021    12:18        4194304 PSDirect-D_79D1575C-19DC-408B-B95D-59DDDE432715.avhdx
-a---          02/04/2021    12:18        4194304 PSDirect-D_8239F1DE-ACB8-4940-9FE2-5534FA154415.avhdx
-a---          01/04/2021    15:53        4194304 PSDirect-D.VHDX
-a---          02/04/2021    12:15    10238296064 PSDirect.Vhdx
```

Figure 12.68: Viewing the checkpoints of PSDirect on HV1

In *step 9*, you create a new file in the `PSDirect` VM, add contents to the file, and then view the files created so far, like this:

```
PS C:\Foo> # 9. Creating and displaying another file in PSDirect
PS C:\Foo> #    (i.e. after you have taken Snapshot2)
PS C:\Foo> $SB = {
             $FileName2 = 'C:\File_After_Checkpoint_2'
             Get-Date | Out-File -FilePath $FileName2
             Get-ChildItem -Path C:\ -File | Format-Table
           }
PS C:\Foo> $ICHT = @{
             VMName     = 'PSDirect'
             ScriptBlock = $SB
             Credential  = $RKCred
           }
PS C:\Foo> Invoke-Command @ICHT

    Directory: C:\

Mode           LastWriteTime   Length Name
----           -------------   ------ ----
-a----  02/04/2021     12:17       62 File_After_Checkpoint_1
-a----  02/04/2021     12:21       62 File_After_Checkpoint_2
```

Figure 12.69: Creating a second file in PSDirect

In *step 10*, you revert the `PSDirect` VM back to the first checkpoint, which creates no output to the console. In *step 11*, you see what files you now have on the `C:\` drive, like this:

```
PS C:\Foo> # 11. Seeing what files we have now on PSDirect
PS C:\Foo> $SB = {
             Get-ChildItem -Path C:\ | Format-Table
           }
PS C:\Foo> $ICHT = @{
             VMName     = 'PSDirect'
             ScriptBlock = $SB
             Credential  = $RKCred
           }
PS C:\Foo> Invoke-Command @ICHT

    Directory: C:\

Mode             LastWriteTime     Length Name
----             -------------     ------ ----
d-----    15/09/2018     08:19            PerfLogs
d-r---    29/03/2021     17:52            Program Files
d-----    25/03/2021     18:13            Program Files (x86)
d-r---    25/03/2021     18:13            Users
d-----    28/03/2021     12:53            Windows
```

Figure 12.70: Checking files in PSDirect

In *step 12*, you restore the `PSDirect` VM to the second checkpoint, which creates no output. In *step 13*, you examine the files created in `PSDirect`, which looks like this:

```
PS C:\Foo> # 13. Observe the files you now have supporting PSDirect
PS C:\Foo> $SB = {
             Get-ChildItem -Path C:\ | Format-Table
           }
PS C:\Foo> $ICHT = @{
             VMName      = 'PSDirect'
             ScriptBlock = $SB
             Credential  = $RKCred
           }
PS C:\Foo> Invoke-Command @ICHT

    Directory: C:\

Mode             LastWriteTime     Length Name
----             -------------     ------ ----
d-----    15/09/2018     08:19            PerfLogs
d-r---    29/03/2021     17:52            Program Files
d-----    25/03/2021     18:13            Program Files (x86)
d-r---    25/03/2021     18:13            Users
d-----    28/03/2021     12:53            Windows
-a----    02/04/2021     12:17         62 File_After_Checkpoint_1
```

Figure 12.71: Checking files in PSDirect after restoring Snapshot2

In *step 14*, you restore `PSDirect` to the first checkpoint, which creates no output. In *step 15*, you check the checkpoints and the virtual disk files again, with output like this:

```
PS C:\Foo> # 15. Checking checkpoints and VM data files again
PS C:\Foo> Get-VMSnapshot -VMName PSDirect

VMName   Name      SnapshotType CreationTime        ParentSnapshotName
------   ----      ------------ ------------        ------------------
PSDirect Snapshot1 Standard     02/04/2021 12:15:59
PSDirect Snapshot2 Standard     02/04/2021 12:18:53 Snapshot1

PS C:\Foo> Get-ChildItem -Path $Parent | Format-Table

    Directory: C:\VM\VHDS\PSDirect\Virtual Hard Disks

Mode        LastWriteTime      Length Name
----        -------------      ------ ----
-a--- 02/04/2021    12:32   541065216 PSDirect_E21807D2-F3EB-4FA6-9F67-8F780AEF577A.avhdx
-a--- 02/04/2021    12:18   168820736 PSDirect_FE7F7810-A143-4593-892A-E4CDD061925A.avhdx
-a--- 02/04/2021    12:28     4194304 PSDirect-D_60959C86-0576-4BCC-9699-931E5CC0525C.avhdx
-a--- 02/04/2021    12:18     4194304 PSDirect-D_79D1575C-19DC-408B-B95D-59DDDE432715.avhdx
-a--- 01/04/2021    15:53     4194304 PSDirect-D.VHDX
-a--- 02/04/2021    12:15 10238296064 PSDirect.Vhdx
```

Figure 12.72: Checking the checkpoints on HV1

In *step 16*, you remove all the checkpoints for `PSDirect` on `HV1`, which generates no output. In the final step in this recipe, you check the VM data files on `HV1`, which produces the following output:

```
PS C:\Foo> # 17. Checking VM data files again
PS C:\Foo> Get-ChildItem -Path $Parent

    Directory: C:\Vm\Vhds\PSDirect\Virtual Hard Disks

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a---          02/04/2021    12:33        4194304 PSDirect-D.VHDX
-a---          02/04/2021    12:33    10271850496 PSDirect.Vhdx
```

Figure 12.73: Viewing disk files after removing the checkpoints

There's more...

In *step 8*, you examine the files that Hyper-V uses to support the `PSDirect` VM checkpoints. The two VHDX files hold the starting state when you took the first checkpoint. Then, there are two more sets of AVHDX files. Each file represents any work you have done after taking a checkpoint and before taking another.

In *step 9*, you view the two files you created in the `PSDirect` VM. After reverting to the first checkpoint, in *step 11*, you see that neither of the two files you created exists on the `C:\` drive. This is because you created those files after creating the first checkpoint. This situation is the expected result of restoring a checkpoint.

13
Managing Azure

In this chapter, we cover the following recipes:

- Getting started using Azure with PowerShell
- Creating Azure resources
- Exploring the Azure storage account
- Creating an Azure SMB file share
- Creating an Azure website
- Creating an Azure Virtual Machine

Introduction

Azure is Microsoft's cloud computing platform and is a competitor to **Amazon Web Services** (**AWS**) and other public cloud providers. Azure provides you with access to a vast and constantly growing range of features. It enables any organization to move some of, most of, or even their entire on-premises infrastructure into the cloud.

Azure features come in three levels:

- **Infrastructure as a Service** (**IaaS**)
- **Platform as a Service** (**PaaS**)
- **Software as a Service** (**SaaS**)

IaaS is, in effect, an instant computing infrastructure that you can provision, manage, and use over the Internet or via a private network connection. IaaS includes essential computing infrastructure components (servers, storage, networking, firewalls, and security), plus the physical plant you require to run these components (for example, power and air conditioning). In an IaaS environment, the servers are Azure **Virtual Machines** (**VMs**) (effectively Hyper-V VMs) and interact with the networking, security, and storage components.

PaaS is a complete deployment environment in the cloud, including the OS, storage, and other infrastructure. One key PaaS offering in Azure is Azure SQL Database. Things like the OS and SQL Server patching, which you would have to deal with if you deployed SQL in an IaaS environment, are all managed by Azure. The Azure SQL offering provides a (nearly) complete SQL service, all managed by Azure. You can do a few things in a full SQL Server implementation that the SQL PaaS offering does not provide. These are generally actions that only the platform owner is allowed to perform. For example, with SQL running inside an IaaS Azure VM, you cannot use SQL database mirroring. If you need SQL services that the Azure SQL offering does not provide, you can create a VM and install and manage a SQL server in the VM.

With SaaS, you just use an application that the vendor has placed in the cloud. A key example of SaaS is Office 365, which bundles Exchange Online, SharePoint Online, Teams, and more. Strictly speaking, Office 365 is not an Azure offering – you purchase it directly from either the Office 365 website or via a Microsoft Partner. In terms of purchase, Office 365 is a single offering with many different plans (combinations of services that include a downloadable version of the Office applications, such as Word and Excel). Using PowerShell to manage Office 365, each of the included applications has a unique operational approach. With Exchange Online, for example, you use PowerShell implicit remoting to manage the Exchange component of your Office 365 subscription. Other commands run locally but make use of REST API calls to Azure across the Internet.

To provide authentication for software running within Azure and other SaaS applications, you can use **Azure Active Directory** (**AAD**). With AAD, you can create a cloud-only directory or synchronize AAD with your on-premises Active Directory. AAD can also be used to provide authentication for a range of other third-party SaaS applications. Full details on managing AAD and Office 365 are outside the scope of this chapter.

We begin this chapter with the first recipe, *Getting started using Azure with PowerShell*, in which you create an environment that allows you to manage Azure and the Office 365 SaaS components. This recipe also shows you how to download the cmdlets you need.

The *Creating Azure resources* recipe guides you through creating a few of the core resources you need to create and manage other Azure resources. These include a resource group and a storage account. You create all Azure resources within a resource group.

You create and store any required storage, such as VHD files for an Azure VM, in a storage group. While the recipes in this chapter use a single resource group and a single storage account for simplicity, in large-scale Azure deployments, you may require multiple instances of these resources.

In the *Exploring the Azure storage account* recipe, we look at setting up Azure Storage using the Azure storage account we created earlier. The *Creating an Azure SMB file share* recipe shows you how you can create an SMB 3 file share that you can access from client applications across the Internet. Instead of having an application point to an on-premises file share, you can now host the share in Azure. This feature might be useful if you use an Azure IaaS VM to host an application that utilizes a shared folder for its data. You could also use it as a file share in the cloud.

The *Creating an Azure website* recipe shows you how you can set up a simple website. The recipe creates a free Azure app plan that supports an IIS website. With this app plan, you can set up a simple website, say for a short-term marketing campaign. You can scale this to deliver Internet-scale websites that you can have Azure scale dynamically according to load.

The final recipe, *Creating an Azure Virtual Machine*, examines how to create an Azure VM and access it via **Remote Desktop Protocol** (**RDP**). Although Azure uses a Hyper-V variant to run its VMs, you cannot use the Hyper-V cmdlets you saw in *Chapter 11* to manage Azure VMs. Azure has its own set of commands, as you see in this recipe.

This chapter is only a taster of using Azure with PowerShell. There is so much more you can do that could not fit into this chapter.

Getting started using Azure with PowerShell

There are two fundamental things you need to do before you can start managing Azure features using PowerShell. The first is to obtain an Azure subscription. The second is to obtain the cmdlets you use to access Azure and Office 365's features.

Azure is a commercial service – each feature you use potentially has a real-world cost attached, which Microsoft bases on your usage of Azure resources. For example, with an Azure VM, you would pay to have the VM running, and there are additional charges for the VM's virtual disk storage and network traffic in and out of your VM.

The charges for Office 365 and other SaaS offerings, on the other hand, are user-based – a given user can use lots of emails, for example, without incurring any additional charges. For details on costs for Azure, see `https://azure.microsoft.com/pricing/`, and for details on Microsoft 365 charges, see `https://www.microsoft.com/microsoft-365/buy/compare-all-microsoft-365-products`.

There are many ways you can get an Azure subscription, including via a Visual Studio subscription (`https://visualstudio.microsoft.com/vs/pricing/`), an Action Pack subscription (`https://docs.microsoft.com/partner-center/mpn-get-action-pack`), or outright purchase on a pay-as-you-go basis.

Microsoft also provides a one-month free trial subscription that helps you test out the recipes in this chapter. The trial subscription provides you with full access to Azure features up to a financial limit, which is 200 US dollars or similar in other currencies at the time of writing. These limits may have changed by the time you read this book. The trial subscription should be sufficient to enable you to learn how to use PowerShell with Azure.

To get a trial subscription, navigate to `https://azure.microsoft.com/free/`, and fill in the forms. Note that a free trial requires you to submit a credit card number. There is no charge for the subscription; the credit card number is used only for identity verification – and it keeps the lawyers happy.

If you take out an Azure trial and wish to keep your Azure resources running after the trial expires, you have to move it to a pay-as-you-go subscription. You should receive an email shortly before the trial expires to transition it.

To use PowerShell with Azure's various features, you need to obtain cmdlets that Microsoft does not provide in Windows Server 2022/Windows 10, Windows PowerShell 5.0/5.1, or PowerShell 7. You get the relevant modules from the PowerShell Gallery. You use the cmdlets in the `PowerShellGet` module to find and download the necessary modules.

As a word of warning, these cmdlets change regularly. For the most part, these changes add functions and fix bugs, but you may find that new versions of a module bring breaking changes that could affect your scripts. The Azure team provides reasonable notice of breaking changes, and you usually have plenty of notice and flexibility over when you deploy any updates.

Getting ready

You run this recipe on `SRV1`, a domain-joined server.

How to do it...

1. Finding core Azure module on the PS Gallery

```
Find-Module -Name Az |
  Format-Table -Wrap -Autosize
```

2. Installing `Az` module

```
Install-Module -Name Az -Force
```

3. Discovering Azure modules and how many cmdlets each contains

```
$HT = @{ Label ='Cmdlets'
         Expression = {(Get-Command -module $_.name).count}}
Get-Module Az* -ListAvailable |
    Sort-Object {(Get-command -Module $_.Name).Count} -Descending |
       Format-Table -Property Name,Version,Author,$HT -AutoSize
```

4. Finding Azure AD cmdlets

```
Find-Module AzureAD |
  Format-Table -Property Name, Version, Author -AutoSize -Wrap
```

5. Installing the Azure AD module

```
Install-Module -Name AzureAD -Force
```

6. Discovering the Azure AD module

```
$FTHT = @{
    Property = 'Name', 'Version', 'Author', 'Description'
    AutoSize = $true
    Wrap     = $true
}
Get-Module -Name AzureAD -ListAvailable |
  Format-Table @FTHT
```

7. Logging in to Azure

```
$CredAZ  = Get-Credential     # Enter your Azure Credential details
$Account = Connect-AzAccount -Credential $CredAZ
$Account
```

8. Getting Azure account name

```
$AccountN = $Account.Context.Account.Id
"Azure Account   : $AccountN"
```

9. Viewing Azure subscription

```
$SubID = $ACCOUNT.Context.Subscription.Id
Get-AzSubscription -SubscriptionId $SubId |
  Format-List -Property *
```

10. Counting Azure locations

```
$AZL = Get-AzLocation
$LOC = $AZL | Sort-Object Location
"Azure locations:  [{0}]" -f $LOC.Count
```

11. Viewing Azure locations

```
$LOC |
  Format-Table Location, DisplayName
```

12. Getting Azure environments

```
Get-AzEnvironment |
    Format-Table -Property name, ManagementPortalURL
```

How it works...

In *step 1*, you use the `Find-Module` command to find the `Az` module on the PowerShell Gallery. The output of this step should resemble this:

```
PS C:\Foo> #  1. Finding core Azure module on the PS Gallery
PS C:\Foo> Find-Module -Name Az |
             Format-Table -Wrap -Autosize

Version Name Repository Description
------- ---- ---------- -----------
5.7.0   Az   PSGallery  Microsoft Azure PowerShell - Cmdlets to manage resources in Azure. This
                        module is compatible with WindowsPowerShell and PowerShell Core.
                        For more information about the Az module, please visit the
                        following: https://docs.microsoft.com/en-us/powershell/azure/
```

Figure 13.1: Finding the Az module

In *step 2*, you install the `Az` module on `SRV1`. This step installs all of the individual sub-modules that you can use to manage Azure. Although this step produces no console output, you may see pop-up progress indicators as PowerShell installs the individual modules.

In *step 3*, you discover the individual modules and how many cmdlets each one contains. The output looks like this:

```
PS C:\Foo> # 3. Discovering Azure modules and how many cmdlets each contains
PS C:\Foo> $HT = @{ Label ='Cmdlets'
                    Expression = {(Get-Command -module $_.name).count}}
PS C:\Foo> Get-Module Az* -ListAvailable |
             Sort-Object {(Get-command -Module $_.Name).Count} -Descending |
               Format-Table -Property Name, Version, Author,$HT -AutoSize

Name                          Version Author                Cmdlets
----                          ------- ------                -------
Az.Network                    4.7.0   Microsoft Corporation     641
Az.Sql                        2.17.0  Microsoft Corporation     269
Az.Compute                    4.10.0  Microsoft Corporation     203
Az.RecoveryServices           3.5.0   Microsoft Corporation     189
Az.Resources                  3.4.0   Microsoft Corporation     147
Az.Storage                    3.5.0   Microsoft Corporation     146
Az.ApiManagement              2.2.0   Microsoft Corporation     136
Az.CosmosDB                   1.1.0   Microsoft Corporation     105
Az.IotHub                     2.7.3   Microsoft Corporation      93
Az.Automation                 1.5.2   Microsoft Corporation      89
Az.DataLakeStore              1.3.0   Microsoft Corporation      87
Az.DataFactory                1.11.5  Microsoft Corporation      86
Az.Batch                      3.1.0   Microsoft Corporation      81
Az.KeyVault                   3.4.1   Microsoft Corporation      75
Az.Monitor                    2.4.0   Microsoft Corporation      71
Az.DataLakeAnalytics          1.0.2   Microsoft Corporation      62
Az.Websites                   2.5.0   Microsoft Corporation      62
Az.ServiceFabric              2.3.0   Microsoft Corporation      60
Az.OperationalInsights        2.3.0   Microsoft Corporation      55
Az.LogicApp                   1.5.0   Microsoft Corporation      51
Az.ServiceBus                 1.5.0   Microsoft Corporation      47
Az.HDInsight                  4.2.0   Microsoft Corporation      43
Az.Cdn                        1.6.0   Microsoft Corporation      42
Az.Accounts                   2.2.7   Microsoft Corporation      41
Az.DataBoxEdge                1.1.0   Microsoft Corporation      39
Az.Kusto                      1.0.1   Microsoft Corporation      39
Az.EventHub                   1.7.2   Microsoft Corporation      37
Az.DataShare                  1.0.0   Microsoft Corporation      36
Az.DesktopVirtualization      2.1.1   Microsoft Corporation      36
Az.FrontDoor                  1.7.0   Microsoft Corporation      34
Az.NotificationHubs           1.1.1   Microsoft Corporation      32
Az.ContainerRegistry          2.2.1   Microsoft Corporation      30
Az.AnalysisServices           1.1.4   Microsoft Corporation      28
Az.Migrate                    1.0.0   Microsoft Corporation      27
Az.DeploymentManager          1.1.0   Microsoft Corporation      24
Az.RedisCache                 1.4.0   Microsoft Corporation      23
Az.TrafficManager             1.0.4   Microsoft Corporation      22
Az.StreamAnalytics            1.0.1   Microsoft Corporation      21
Az.StorageSync                1.4.0   Microsoft Corporation      21
Az.Relay                      1.0.3   Microsoft Corporation      20
Az.Aks                        2.0.2   Microsoft Corporation      19
Az.EventGrid                  1.3.0   Microsoft Corporation      19
Az.RedisEnterpriseCache       1.0.0   Microsoft Corporation      19
Az.ApplicationInsights        1.1.0   Microsoft Corporation      17
Az.MachineLearning            1.1.3   Microsoft Corporation      16
Az.Billing                    2.0.0   Microsoft Corporation      16
Az.CognitiveServices          1.8.0   Microsoft Corporation      15
Az.PrivateDns                 1.0.3   Microsoft Corporation      15
Az.Functions                  2.0.0   Microsoft Corporation      15
Az.PowerBIEmbedded            1.1.2   Microsoft Corporation      15
Az.SqlVirtualMachine          1.1.0   Microsoft Corporation      14
Az.SignalR                    1.2.0   Microsoft Corporation      12
Az.Maintenance                1.1.0   Microsoft Corporation      11
Az.Dns                        1.1.2   Microsoft Corporation      11
Az.Media                      1.1.1   Microsoft Corporation      11
Az.DevTestLabs                1.0.2   Microsoft Corporation      10
Az.PolicyInsights             1.4.1   Microsoft Corporation       9
Az.Support                    1.0.0   Microsoft Corporation       8
Az.Databricks                 1.1.0   Microsoft Corporation       8
Az.AppConfiguration           1.0.0   Microsoft Corporation       7
Az.ManagedServices            2.0.0   Microsoft Corporation       6
Az.Advisor                    1.1.1   Microsoft Corporation       5
Az.HealthcareApis             1.2.0   Microsoft Corporation       4
Az.ContainerInstance          1.0.3   Microsoft Corporation       4
Az.MarketplaceOrdering        1.0.2   Microsoft Corporation       2
Az                            5.7.0   Microsoft Corporation       0
```

Figure 13.2: Viewing Azure modules

In *step 4*, you find the Azure AD module on the PowerShell Gallery. The output of this step is as follows:

```
PS C:\Foo> # 4. Finding Azure AD cmdlets
PS C:\Foo> Find-Module AzureAD |
             Format-Table -Property Name,Version,Author -AutoSize -Wrap

Name    Version   Author
----    -------   ------
AzureAD 2.0.2.130 Microsoft Corporation
```

Figure 13.3: Finding the Azure AD module

In *step 5*, you install the Azure AD module, which generates no output to the console. In *step 6*, you look at more information about this module, with output like this:

```
PS C:\Foo> # 6. Discovering Azure AD module
PS C:\Foo> $FTHT = @{
             Property = 'Name', 'Version', 'Author', 'Description'
             AutoSize = $true
             Wrap     = $true
           }
PS C:\Foo> Get-Module -Name AzureAD -ListAvailable |
             Format-Table @FTHT

Name    Version   Author                Description
----    -------   ------                -----------
AzureAD 2.0.2.130 Microsoft Corporation Azure Active Directory V2 General Availability Module.
                                        PowerShell Module. This is the General Availability
                                        release of Azure Active Directory V2. For detailed
                                        information on how to install and run this module
                                        from the PowerShell Gallery including prerequisites, please
                                        refer to https://docs.microsoft.com/en-us/powershell/scripting/gallery/overview
```

Figure 13.4: Viewing more information on the Azure AD module

In *step 7*, you use `Connect-AzAccount` to log in to Azure. The output of this command looks like this:

```
PS C:\Foo> # 7. Logging into Azure
PS C:\Foo> $CredAZ = Get-Credential     # Enter your Azure Credential details
PowerShell credential request
Enter your credentials.
User: tfl@reskit.org
Password for user tfl@reskit.org: ************************
PS C:\Foo> $Account = Connect-AzAccount -Credential $CredAZ
PS C:\Foo> $Account

Account         SubscriptionName TenantId                             Environment
-------         ---------------- --------                             -----------
tfl@reskit.org Reskit.Org       b7280e42-9be9-469e-97c2-04201d196d39 AzureCloud
```

Figure 13.5: Logging in to Azure

Once you have logged in successfully, in *step 8*, you view the account name for this subscription, with output like this:

```
PS C:\Foo> # 8. Getting Azure subscription details
PS C:\Foo> $AccountN = $Account.Context.Account.Id
PS C:\Foo> "Azure Account    : $AccountN"
Azure Account    : tfl@reskit.org
```

Figure 13.6: Viewing the Azure account name

In *step 9*, you view details of the Azure subscription, with output like this:

```
PS C:\Foo> # 9. Viewing Azure subscription
PS C:\Foo> $SubID = $Subscription.Context.Subscription.Id
PS C:\Foo> Get-AzSubscription -SubscriptionId $SubId |
             Format-List -Property *

Id                          : 449846ba-2d26-45cf-8420-df1aade52b42
Name                        : Free Trial
State                       : Enabled
SubscriptionId              : 449846ba-2d26-45cf-8420-df1aade52b42
TenantId                    : b7280e42-9be9-469e-97c2-04201d196d39
HomeTenantId                : b7280e42-9be9-469e-97c2-04201d196d39
ManagedByTenantIds          : {}
CurrentStorageAccountName   :
SubscriptionPolicies        : {
                                "LocationPlacementId": "Public_2014-09-01",
                                "QuotaId": "Reskit.Org_2014-09-01",
                                "SpendingLimit": "On"
                              }
ExtendedProperties          : {[Tenants, b7280e42-9be9-469e-97c2-04201d196d39], [Environment, AzureCloud],
                              [HomeTenant, b7280e42-9be9-469e-97c2-04201d196d39], [Account, tfl@reskit.org],
                              [SubscriptionPolices, {"locationPlacementId":"Public_2014-09-01","quotaId":
                              "Reskit.Org_2014-09-01","spendingLimit":"On"}]}
CurrentStorageAccount       :
```

Figure 13.7: Viewing details of the Azure subscription

In *step 10*, you use the `Get-AzLocation` cmdlet to discover and count the Azure locations of Azure's worldwide collection of data centers. The output, at least at the time of writing, looks like this:

```
PS C:\Foo> # 10. Getting Azure locations
PS C:\Foo> $AZL = Get-AzLocation
PS C:\Foo> $LOC = $AZL | Sort-Object Location
PS C:\Foo> "Azure locations:  [{0}]" -f $LOC.Count
Azure locations:  [42]
```

Figure 13.8: Counting Azure global locations

In *step 11*, you view the Azure locations, with output like this:

```
PS C:\Foo> # 11. Viewing Azure locations
PS C:\Foo> $LOC |
             Format-Table Location, DisplayName

Location           DisplayName
--------           -----------
australiacentral   Australia Central
australiacentral2  Australia Central 2
australiaeast      Australia East
australiasoutheast Australia Southeast
brazilsouth        Brazil South
brazilsoutheast    Brazil Southeast
canadacentral      Canada Central
canadaeast         Canada East
centralindia       Central India
centralus          Central US
eastasia           East Asia
eastus             East US
eastus2            East US 2
francecentral      France Central
francesouth        France South
germanynorth       Germany North
germanywestcentral Germany West Central
japaneast          Japan East
japanwest          Japan West
koreacentral       Korea Central
koreasouth         Korea South
northcentralus     North Central US
northeurope        North Europe
norwayeast         Norway East
norwaywest         Norway West
southafricanorth   South Africa North
southafricawest    South Africa West
southcentralus     South Central US
southeastasia      Southeast Asia
southindia         South India
switzerlandnorth   Switzerland North
switzerlandwest    Switzerland West
uaecentral         UAE Central
uaenorth           UAE North
uksouth            UK South
ukwest             UK West
westcentralus      West Central US
westeurope         West Europe
westindia          West India
westus             West US
westus2            West US 2
westus3            West US 3
```

Figure 13.9: Viewing Azure global locations

Microsoft has created several independent Azure environments, each with an independent management portal and sets of services. The `Az` cmdlets work with any environment you can access, although some services may not exist in all environments. In *step 12*, you view the current Azure environments, with output like this:

```
PS C:\Foo> # 12. Getting Azure environments
PS C:\Foo> Get-AzEnvironment |
             Format-Table -Property name, ManagementPortalURL

Name             ManagementPortalUrl
----             -------------------
AzureChinaCloud  https://go.microsoft.com/fwlink/?LinkId=301902
AzureUSGovernment https://manage.windowsazure.us
AzureGermanCloud https://portal.microsoftazure.de/
AzureCloud       https://go.microsoft.com/fwlink/?LinkId=254433
```

Figure 13.10: Viewing Azure cloud environments

There's more...

In *step 3*, you enumerate the Azure modules loaded in the previous step, and for each one, you display the name and the number of commands in the module. Some of the modules are large and contain a wealth of commands; others are small. While the `AZ.Network` module contains over 600 commands, the `Az.MarketPlaceOrdering` module contains just 2 commands. By the time you read this book, the number of modules and the commands in each have probably changed. The core modules and cmdlets you use in this chapter should not have changed, but you never know.

In *step 4* through *step 6*, you find, download, and examine the Azure AD module. You use these cmdlets to manage the Azure AD service. This chapter does not cover using these cmdlets.

In *step 7*, you log in to Azure. In production, using a simple username and password, even a long one, is not secure. A best practice is to use **multi-factor authentication** (**MFA**). In which case, you could use `Connect-AzAccount` without parameters and log in via Microsoft's GUI. This recipe shows how to log in to Azure using only a credential object.

In *step 10* and *step 11*, you count and view the Azure locations around the world. Each Azure location is one, and sometimes more than one, physical data center containing a large number of compute and data servers that deliver Azure services. Not every Azure location delivers all Azure service offerings, especially as Microsoft rolls out new features. Microsoft is constantly investing in new locations, so there may be even more locations by the time you are reading this chapter.

Microsoft maintains several separate independently run and operated Azure environments. In addition to the public Azure cloud environment, Microsoft provides three additional parallel environments: China, Germany, and the US government. In *step 12*, you view the publicly acknowledged environments – there may be more (or not).

Creating Azure resources

In the previous recipe, you created and used the basic Azure management environment by downloading the key modules, logging in to Azure, and looking at the environment. In this recipe, you create certain key Azure assets, including a resource group, a storage account, and tags.

You create all Azure resources within a **resource group**. A resource group is a grouping of Azure resources. Resource groups are fundamental to managing Azure resources.

Any storage you create within Azure resides in a storage account, a fundamental building block within Azure. You create a storage account within one of the Azure locations you saw in the *Getting started using Azure with PowerShell* recipe. When you create your storage account, you specify the level of resiliency and durability that you wish Azure to provide. There are several levels of replication provided within Azure. These levels provide for multiple copies of the data, which Microsoft replicates automatically. You can have Azure store these replicas in the local Azure data region or other Azure locations. These data replicas provide extra resilience and an increased likelihood of recovery should the unthinkable happen and an entire data center somehow fails in a catastrophic way.

You can provision a storage account as either standard or premium. A standard storage account allows you to store any data (as you see in the *Exploring the Azure storage account* recipe). A premium storage account provides extra features but at a cost.

Tags are name/value pairs that allow you to organize your resources within your subscription. For more details on how you can use tags to organize your Azure resources, see `https://docs.microsoft.com/azure/azure-resource-manager/resource-group-using-tags/`.

Getting ready

You run this recipe on `SRV1`, on which you have installed the `Az` module(s).

How to do it...

1. Setting key variables

```
$Locname    = 'uksouth'     # Location name
$RgName     = 'packt_rg'    # Resource group we are using
$SAName     = 'packt42sa'   # A unique storage account name
```

2. Logging in to your Azure account

```
$CredAZ  = Get-Credential
$Account = Connect-AzAccount
```

3. Creating a resource group and tagging it

```
$RGTag  = [Ordered] @{Publisher='Packt'
                      Author='Thomas Lee'}
$RGHT = @{
    Name     = $RgName
    Location = $Locname
    Tag      = $RGTag
}
$RG = New-AzResourceGroup @RGHT
```

4. Viewing the resource group with tags

```
Get-AzResourceGroup -Name $RGName |
    Format-List -Property *
```

5. Testing to see if the storage account name is taken

```
Get-AzStorageAccountNameAvailability $SAName
```

In *step 5*, you ensure that the storage account name is unique, which means the name is available for you to use. If the name is not unique, you should change the value of `$SAName` and retry the creation of the storage account before proceeding.

6. Creating a new storage account

```
$SAHT = @{
  Name              = $SAName

  SkuName           = 'Standard_LRS'
  ResourceGroupName = $RgName
  Tag               = $RGTag
  Location          = $Locname

}
New-AzStorageAccount @SAHT
```

7. Getting an overview of the storage account in this resource group

```
$SA = Get-AzStorageAccount -ResourceGroupName $RgName
$SA |
  Format-List -Property *
```

8. Getting primary endpoints for the storage account

```
$SA.PrimaryEndpoints
```

9. Reviewing the SKU

```
$SA.Sku
```

10. Viewing the storage account's context property

```
$SA.Context
```

How it works...

In *step 1*, you define key variables, which produces no console output. In *step 2*, you log in to your Azure account, entering your credentials via the GUI. In the *Getting started using Azure with PowerShell* recipe, you logged in using your username and password. In this recipe, you use the GUI to sign in, using MFA if you have that configured. When you complete the login process, you should see this in the default browser:

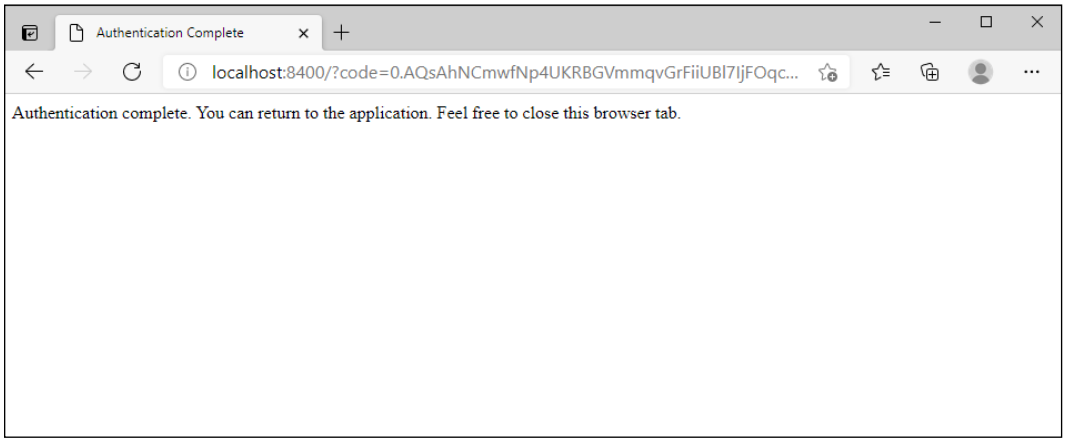


Figure 13.11: Logging in to Azure

In *step 3*, you create a new Azure resource group with two tags. This step creates no output to the console.

In *step 4*, you view the newly created resource group, with output like this:

```
PS C:\Foo> # 4. Viewing the resource group with tags
PS C:\Foo> Get-AzResourceGroup -Name $RGName |
             Format-List -Property *

ResourceGroupName : packt_rg
Location          : uksouth
ProvisioningState : Succeeded
Tags              : {Publisher, Author}
TagsTable         :
                    Name       Value
                    =========  ==========
                    Publisher  Packt
                    Author     Thomas Lee

ResourceId        : /subscriptions/449846ba-2d26-45cf-8420-df1aade52b42/resourceGroups/packt_rg
ManagedBy
```

Figure 13.12: Viewing the new Azure resource group

With Azure, a storage account name must be globally unique. In *step 5*, you use the `Get-AzureAccountNameAvailability` cmdlet to test whether the name you want to use for the storage account is available. The output looks like this:

```
PS C:\Foo> # 5. Testing to see if the storage account name is taken
PS C:\Foo> Get-AzStorageAccountNameAvailability $SAName

NameAvailable Reason Message
------------- ------ -------
         True
```

Figure 13.13: Testing the availability of the storage account name

In *step 6*, you create a new storage account, which generates the following output:

```
PS C:\Foo> # 6. Creating a new storage account
PS C:\Foo> $SAHT = @{
             Name              = $SAName
             SkuName           = 'Standard_LRS'
             ResourceGroupName = $RgName
             Tag               = $RGTag
             Location          = $Locname
           }
PS C:\Foo> New-AzStorageAccount @SAHT | Format-List

ResourceGroupName            : packt_rg
StorageAccountName           : packt42sa
Id                           : /subscriptions/449846ba-2d26-45cf-8420-df1aade52b42/resourceGroups/
                               packt_rg/providers/Microsoft.Storage/storageAccounts/packt42sa
Location                     : uksouth
Sku                          : Microsoft.Azure.Commands.Management.Storage.Models.PSSku
Kind                         : StorageV2
Encryption                   : Microsoft.Azure.Management.Storage.Models.Encryption
AccessTier                   : Hot
CreationTime                 : 04/20/2021 09:57:52
CustomDomain                 :
Identity                     :
LastGeoFailoverTime          :
PrimaryEndpoints             : Microsoft.Azure.Management.Storage.Models.Endpoints
PrimaryLocation              : uksouth
ProvisioningState            : Succeeded
SecondaryEndpoints           :
SecondaryLocation            :
StatusOfPrimary              : Available
StatusOfSecondary            :
Tags                         : {[Author, Thomas Lee], [Publisher, Packt]}
EnableHttpsTrafficOnly       : True
AzureFilesIdentityBasedAuth  :
EnableHierarchicalNamespace  :
FailoverInProgress           :
LargeFileSharesState         :
NetworkRuleSet               : Microsoft.Azure.Commands.Management.Storage.Models.PSNetworkRuleSet
RoutingPreference            :
BlobRestoreStatus            :
GeoReplicationStats          :
AllowBlobPublicAccess        :
MinimumTlsVersion            :
EnableNfsV3                  :
AllowSharedKeyAccess         :
Context                      : Microsoft.WindowsAzure.Commands.Common.Storage.LazyAzureStorageContext
ExtendedProperties           : {}
```

Figure 13.14: Creating a new storage account

In *step 7*, you retrieve and view the storage account details, with output like this:

```
PS C:\Foo> # 7. Getting an overview of the storage account in this resource group
PS C:\Foo> $SA = Get-AzStorageAccount -ResourceGroupName $RgName
PS C:\Foo> $SA |
             Format-List -Property *

ResourceGroupName           : packt_rg
StorageAccountName          : packt42sa
Id                          : /subscriptions/449846ba-2d26-45cf-8420-df1aade52b42/resourceGroups/
                              packt_rg/providers/Microsoft.Storage/storageAccounts/packt42sa
Location                    : uksouth
Sku                         : Microsoft.Azure.Commands.Management.Storage.Models.PSSku
Kind                        : StorageV2
Encryption                  : Microsoft.Azure.Management.Storage.Models.Encryption
AccessTier                  : Hot
CreationTime                : 03/04/2021 10:19:11
CustomDomain                :
Identity                    :
LastGeoFailoverTime         :
PrimaryEndpoints            : Microsoft.Azure.Management.Storage.Models.Endpoints
PrimaryLocation             : uksouth
ProvisioningState           : Succeeded
SecondaryEndpoints          :
SecondaryLocation           :
StatusOfPrimary             : Available
StatusOfSecondary           :
Tags                        : {[Author, Thomas Lee], [Publisher, Packt]}
EnableHttpsTrafficOnly      : True
AzureFilesIdentityBasedAuth :
EnableHierarchicalNamespace :
FailoverInProgress          :
LargeFileSharesState        :
NetworkRuleSet              : Microsoft.Azure.Commands.Management.Storage.Models.PSNetworkRuleSet
RoutingPreference           :
BlobRestoreStatus           :
GeoReplicationStats         :
AllowBlobPublicAccess       :
MinimumTlsVersion           :
EnableNfsV3                 :
AllowSharedKeyAccess        :
Context                     : Microsoft.WindowsAzure.Commands.Common.Storage.LazyAzureStorageContext
ExtendedProperties          : {}
```

Figure 13.15: Viewing your new storage account

Each of the Azure Storage endpoint types (such as a blob, file, or queue) is prefixed with your storage account name followed by a standardized primary endpoint suffix. You view these endpoints in *step 8*, with output like this:

```
PS C:\Foo> # 8. Getting primary endpoints for the storage account
PS C:\Foo> $SA.PrimaryEndpoints

Blob               : https://packt42sa.blob.core.windows.net/
Queue              : https://packt42sa.queue.core.windows.net/
Table              : https://packt42sa.table.core.windows.net/
File               : https://packt42sa.file.core.windows.net/
Web                : https://packt42sa.z33.web.core.windows.net/
Dfs                : https://packt42sa.dfs.core.windows.net/
MicrosoftEndpoints : null
```

Figure 13.16: Viewing primary Azure Storage endpoint suffixes

In *step 9*, you view the **Stock-Keeping Unit** (**SKU**) for this storage group, with output like this:

```
PS C:\Foo> # 9. Reviewing the SKU
PS C:\Foo> $SA.Sku

Name         : Standard_LRS
Tier         : Standard
ResourceType :
Kind         :
Locations    :
Capabilities :
Restrictions :
```

Figure 13.17: Viewing the Azure storage group's SKU

In the final step in this recipe, *step 10*, you view the storage account's context property, with output like this:

```
PS C:\Foo> # 10. Viewing the storage account's context property
PS C:\Foo> $SA.Context

BlobEndPoint       : https://packt42sa.blob.core.windows.net/
TableEndPoint      : https://packt42sa.table.core.windows.net/
QueueEndPoint      : https://packt42sa.queue.core.windows.net/
FileEndPoint       : https://packt42sa.file.core.windows.net/
StorageAccount     : BlobEndpoint=https://packt42sa.blob.core.windows.net/;QueueEndpoint=https://packt4
                     2sa.queue.core.windows.net/;TableEndpoint=https://packt42sa.table.core.windows.net
                     /;FileEndpoint=https://packt42sa.file.core.windows.net/;AccountName=packt42sa;Acco
                     untKey=[key hidden]
StorageAccountName : packt42sa
Context            : Microsoft.WindowsAzure.Commands.Common.Storage.LazyAzureStorageContext
Name               : packt42sa
EndPointSuffix     : core.windows.net/
ConnectionString   : BlobEndpoint=https://packt42sa.blob.core.windows.net/;QueueEndpoint=https://packt4
                     2sa.queue.core.windows.net/;TableEndpoint=https://packt42sa.table.core.windows.net
                     /;FileEndpoint=https://packt42sa.file.core.windows.net/;AccountName=packt42sa;Acco
                     untKey=L8H/U7DvDq420S4o9ECy2LzAI1Ekh/zaglT6A6tSj9ShJLe2EI94EaJOQzHivf3cixV4f2tQ8up
                     cyIYFOZc9ig==
ExtendedProperties : {}
```

Figure 13.18: Viewing the storage group's context property

There's more...

In *step 1*, you define variables that hold important values you use later in the recipe. If you are performing this recipe, you need to change the values you use in this step to reflect your unique names for your resource group and storage account.

In *step 5*, you test to check whether the proposed storage account name is available. Someone else may have used the storage group's name, meaning it is not available to you. If your choice of storage account name has already been taken, you need to adjust the name to be unique. You could use a GUID for the account name, which you create using the `New-Guid` command as one alternative, although that may be less easy to type.

In *step 9*, you examined the storage account's SKU. The SKU represents the specific resource tier in which you have created the storage account. Each tier, each SKU, has different capabilities and different prices.

In *step 10*, you examine the storage account's context. Azure uses context objects to hold subscription and authentication information. For more information on context objects in Azure, see `https://docs.microsoft.com/powershell/azure/context-persistence`.

Exploring the Azure storage account

Many Azure features use Azure Storage. When you create an Azure VM, for example, you store the VHD file in Azure Storage. Azure storage accounts can hold a variety of data, with different mechanisms for managing each data type. Additionally, the storage account provides both scalability and data durability and resiliency.

Azure Storage manages five distinct types of data:

- Binary large object (blob)
- Table
- Queue
- File
- Disk

A blob is unstructured data that you store in Azure. Blob storage can hold any type of data in any form – MP4 movies, ISO images, VHD drives, JPG files, and more. Individual blobs reside within blob containers, are equivalent to file store folders, but with very limited nesting capability.

Blobs come in three types: block blobs, append blobs, and page blobs. Block blobs are physically optimized for storing documents in the cloud and for streaming applications. Append blobs are optimized for append operations and are useful for logging. Page blobs are optimized for read/write operations – Azure VHDs, for example, are always of the page blob type. For more information about blob types, take a look at `https://docs.microsoft.com/azure/storage/blobs/storage-blobs-introduction`.

An Azure table is a non-relational storage system that utilizes key-value pairs. You can use Azure tables for storing unstructured or semi-structured data. Azure tables are not the same as SQL tables, which hold highly normalized data. An Azure table consists of a grouping of entities. See `https://azure.microsoft.com/services/storage/tables/` for more information about Azure table storage.

An Azure queue is a durable message queuing feature that you can use to implement massively scalable applications. With message queues, one part of an application can write a transaction to the queue for another part to process. A queue enables you to decouple application components for independent scaling and to provide greater resiliency. For more details on Azure queues, see `https://azure.microsoft.com/services/storage/queues/`.

The Azure Files feature provides cross-platform file storage that you can access using SMB. Azure Files allows you to create and use SMB file shares in the cloud and access them just like you would access on-premises SMB shares. Azure Files supports SMB 3.0, making it simple and easy to migrate legacy applications that rely on file shares. For more information on Azure Files, see `https://azure.microsoft.com/services/storage/files/`.

Azure's disk storage provides persistent, highly secure disk options, particularly for Azure VMs. Azure has designed its disk storage service for low latency and high throughput. You can create disks using both traditional spinning disks as well as SSD disks. SSD disks provide better I/O performance for I/O-intensive applications but are generally more expensive. For more details on Azure disk storage, see `https://azure.microsoft.com/services/storage/disks/`.

Azure's storage features continue to evolve, with more options available as time goes by. For more details on Azure Storage as a whole, see `https://docs.microsoft.com/azure/storage/common/storage-introduction`.

As we noted earlier, Azure names storage accounts based on a global naming scheme that uses HTTPS URLs. The Azure REST API relies on these URLs to manage the Azure resources in your resource groups. The HTTP URL contains your storage account followed by the Azure data type and then `.core.windows.net`, like this:

```
https://<storageaccountname>.<datatype>.core.windows.net/...
```

Getting ready

This recipe uses `SRV1`, which you have used to access Azure so far in this chapter.

How to do it...

1. Setting key variables

```
$Locname    = 'uksouth'          # location name
$RgName     = 'packt_rg'         # resource group we are using
$SAName     = 'packt42sa'        # storage account name
$CName      = 'packtcontainer'   # a blob container name
$CName2     = 'packtcontainer2' # a second blob container name
```

2. Logging in to your Azure account

```
$Account = Connect-AzAccount
```

3. Getting and displaying the storage account key

```
$SAKHT = @{
    Name              = $SAName
    ResourceGroupName = $RgName
```

```
}
$Sak = Get-AzStorageAccountKey  @SAKHT
$Sak
```

4. Extracting the first key's "password"

```
$Key = ($Sak | Select-Object -First 1).Value
```

5. Getting the storage account context, which encapsulates credentials for the storage account

```
$SCHT = @{
    StorageAccountName = $SAName
    StorageAccountKey = $Key
}
$SACon = New-AzStorageContext @SCHT
$SACon
```

6. Creating two blob containers

```
$CHT = @{
  Context    = $SACon
  Permission = 'Blob'
}
New-AzStorageContainer -Name $CName @CHT
New-AzStorageContainer -Name $CName2 @CHT
```

7. Viewing blob containers

```
Get-AzStorageContainer -Context $SACon |
    Select-Object -ExpandProperty CloudBlobContainer
```

8. Creating a blob

```
'This is a small Azure blob!!' | Out-File .\azurefile.txt
$BHT = @{
    Context = $SACon
    File = '.\azurefile.txt'
    Container = $CName
}
$Blob = Set-AzStorageBlobContent  @BHT
$Blob
```

9. Constructing and displaying the blob name

```
$BlobUrl = "$($Blob.Context.BlobEndPoint)$CName/$($Blob.name)"
$BlobUrl
```

10. Downloading and viewing the blob

```
$OutFile = 'C:\Foo\Test.Txt'
Start-BitsTransfer -Source $BlobUrl -Destination $OutFile
Get-Content -Path $OutFile
```

How it works...

In *step 1*, you set values for variables you use later in this recipe, creating no console output. If you discover, in the *Getting started using Azure with PowerShell* recipe, that you cannot acquire the storage account name, you need to create the storage with a different name. Don't forget to change the name in this step (and in subsequent recipes in this chapter).

In *step 2*, you connect to your Azure account using the GUI. As with other recipes, once you finish, you should see this in the default browser:

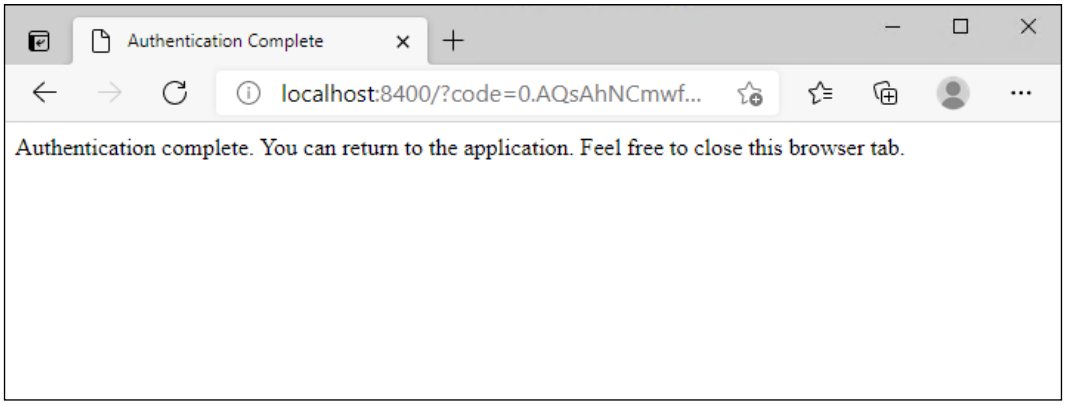


Figure 13.19: Logging in to Azure

In *step 3*, you use the `Get-AzStorageAccountKey` cmdlet to retrieve your storage account's key data, with output like this:

```
PS C:\Foo> # 3. Getting and displaying the storage account key
PS C:\Foo> $SAKHT = @{
             Name              = $SAName
             ResourceGroupName = $RgName
           }
PS C:\Foo> $Sak = Get-AzStorageAccountKey  @SAKHT
PS C:\Foo> $Sak

KeyName Value                                                                    Permissions
------- -----
key1    L8H/U7DvDq9uXS4o9ECy2LzAI1Ekh/ziglT6A6tSj9ShJLe2EI94EaJOQzHivf3cixV420tQ8upcyIYFOZc9ig==         Full
key2    7cFKLhqmlRdlLsftJAG/WgJpPsFSW5uJGaXBVaQ8R8QzsD4854fKtNXzh420Cpg+KX5ad6Qkacfu8cHaC/KBNw==         Full
```

Figure 13.20: Getting and viewing your storage account key

In *step 4*, you extract the first key to later serve as a password. This step creates no output to the console. In *step 5*, you view the storage account's context, which looks like this:

```
PS C:\Foo> # 5. Getting the storage account context which encapsulates credentials
PS C:\Foo> #   for the storage account
PS C:\Foo> $SCHT = @{
             StorageAccountName = $SAName
             StorageAccountKey = $Key
           }
PS C:\Foo> $SACon = New-AzStorageContext @SCHT
PS C:\Foo> $SACon

StorageAccountName  : packt42sa
BlobEndPoint        : https://packt42sa.blob.core.windows.net/
TableEndPoint       : https://packt42sa.table.core.windows.net/
QueueEndPoint       : https://packt42sa.queue.core.windows.net/
FileEndPoint        : https://packt42sa.file.core.windows.net/
Context             : Microsoft.WindowsAzure.Commands.Storage.AzureStorageContext
Name                :
StorageAccount      : BlobEndpoint=https://packt42sa.blob.core.windows.net/;
                      QueueEndpoint=https://packt42sa.queue.core.windows.net/;
                      TableEndpoint=https://packt42sa.table.core.windows.net/;
                     FileEndpoint=https://packt42sa.file.core.windows.net/;
                      AccountName=packt42sa;AccountKey=[key hidden]
TableStorageAccount : BlobEndpoint=https://packt42sa.blob.core.windows.net/;
                      QueueEndpoint=https://packt42sa.queue.core.windows.net/;
                      TableEndpoint=https://packt42sa.table.core.windows.net/;
                      FileEndpoint=https://packt42sa.file.core.windows.net/;
                      DefaultEndpointsProtocol=https;AccountName=packt42sa;AccountKey=[key hidden]
Track2OauthToken    :
EndPointSuffix      : core.windows.net/
ConnectionString    : BlobEndpoint=https://packt42sa.blob.core.windows.net/;
                      QueueEndpoint=https://packt42sa.queue.core.windows.net/;
                      TableEndpoint=https://packt42sa.table.core.windows.net/;
                      FileEndpoint=https://packt42sa.file.core.windows.net/;
                      AccountName=packt42sa;AccountKey=[key hidden]]
```

Figure 13.21: Getting and viewing your storage account's context object

In *step 6*, you create two new Azure Storage blob containers, with output like this:

```
PS C:\Foo> # 6. Creating 2 blob containers
PS C:\Foo> $CHT = @{
             Context    = $SACon
             Permission = 'Blob'
           }
PS C:\Foo> New-AzStorageContainer -Name $CName @CHT

   Blob End Point: https://packt42sa.blob.core.windows.net/

Name                PublicAccess         LastModified
----                ------------         ------------
packtcontainer      Blob                 04/04/2021 20:04:15 +00:00

PS C:\Foo> New-AzStorageContainer -Name $CName2 @CHT

   Blob End Point: https://packt42sa.blob.core.windows.net/

Name                PublicAccess         LastModified
----                ------------         ------------
packtcontainer2     Blob                 04/04/2021 20:04:15 +00:00
```

Figure 13.22: Creating two Azure Storage blob containers

In *step 7*, you use the `Get-AzStorageContainer` cmdlet to get details about the Azure blob containers within the storage account, with output like this:

```
PS C:\Foo> # 7. Viewing blob containers
PS C:\Foo> Get-AzStorageContainer -Context $SACon |
             Select-Object -ExpandProperty CloudBlobContainer

   Blob End Point: https://packt42sa.blob.core.windows.net/

Name             Uri                                                    LastModified
----             ---                                                    ------------
packtcontainer   https://packt42sa.blob.core.windows.net/packtcontainer   2021-04-04 20:04:15Z
packtcontainer2  https://packt42sa.blob.core.windows.net/packtcontainer2  2021-04-04 20:04:15Z
```

Figure 13.23: Viewing Azure Storage blob containers

In *step 8*, you create a new Azure blob with output like this:

```
PS C:\Foo> # 8. Creating a blob
PS C:\Foo> 'This is a small Azure blob!!' | Out-File .\azurefile.txt
PS C:\Foo> $BHT = @{
             Context = $SACon
             File = '.\azurefile.txt'
             Container = $CName
           }
PS C:\Foo> $Blob = Set-AzStorageBlobContent  @BHT
PS C:\Foo> $Blob

   AccountName: packt42sa, ContainerName: packtcontainer

Name          BlobType  Length  ContentType               LastModified         AccessTier SnapshotTime  IsDeleted  VersionId
----          --------  ------  -----------               ------------         ---------- ------------  ---------  ---------
azurefile.txt BlockBlob 30      application/octet-stream  2021-04-04 20:05:22Z Hot                       False
```

Figure 13.24: Creating an Azure blob

In *step 9*, you display the blob's URL, with output like this:

```
PS C:\Foo> # 9. Constructing and displaying the blob name
PS C:\Foo> $BlobUrl = "$($Blob.Context.BlobEndPoint)$CName/$($Blob.name)"
PS C:\Foo> $BlobUrl
https://packt42sa.blob.core.windows.net/packtcontainer/azurefile.txt
```

Figure 13.25: Displaying the blob's URL

In *step 10*, you use the URL of the new blob to download the file and view it, like this:

```
PS C:\Foo> # 10. Downloading and viewing the blob
PS C:\Foo> $OutFile = 'C:\Foo\Test.Txt'
PS C:\Foo> Start-BitsTransfer -Source $BlobUrl -Destination $OutFile
PS C:\Foo> Get-Content -Path $OutFile
This is a small Azure blob!!
```

Figure 13.26: Viewing the blob in your default browser

There's more...

In *step 3*, you display your storage account's account key, which contains two passwords for your account.

In *step 10*, you use the **Background Intelligent Transfer Service** (**BITS**) to download the blob to a local file. You could have also used your default browser and navigated to the blob's URL, or used other means to download the data you stored in Azure.

Creating an Azure SMB file share

Azure provides you with the ability to create SMB shares with an Azure storage account. These SMB shares act the same as the local on-premises SMB shares you used in *Chapter 10*, *Managing Shared Data*. The key difference is how you create them and the credentials that you use to access the shares.

Before an SMB client can access data held in an SMB share, the SMB client needs to authenticate with the SMB server. With Windows-based shares, you either specify a user credential object (user ID and password) or, in a domain environment, the SMB client utilizes Kerberos to authenticate. With Azure, you use the storage account name as the user ID and the storage account key as the password.

The storage account key contains two properties, imaginatively named `key1` and `key2`. The values of these two properties are valid passwords for Azure SMB file shares. Having two keys enables you to do regular key rotation. If your application uses the value of `key1`, you can reconfigure your application to use the `key2` value as the share's password and then regenerate the `key1` value. At a later time, you can repeat this, changing the application to use the now updated `key1`'s value and then regenerating `key2`. Key rotation provides you with an immediate key update where you need it. Armed with either key value, you can create SMB shares that you can directly access across the Internet.

Azure SMB file shares differ from Azure blobs in terms of how you access the data. You access a blob via HTTP, whereas you access an Azure file share via the standard SMB 3 networking protocol that you used in, for example, *Chapter 10*. Also, with Azure blob storage, you can only have a single level of a folder (that is, the blob container). With Azure Files, on the other hand, you can have as many folders as you need.

When using Azure SMB shares, the storage account key is the password for the share, and the storage account name is the user ID. As with all credentials, you should exercise caution when including the account key in your PowerShell scripts.

In this recipe, you use the resource group and storage account we created earlier (in the *Creating Azure resources* recipe). This recipe also checks to ensure that these exist and creates them if they are not available.

Getting ready

This recipe uses SRV1, which you have used to access Azure so far in this chapter.

How to do it...

1. Defining variables

```
$Locname   = 'uksouth'      # location name
$RgName    = 'packt_rg'     # resource group we are using
$SAName    = 'packt42sa'    # storage account name
$ShareName = 'packtshare'   # must be lower case!
```

2. Logging in to your Azure account

```
$Account = Connect-AzAccount
```

3. Getting storage account, account key, and context

```
$SA = Get-AzStorageAccount -ResourceGroupName $Rgname
$SAKHT = @{
    Name              = $SAName
    ResourceGroupName = $RgName
}
$Sak = Get-AzStorageAccountKey @SAKHT
$Key = ($Sak | Select-Object -First 1).Value
$SCHT = @{
    StorageAccountName = $SAName
    StorageAccountKey  = $Key
}
$SACon = New-AzStorageContext @SCHT
```

4. Adding credentials to the local credentials store

```
$T = "$SAName.file.core.windows.net"
cmdkey /add:$T /user:"AZURE\$SAName" /pass:$Key
```

5. Creating an Azure share

```
New-AzStorageShare -Name $ShareName -Context $SACon
```

6. Checking that the share is reachable

```
$TNCHT = @{
  ComputerName = "$SAName.file.core.windows.net"
  Port         = 445
}
Test-NetConnection @TNCHT
```

7. Mounting the share as `M:`

```
$Mount = 'M:'
$Rshare = "\\$SaName.file.core.windows.net\$ShareName"
$SMHT = @{
    LocalPath  = $Mount
    RemotePath = $Rshare
    UserName   = $SAName
    Password   = $Key
}
New-SmbMapping @SMHT
```

8. Viewing the share in Azure

```
Get-AzStorageShare -Context $SACon  |
    Format-List -Property *
```

9. Viewing local SMB mappings

```
Get-SmbMapping -LocalPath M:
```

10. Creating a file in the share

```
New-Item -Path M:\Foo -ItemType Directory | Out-Null
'Azure and PowerShell 7 Rock!!!' |
   Out-File -FilePath M:\Foo\Recipe.Txt
```

11. Retrieving details about the share contents

```
Get-ChildItem -Path M:\ -Recurse |
    Format-Table -Property FullName, Mode, Length
```

12. Getting the content from the file

```
Get-Content -Path M:\Foo\Recipe.txt
```

13. Cleaning up – removing data from the share

```
Remove-Item -Path M: -Force -Recurse -ErrorAction SilentlyContinue
```

14. Cleaning up – removing SMB mapping

```
Get-SmbMapping -LocalPath M: |
  Remove-SmbMapping -Force
```

15. Removing the Azure share

```
Get-AzStorageShare -Name $ShareName -Context $SACon |
  Remove-AzStorageShare
```

How it works...

With *step 1*, you define variables to hold important Azure names you use in this recipe, which generates no output. In *step 2*, you log in using the Azure GUI and MFA. Once you have logged in successfully, you see the following:

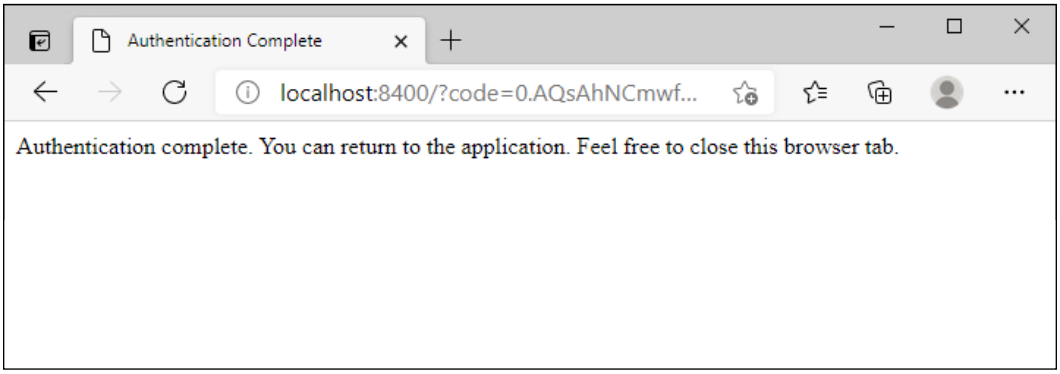


Figure 13.27: Logging in to your Azure account

In *step 3*, you retrieve the storage account, account key, and context, which generates no output to the console. To simplify your use of SMB shares, in *step 4*, you add credentials in the local credential store. Windows uses these credentials when connecting to the share. The output of this step is as follows:

```
PS C:\Foo> # 4. Adding credentials to the local credentials store
PS C:\Foo> $T = "$SAName.file.core.windows.net"
PS C:\Foo> cmdkey /add:$T /user:"AZURE\$SAName" /pass:$Key

CMDKEY: Credential added successfully.
```

Figure 13.28: Storing credentials in the local Windows credential store

In *step 5*, you create an Azure SMB file share, with output like this:

```
PS C:\Foo> # 5. Creating an Azure share
PS C:\Foo> New-AzStorageShare -Name $ShareName -Context $SACon

   File End Point: https://packt42sa.file.core.windows.net/

Name       QuotaGiB LastModified               IsSnapshot SnapshotTime
----       -------- ------------               ---------- ------------
packtshare          05/04/2021 10:29:03 +00:00 False
```

Figure 13.29: Creating an Azure SMB file share

With *step 6*, you use `Test-NetConnection` to verify you can reach the Azure storage host that holds your newly created SMB share, with output like this:

```
PS C:\Foo> # 6. Checking that the share is reachable
PS C:\Foo> $TNCHT = @{
             ComputerName = "$SAName.file.core.windows.net"
             Port         = 445
           }
PS C:\Foo> Test-NetConnection @TNCHT

ComputerName     : packt42sa.file.core.windows.net
RemoteAddress    : 51.141.128.40
RemotePort       : 445
InterfaceAlias   : PureVPN
SourceAddress    : 10.2.141.212
TcpTestSucceeded : True
```

Figure 13.30: Testing connection to the Azure file share

In *step 7*, you create an SMB drive mapping, mapping the local `M:` drive to your Azure file share. This step produces output like this:

```
PS C:\Foo> # 7. Mounting the share as M:
PS C:\Foo> $Mount = 'M:'
PS C:\Foo> $Rshare = "\\$SaName.file.core.windows.net\$ShareName"
PS C:\Foo> $SMHT = @{
             LocalPath  = $Mount
             RemotePath = $Rshare
             UserName   = $SAName
             Password   = $Key
           }
PS C:\Foo> New-SmbMapping @SMHT

Status Local Path Remote Path
------ ---------- ------------
OK     M:         \\packt42sa.file.core.windows.net\packtshare
```

Figure 13.31: Creating a new SMB drive mapping

With *step 8*, you use the `Get-AzStorageShare` cmdlet to view the share's properties from within Azure. The output from this step looks like this:

```
PS C:\Foo> # 8. Viewing the share in Azure
PS C:\Foo> Get-AzStorageShare -Context $SACon  |
             Format-List -Property *

CloudFileShare  : Microsoft.Azure.Storage.File.CloudFileShare
SnapshotTime    :
IsSnapshot      : False
LastModified    : 05/04/2021 10:29:03 +00:00
Quota           : 5120
ShareClient     : Azure.Storage.Files.Shares.ShareClient
ShareProperties : Azure.Storage.Files.Shares.Models.ShareProperties
Context         : Microsoft.WindowsAzure.Commands.Storage.AzureStorageContext
Name            : packtshare
```

Figure 13.32: Viewing the Azure file share

In *step 9*, you view the local SMB mapping, with output like this:

```
PS C:\Foo> # 9. Viewing local SMB mappings
PS C:\Foo> Get-SmbMapping -LocalPath M:

Status       Local Path Remote Path
------       ---------- -----------
Connected    M:         \\packt42sa.file.core.windows.net\packtshare
```

Figure 13.33: Viewing the local SMB share mapping

In *step 10*, you create a new file in the Azure file share, generating no output. In *step 11*, you view details of the share contents, including the new file you just created. The output of this step looks like this:

```
PS C:\Foo> # 11. Retrieving details about the share contents
PS C:\Foo> Get-ChildItem -Path M:\ -Recurse |
             Format-Table -Property FullName, Mode, Length

FullName           Mode  Length
--------           ----  ------
M:\Foo             d----
M:\Foo\Recipe.Txt -a--- 30
```

Figure 13.34: Retrieving details about the share contents

In *step 12*, you examine the contents of the file you retrieved from Azure, with output like this:

```
PS C:\Foo> # 12. Getting the content from the file
PS C:\Foo> Get-Content -Path M:\Foo\Recipe.txt

Azure and PowerShell 7 Rock!!!
```

Figure 13.35: Getting the content from the file

In *step 13*, you begin the cleanup process by removing all the data in the SMB share. In *step 14*, you remove the local SMB mapping. In *step 15*, you remove the share from Azure. These final three steps produce no output.

There's more...

In *step 2*, you log in to your Azure account. If you have just completed working with any of the Azure recipes in this chapter, Azure may still have you logged in. In that case, you can skip this step.

In *step 5*, you create the Azure file share, which you then view in *step 6*. Depending on your Internet connection and your Internet router, you may have issues here. Some small business routers by default block SMB traffic outbound. You may be able to reconfigure the router. If not, you can always use a VPN. You could use one of the popular Internet VPN suppliers and use any VPN into the Internet. Once you have your VPN connection established and routing packets onto the Internet, your access to this share should be successful. The screenshot taken in this chapter relied on using a VPN from PureVPN. See `https://www.purevpn.com/` for more details on this firm's offerings.

In *step 13*, *step 14*, and *step 15*, you tidy up and remove the SMB share from Azure, including the data (stored in Azure) and the local SMB mapping. If you intend to keep the share, then omit these final steps.

Creating an Azure website

Azure provides many ways for you to create rich web and mobile applications in the cloud. You could set up your VMs, install IIS, and add your web application. If your application needs to store data, you can create a separate SQL Server VM (or use Azure's SQL Database PaaS offering).

A simpler way is to create an Azure App Service. An Azure App Services enables you to build, deploy, and manage rich websites and web applications. You can use frameworks such as .NET, Node.js, PHP, and Python in these applications and any database software that's appropriate for your needs. You can also take advantage of its DevOps capabilities, such as continuous deployment from Azure DevOps, GitHub, Docker Hub, and other sources, package management, staging environments, custom domain, and TLS/SSL certificates.

An Azure App Service web app can be a simple static HTML site or a rich multi-tier application providing both web and mobile platforms. You have a lot of choices and features to exploit.

In this recipe, you create a simple single-page static website. You upload the page via FTP. To simplify the content upload, you use the PSFTP third-party module, which you get and install from the PowerShell Gallery.

Getting ready

This recipe uses `SRV1`, which you have used to access Azure so far in this chapter.

How to do it...

1. Defining variables

```
$Locname    = 'uksouth'     # location name
$RgName     = 'packt_rg'    # resource group we are using
$SAName     = 'packt42sa'   # storage account name
$AppSrvName = 'packt42'
$AppName    = 'packt42website'
```

2. Logging in to your Azure account

```
Login-AzAccount -Credential $CredAz
```

3. Getting the resource group

```
$RGHT1 = @{
  Name        = $RgName
  ErrorAction = 'Silentlycontinue'
}
$RG = Get-AzResourceGroup @RGHT1
```

4. Getting the storage account

```
$SAHT = @{
  Name              = $SAName
  ResourceGroupName = $RgName
  ErrorAction       = 'SilentlyContinue'
}
$SA = Get-AzStorageAccount @SAHT
```

5. Creating an application service plan

```
$SPHT = @{
     ResourceGroupName   = $RgName
     Name                = $AppSrvName
     Location            = $Locname
     Tier                =  'Free'
}
New-AzAppServicePlan @SPHT |  Out-Null
```

6. Viewing the service plan

```
$PHT = @{
  ResourceGroupName = $RGname
  Name              = $AppSrvName
}
Get-AzAppServicePlan @PHT
```

7. Creating the new Azure web app

```
$WAHT = @{
  ResourceGroupName = $RgName
  Name              = $AppName
  AppServicePlan    = $AppSrvName
  Location          = $Locname
}
New-AzWebApp @WAHT |  Out-Null
```

The web app name, held in the `$AppName` variable, needs to be unique globally, just like the storage account name. Therefore, this step is only successful if the name is unique. If you get errors from this step, you need to change the web app name until you find one that works.

8. Viewing web app details

```
$WebApp = Get-AzWebApp -ResourceGroupName $RgName -Name $AppName
$WebApp |
  Format-Table -Property Name, State, Hostnames, Location
```

9. Checking the website

```
$SiteUrl = "https://$($WebApp.DefaultHostName)"
Start-Process -FilePath $SiteUrl
```

10. Installing the PSFTP module

```
Install-Module PSFTP -Force | Out-Null
Import-Module PSFTP
```

11. Getting publishing profile XML and extracting FTP upload details

```
$APHT = @{
  ResourceGroupName = $RgName
  Name              = $AppName
  OutputFile        = 'C:\Foo\pdata.txt'
}
$X = [xml] (Get-AzWebAppPublishingProfile @APHT)
$X.publishData.publishProfile[1]
```

12. Extracting credentials and site details

```
$UserName = $X.publishData.publishProfile[1].userName
$UserPwd  = $X.publishData.publishProfile[1].userPWD
$Site     = $X.publishData.publishProfile[1].publishUrl
```

13. Setting the FTP connection

```
$FTPSN  = 'FTPtoAzure'
$PS     = ConvertTo-SecureString $UserPWD -AsPlainText -Force
$T      = 'System.Management.automation.PSCredentiaL'
$Cred   = New-Object -TypeName $T -ArgumentList $UserName,$PS
$FTPHT  = @{
  Credentials = $Cred
  Server      = $Site
  Session     = $FTPSN
  UsePassive  = $true
}
Set-FTPConnection @FTPHT
```

14. Opening an FTP session

```
$Session = Get-FTPConnection -Session $FTPSN
$Session
```

15. Creating a web page and uploading it

```
'My First Azure Web Site' | Out-File -FilePath C:\Foo\Index.Html
$Filename = 'C:\foo\index.html'
$IHT = @{
  Path       = '/'
  LocalPath  = 'C:\foo\index.html'
  Session    = $FTPSN
}
Add-FTPItem @IHT
```

16. Examining the site using your default browser

```
$SiteUrl = "https://$($WebApp.DefaultHostName)"
Start-Process -FilePath $SiteUrl
```

17. Tidying up – removing the web app

```
$WebApp | Remove-AzWebApp  -Force
```

18. Tidying up – removing the service plan

```
Get-AzAppServicePlan @PHT |
  Remove-AzAppServicePlan -Force
```

How it works...

As with other recipes in this chapter, in *step 1,* you define variables for use with this recipe. This step produces no console output.

In *step 2*, you log in to your Azure account, using the Azure GUI and MFA. Once you are logged in, you should see the following in your default browser:

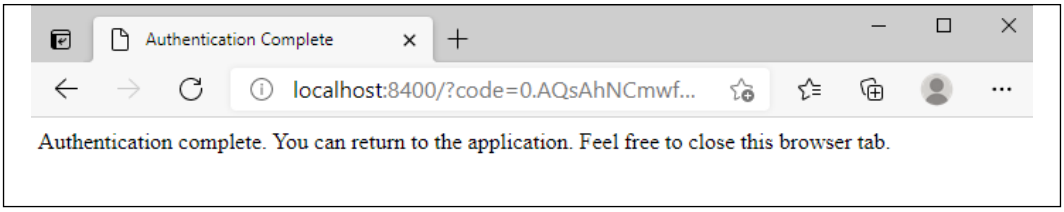


Figure 13.36: Logging in to Azure

In *step 3*, you get the resource group, and in *step 4*, you get your storage account. You use these objects in this recipe. In *step 5*, you create the Azure App Service plan using the Free resource tier. These three steps create no console output.

In *step 6*, you view the App Service plan with output like this:

```
PS C:\Foo> # 6. Viewing the service plan
PS C:\Foo> $PHT = @{
             ResourceGroupName = $RGname
             Name              = $AppSrvName
           }
PS C:\Foo> Get-AzAppServicePlan @PHT

WorkerTierName            :
Status                    : Ready
Subscription              : 449846ba-2d26-45cf-8420-df1aade52b42
HostingEnvironmentProfile :
MaximumNumberOfWorkers    : 1
GeoRegion                 : UK South
PerSiteScaling            : False
MaximumElasticWorkerCount : 1
NumberOfSites             : 0
IsSpot                    : False
SpotExpirationTime        :
FreeOfferExpirationTime   :
ResourceGroup             : packt_rg
Reserved                  : False
IsXenon                   : False
HyperV                    : False
TargetWorkerCount         : 0
TargetWorkerSizeId        : 0
ProvisioningState         : Succeeded
Sku                       : Microsoft.Azure.Management.WebSites.Models.SkuDescription
Id                        : /subscriptions/449846ba-2d26-45cf-8420-df1aade52b42/resourceGroups/
                            packt_rg/providers/Microsoft.Web/serverfarms/packt42
Name                      : packt42
Kind                      : app
Location                  : UK South
Type                      : Microsoft.Web/serverfarms
Tags                      :
```

Figure 13.37: Viewing the App Service plan

In *step 7*, you create the actual web application (website) using the `New-AzWebApp` cmdlet. There is no output from this step.

In *step 8*, you use the `Get-AzWebApp` command to view details of your new web application, with output like this:

```
PS C:\Foo> # 8. Viewing web app details
PS C:\Foo> $WebApp = Get-AzWebApp -ResourceGroupName $RgName -Name $AppName
PS C:\Foo> $WebApp |
             Format-Table -Property Name, State, Hostnames, Location


Name           State   HostNames                          Location
----           -----   ---------                          --------
packt42website Running {packt42website.azurewebsites.net} UK South
```

Figure 13.38: Viewing the Azure web application

In *step 9*, you use your default browser to navigate to the website, with output like this:

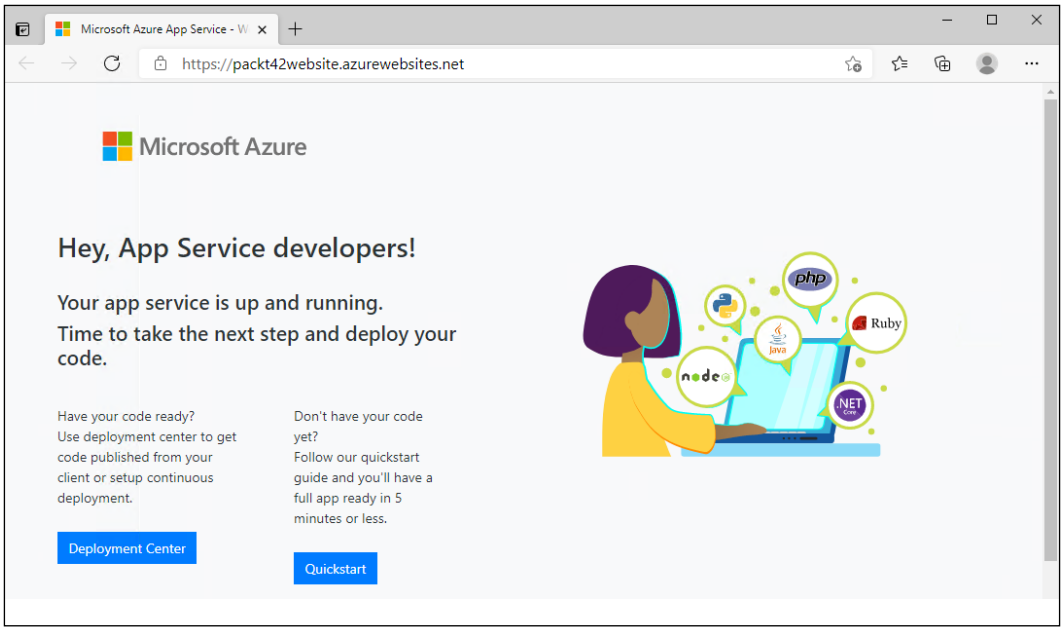


Figure 13.39: Viewing the Azure website

To add content to your site, you need to upload the site contents using FTP. You can use the `PSFTP` module to do the uploads. You first have to download and install the `PSFTP` module from the PowerShell Gallery, which you do in *step 10*, producing no output.

In *step 11*, you use `Get-AzWebAppPublishingProfile` to get the Azure publishing profile. This object contains details on how you can publish your website contents. This step produces the following output:

```
PS C:\Foo> # 11.  Getting publishing profile XML and extracting FTP upload details
PS C:\Foo> $APHT = @{
             ResourceGroupName = $RgName
             Name              = $AppName
             OutputFile        = 'C:\Foo\Pdata.Txt'
           }
PS C:\Foo> $X = [xml] (Get-AzWebAppPublishingProfile @APHT)
PS C:\Foo> $X.publishData.publishProfile[1]

profileName                 : packt42website - FTP
publishMethod               : FTP
publishUrl                  : ftp://waws-prod-ln1-059.ftp.azurewebsites.windows.net/site/wwwroot
ftpPassiveMode              : True
userName                    : packt42website\$packt42website
userPWD                     : ABBfspe706Zu2C7G086vTfqlhRt420qbBcTXaFfvjpk7vjW3X2N7YMv9jePl
destinationAppUrl           : http://packt42website.azurewebsites.net
SQLServerDBConnectionString :
mySQLDBConnectionString     :
hostingProviderForumLink    :
controlPanelLink            : http://windows.azure.com
webSystem                   : WebSites
```

Figure 13.40: Obtaining Azure site publishing details

In *step 12*, you extract the site, username, and password for the FTP site to upload site content. This step produces no output.

In *step 13*, you create an FTP session to upload your site content. The output of this step looks like this:

```
PS C:\Foo> # 13. Setting the FTP connection
PS C:\Foo> $FTPSN  = 'FTPtoAzure'
PS C:\Foo> $PS     = ConvertTo-SecureString $UserPWD -AsPlainText -Force
PS C:\Foo> $T      = 'System.Management.automation.PSCredentiaL'
PS C:\Foo> $Cred   = New-Object -TypeName $T -ArgumentList $UserName,$PS
PS C:\Foo> $FTPHT  = @{
               Credentials = $Cred
               Server      = $Site
               Session     = $FTPSN
               UsePassive  = $true
             }
PS C:\Foo> Set-FTPConnection @FTPHT
PS C:\Foo> $Session = Get-FTPConnection -Session $FTPSN

ContentLength           : -1
Headers                 : {}
SupportsHeaders         : True
ResponseUri             : ftp://waws-prod-ln1-059.ftp.azurewebsites.windows.net/site/wwwroot
StatusCode              : ClosingData
StatusDescription       : 226 Transfer complete.

LastModified            : 01/01/0001 00:00:00
BannerMessage           : 220 Microsoft FTP Service

WelcomeMessage          : 230 User logged in.

ExitMessage             : 221 Goodbye.

ContentType             :
IsFromCache             : False
IsMutuallyAuthenticated : False
```

Figure 13.41: Setting an FTP connection

In *step 14*, you open an FTP session to Azure and display the details, with output like this:

```
PS C:\Foo> # 14. Opening an FTP session
PS C:\Foo> $Session = Get-FTPConnection -Session $FTPSN
PS C:\Foo> $Session

Session    RequestUri                                                    User           Alive Binary Passive Ssl
-------    ----------                                                    ----           ----- ------ ------- ---
FTPtoAzure ftp://waws-prod-ln1-005.ftp.azurewebsites.windows.net/site/wwwroot $packt42website True  False  True    False
```

Figure 13.42: Opening an FTP session with Azure

In *step 15*, you create a simple web page, `Index.html`, and upload it using the FTP session opened earlier in this recipe. The output from this step looks like this:

```
PS C:\Foo> # 15. Creating a web page and uploading it
PS C:\Foo> 'My First Azure Web Site' | Out-File -FilePath C:\Foo\Index.Html
PS C:\Foo> $Filename = 'C:\Foo\Index.html'
PS C:\Foo> $IHT = @{
             Path       = '/'
             LocalPath  = 'C:\Foo\Index.html'
             Session    = $FTPSN
           }
PS C:\Foo> Add-FTPItem @IHT

   Parent: ftp://waws-prod-ln1-059.ftp.azurewebsites.windows.net/site/wwwroot

Dir Right     Ln  User   Group  Size   ModifiedDate         Name
--- -----     --  ----   -----  ----   ------------         ----
-                            25B    04-06-21  04:20PM    Index.html
```

Figure 13.43: Creating and uploading a web page

Now that you have uploaded some content, in *step 16*, you view your website again, with output like this:

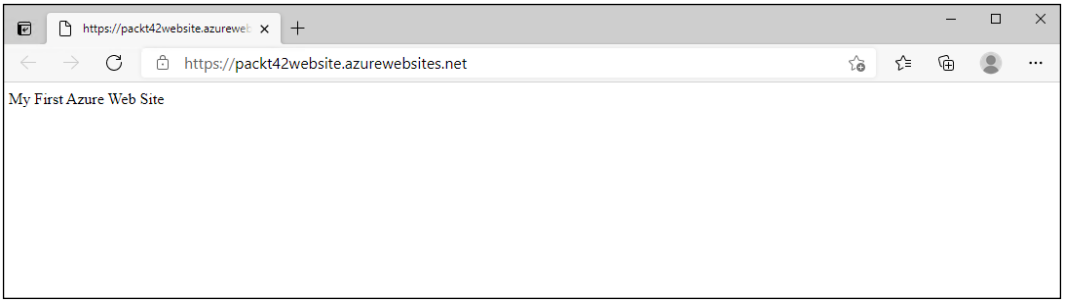


Figure 13.44: Viewing your website

There's more...

In *step 9*, you view your newly created Azure website. The output shown in *Figure 13.39* is what Azure displays before you upload actual site content. This graphic tells you that the Azure site is up and running, letting you know you now need to upload some content.

In *step 11*, you get the publishing profile. This object contains the FTP publishing URL and the username and password for the FTP site. This object also gives you the full URL to the website itself.

In *step 15*, you create then upload a single HTML page (`Index.html`), generating no console output.

In *step 16*, you use your default browser to view this site – this time, you see your site's content. Of course, in production, there would be more pages and other files you would upload. It can take a few seconds before Azure displays the updated content.

In *step 17*, you begin to tidy up by removing the web app. In *step 18*, you complete the tidy-up by removing the service plan from Azure. These two steps produce no output to the console.

Creating an Azure Virtual Machine

Azure provides a range of on-demand computing resources, one of which is VMs. An Azure VM is a good solution for more control over the computing environment than you might obtain using a PaaS service.

An Azure VM is essentially a Hyper-V VM that you run within Azure. There are some differences between the Hyper-V VMs you create within Server 2022 (or Windows 10) and Azure VMs, but they are minor. The Az cmdlets you use to manage Azure VMs are slightly different in style from the Hyper-V cmdlets, which may mean a bit of a learning curve.

Getting ready

This recipe uses `SRV1`, which you have used to access Azure so far in this chapter.

How to do it...

1. Defining key variables

```
$Locname = 'uksouth'          # location name
$RgName  = 'packt_rg'         # resource group name
$SAName  = 'packt42sa'        # Storage account name
$VNName  = 'packtvnet'        # Virtual Network Name
$CloudSN = 'packtcloudsn'     # Cloud subnet name
```

```
$NSGName = 'packt_nsg'        # NSG name
$Ports   = @(80, 3389)        # ports to open in VPN
$IPName  = 'Packt_IP1'        # Private IP Address name
$User    = 'AzureAdmin'       # User Name
$UserPS  = 'JerryRocks42!'    # User Password
$VMName  = 'Packt42VM'        # VM Name
```

2. Logging in to your Azure account

```
$CredAZ = Get-Credential     # Enter your Azure Credential details
$Account = Connect-AzAccount -Credential $CredAZ
```

3. Getting the resource group

```
$RG = Get-AzResourceGroup -Name $RgName
```

4. Getting the storage account

```
$SA = Get-AzStorageAccount -Name $SAName -ResourceGroupName $RgName
```

5. Creating VM credentials

```
$T = 'System.Management.Automation.PSCredential'
$P = ConvertTo-SecureString -String $UserPS -AsPlainText -Force
$VMCred = New-Object -TypeName $T -ArgumentList $User, $P
```

6. Creating a simple VM using defaults

```
$VMHT = @{
    ResourceGroupName   = $RgName
    Location            = $Locname
    Name                = $VMName
    VirtualNetworkName  = $VNName
    SubnetName          = $CloudSN
    SecurityGroupName   = $NSGName
    PublicIpAddressName = $IPName
    OpenPorts           = $Ports
    Credential          = $VMCred
}
New-AzVm @VMHT
```

7. Getting the VM's external IP address

```
$VMIP = Get-AzPublicIpAddress -ResourceGroupName $RGname
$VMIP = $VMIP.IpAddress
"VM Public IP Address: [$VMIP]"
```

8. Connecting to the VM

```
mstsc /v:"$VMIP"
```

9. Tidying up – stopping and removing the Azure VM

```
Stop-AzVm -Name $VMName -Resourcegroup $RgName -Force |
  Out-Null
Remove-AZVm -Resourcegroup $RgName -Name $VMName -Force |
  Out-Null
```

10. Tidying up – removing the VM's networking artifacts

```
Remove-AzNetworkInterface -Resourcegroup $RgName -Name $VMName -Force
Remove-AzPublicIpAddress -Resourcegroup $RgName -Name $IPName -Force
Remove-AzNetworkSecurityGroup -Resourcegroup $RgName -Name $NSGName -Force
Remove-AzVirtualNetwork -Name $VNName -Resourcegroup $RgName -Force
Get-AzNetworkWatcher | Remove-AzNetworkWatcher
Get-AzResourceGroup -Name NetworkWatcherRG  |
  Remove-AzResourceGroup -Force |
    Out-Null
```

11. Tidying up – removing the VM's disk

```
Get-AzDisk | Where-Object name -match "packt42vm" |
  Remove-AzDisk -Force |
    Out-Null
```

12. Tidying Up - Removing the storage account and resource group

```
$RHT = @{
  StorageAccountName = $SAName
  ResourceGroupName  = $RgName
}
Get-AzStorageAccount @RHT |
  Remove-AzStorageAccount -Force
Get-AzResourceGroup -Name $RgName |
  Remove-AzResourceGroup -Force
    Out-Null
```

How it works...

In *step 1*, you create variables to hold important values for this recipe. This step creates no console output.

In *step 2*, you log in to your Azure account, with output like this:

```
PS C:\Foo> # 2. Logging into your Azure account
PS C:\Foo> $CredAZ = Get-Credential
PowerShell credential request
Enter your credentials.
User: tfl@reskit.org
Password for user tfl@reskit.org: ************************
PS C:\Foo> $Account = Login-AzAccount -Credential $CredAZ
PS C:\Foo> $Account

Account         SubscriptionName TenantId                             Environment
-------         ---------------- --------                             -----------
tfl@reskit.org Reskit.Org       b7280ea2-9be9-469e-97c2-0d5d1d196d39 AzureCloud
```

Figure 13.45: Logging in to Azure

In *step 3*, you retrieve the resource group, and in *step 4*, you retrieve the storage account, creating no console output. You created these in an earlier step.

In *step 5*, you create a credential object you use with the Azure VM. This step also produces no output.

In *step 6*, you use the `New-AzVm` command to create a new Azure VM, which looks like this:

```
PS C:\Foo> # 6. Creating a simple VM using defaults
PS C:\Foo> $VMHT = @{
              ResourceGroupName   = $RgName
              Location            = $Locname
              Name                = $VMName
              VirtualNetworkName  = $VNName
              SubnetName          = $CloudSN
              SecurityGroupName   = $NSGName
              PublicIpAddressName = $IPName
              OpenPorts           = $Ports
              Credential          = $VMCred
           }
PS C:\Foo> New-AzVm @VMHT

ResourceGroupName        : packt_rg
Id                       : /subscriptions/449846ba-2d26-45cf-8818-df1aade52b7c/
                           resourceGroups/packt_rg/providers/Microsoft.Compute/
                           virtualMachines/Packt42VM
VmId                     : 933557e6-99fc-43c7-a740-003326eee246
NetworkProfile           : {NetworkInterfaces}
OSProfile                : {ComputerName, AdminUsername, WindowsConfiguration, Secrets,
                           AllowExtensionOperations, RequireGuestProvisionSignal}
ProvisioningState        : Succeeded
StorageProfile           : {ImageReference, OsDisk, DataDisks}
FullyQualifiedDomainName : packt42vm-0953fc.uksouth.cloudapp.azure.com
```

Figure 13.46: Creating an Azure VM

In *step 7*, you use the `Get-AzPublicIpAddress` cmdlet to obtain the IPv4 address of the Azure VM. The output should look like this:

```
PS C:\Foo> # 7. Getting the VM's external IP address
PS C:\Foo> $VMIP = Get-AzPublicIpAddress -ResourceGroupName $RGname
PS C:\Foo> $VMIP = $VMIP.IpAddress
PS C:\Foo> "VM Public IP Address: [$VMIP]"
VM Public IP Address: [51.140.116.221]
```

Figure 13.47: Creating an Azure VM

In *step 8*, you use the Windows RDP client and open up a terminal connection with the VM over the Internet. After typing a few commands in the terminal window, the results should look like this:

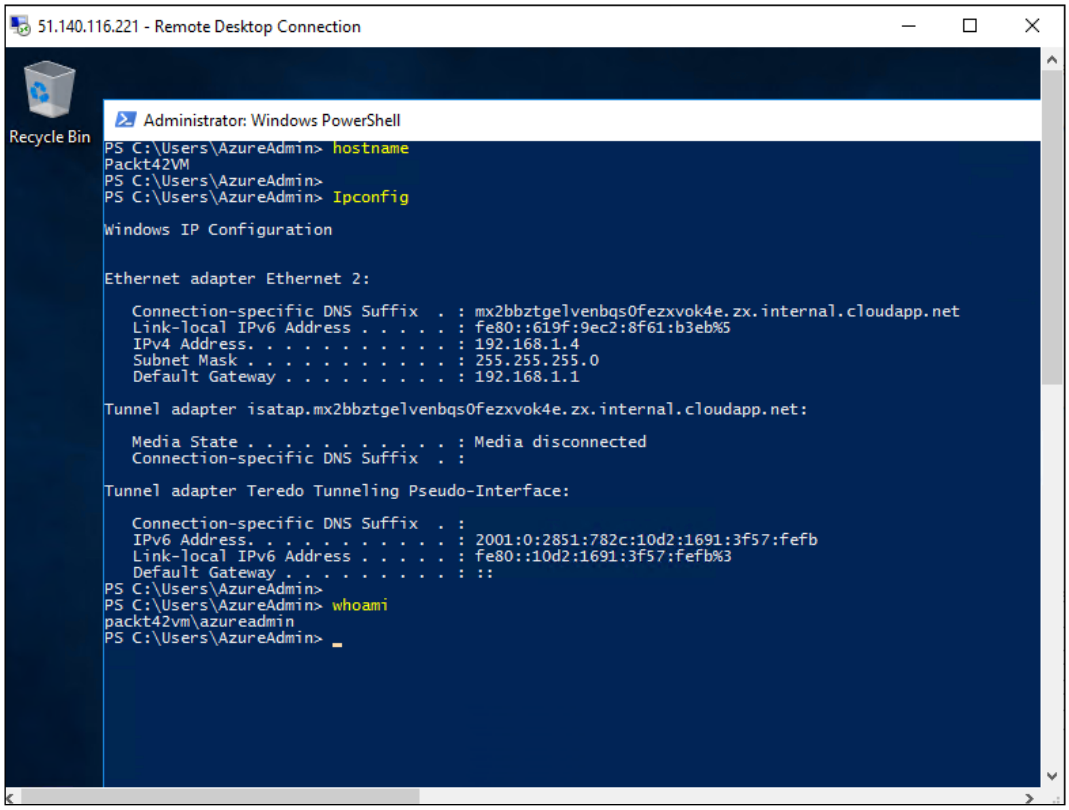


Figure 13.48: Using a terminal session into your Azure VM

In *step 9*, you begin to tidy up by stopping and then removing the Azure VM. In *step 10*, you remove the network-related artifacts, including the VM's network interface, the public IP address, and so on. In *step 11*, you remove the VM's virtual disk. In *step 12*, you remove both the storage account and resource group you have used in the recipes in this chapter. These four steps produce no console output.

There's more...

In *step 1*, you create variables to hold important values for this recipe. In production or testing, you would probably need to change some of these values since certain names, such as the storage account or Azure web app name, need to be unique globally. You need to test any such names for uniqueness. If you plan to deploy Azure VMs in production, you should devise a corporate naming convention. In creating your naming convention, you have to balance ease of use versus uniqueness. A 32-character VM storage account name is highly likely to be unique but can be challenging to type at the console.

In *step 6*, you create a new Azure VM. The creation process can take several minutes to complete since Azure has to create your VM on a compute server, provision a disk image, then complete the Windows Server installation. The performance you observe may well depend on the Azure location and its current workload. Also, when you created the VM, you did not specify the `-Size` parameter. Unless you specify a different VM size, Azure defaults to `Standard_D2s_V3`. For more information on VM sizes, see `https://docs.microsoft.com/azure/virtual-machines/sizes`.

In *step 8*, you use the Windows Remote Desktop client application to create a terminal session into your new Azure VM. Note that having the RDP port open inbound from the Internet provides good functionality, but it is a security risk. If you need RDP access, you access the RDP session through a VPN.

In the final four steps, you remove the Azure VM, the VM's network artifacts, the VM disk, and finally the storage account and resource group you have used throughout this recipe and chapter.

14

Troubleshooting with PowerShell

In this chapter, we cover the following recipes:

- Using PowerShell Script Analyzer
- Using Best Practices Analyzer
- Network troubleshooting
- Checking network connectivity using Get-NetView
- Exploring PowerShell script debugging

Introduction

You can think about debugging as the art and science of removing bugs from your PowerShell scripts. You may find your script does not always do what you or your users want, both during the development of the script and later, after you have used it in production. Troubleshooting is a process you go through to determine why your script is not doing what you want (and helps you resolve your issues with the script).

There are three broad classes of problems that you encounter:

- Syntax errors
- Logic errors
- Runtime errors

Syntax errors are all too common – especially if your typing is less than perfect. It is so easy to type `Get-ChildTiem` as opposed to `Get-ChildItem`. The good news is that until you resolve your syntax errors, your script cannot run successfully. There are several ways to avoid syntax errors and to simplify the task of finding and removing them. One simple way is to use a good code editor, such as VS Code. Just like Microsoft Word, VS Code highlights potential syntax errors to help you identify, fix, and eliminate them.

Another way to reduce at least some typos or other syntax issues is to use tab completion in the PowerShell console or the VS Code editor. You type some of the necessary text, hit the *Tab* key, and PowerShell does the rest of the typing for you.

Logic errors are bits of code that do not do what you want or what you expect. There is a myriad of reasons why code could have a logic error. One issue many IT pros encounter is defining a variable but not using it later or typing the variable name incorrectly. Tools such as **PowerShell Script Analyzer** can analyze your code and help you track down potential issues in your code.

You may have a working system or service that, in some cases, could become problematic if you are unlucky. **Best Practices Analyzer** enables you to examine core Windows services to ensure you run them in the best possible way.

You can also encounter runtime errors. For example, your script to add and configure a user in your AD could encounter a runtime problem. The AD service on a **domain controller** (**DC**) may have crashed, the **network interface controller** (**NIC**) in your DC might have failed, or the network path from a user to the DC might have a failed router or one with an incorrect routing table. Checking network connectivity ensures the network path from your user to the relevant servers is working as required. But you also need to ensure your networking configuration itself is correct.

Network issues, over and above basic connectivity, can be challenging to resolve since there are so many potential issues that might arise. Many issues are simple to diagnose and resolve. You can see this in the *Network troubleshooting* recipe. For more challenging issues, you may need to delve deeper into the networking stack to gather more information. The `Get-NetView` module and cmdlet are useful in obtaining a huge amount of troubleshooting information that a network professional can use to resolve problems.

Finally, an important tool for finding script errors is PowerShell's script debugging features. As we see in the final recipe, using these features makes it easier to find and remove errors in your scripts.

Using PowerShell Script Analyzer

PowerShell Script Analyzer is a PowerShell module produced by the PowerShell team that analyzes your code and provides opportunities to improve. You can download the latest version of the module from the PowerShell Gallery.

If you are using the VS Code editor to develop your code, you should know that Script Analyzer is built into VS Code. So as you are developing your PowerShell script, VS Code highlights any errors that Script Analyzer finds. VS Code, therefore, helps you to write better code straight away.

Another feature of PowerShell Script Analyzer is the ability to reformat PowerShell code to be more readable. You have numerous settings you can configure to tell Script Analyzer how it should reformat your code.

Getting ready

This recipe uses SRV1, a domain-joined Windows Server 2022 host.

How to do it...

1. Discovering the PowerShell Script Analyzer module

```
Find-Module -Name PSScriptAnalyzer |
  Format-List Name, Type, Desc*, Author, Company*, *Date, *URI*
```

2. Installing the Script Analyzer module

```
Install-Module -Name PSScriptAnalyzer -Force
```

3. Discovering the commands in the Script Analyzer module

```
Get-Command -Module PSScriptAnalyzer
```

4. Discovering analyzer rules

```
Get-ScriptAnalyzerRule |
  Group-Object -Property Severity |
    Sort-Object -Property Count -Descending
```

5. Examining a rule

```
Get-ScriptAnalyzerRule |
  Select-Object -First 1 |
    Format-List
```

6. Creating a script file with issues

```
@'
# Bad.ps1
# A file to demonstrate Script Analyzer
#
### Uses an alias
$Procs = gps
### Uses positional parameters
```

```
$Services = Get-Service 'foo' 21
### Uses poor function header
Function foo {"Foo"}
### Function redefines a built in command
Function Get-ChildItem {"Sorry Dave I cannot do that"}
### Command uses a hard coded computer name
Test-Connection -ComputerName DC1
### A line that has trailing white space
$foobar ="foobar"
### A line using a global variable
$Global:foo
'@ | Out-File -FilePath "C:\Foo\Bad.ps1"
```

7. Checking the newly created script file

```
Get-ChildItem C:\Foo\Bad.ps1
```

8. Analyzing the script file

```
Invoke-ScriptAnalyzer -Path C:\Foo\Bad.ps1 |
  Sort-Object -Property Line
```

9. Defining a function to format more nicely

```
$Script1 = @'
function foo {"hello!"
Get-ChildItem -Path C:\FOO
}
'@
```

10. Defining formatting settings

```
$Settings = @{
  IncludeRules = @("PSPlaceOpenBrace", "PSUseConsistentIndentation")
  Rules = @{
    PSPlaceOpenBrace = @{
      Enable = $true
      OnSameLine = $true
    }
    PSUseConsistentIndentation = @{
      Enable = $true
    }
  }
}
```

11. Invoking the formatter

```
Invoke-Formatter -ScriptDefinition $Script1 -Settings $Settings
```

12. Changing settings and reformatting

```
$Settings.Rules.PSPlaceOpenBrace.OnSameLine = $False
Invoke-Formatter -ScriptDefinition $Script1 -Settings $Settings
```

How it works...

In *step 1*, you use the `Find-Module` command to find the `PSScriptAnalyzer` module in the PowerShell Gallery. The output of this step looks like this:

```
PS C:\Foo> # 1. Discovering the PowerShell Script Analyzer module
PS C:\Foo> Find-Module -Name PSScriptAnalyzer |
             Format-List Name, Type, Desc*, Author, Company*, *Date, *URI*

Name           : PSScriptAnalyzer
Type           : Module
Description    : PSScriptAnalyzer provides script analysis and checks for potential
                 code defects in the scripts by applying a group of built-in or
                 customized rules on the scripts being analyzed.
Author         : Microsoft Corporation
CompanyName    : {PowerShellTeam, JamesTruher-MSFT}
PublishedDate  : 28/07/2020 16:41:52
InstalledDate  :
UpdatedDate    :
LicenseUri     : https://github.com/PowerShell/PSScriptAnalyzer/blob/master/LICENSE
ProjectUri     : https://github.com/PowerShell/PSScriptAnalyzer
IconUri        : https://raw.githubusercontent.com/powershell/psscriptanalyzer/master/logo.png
```

Figure 14.1: Finding the PowerShell Script Analyzer module

In *step 2*, you install the `PSScriptAnalyzer` module, generating no output. In *step 3*, you use the `Get-Command` command to discover the commands inside the module, with output like this:

```
PS C:\Foo> # 3. Discovering the commands in the Script Analyzer module
PS C:\Foo> Get-Command -Module PSScriptAnalyzer

CommandType   Name                     Version  Source
-----------   ----                     -------  ------
Cmdlet        Get-ScriptAnalyzerRule   1.19.1   PSScriptAnalyzer
Cmdlet        Invoke-Formatter         1.19.1   PSScriptAnalyzer
Cmdlet        Invoke-ScriptAnalyzer    1.19.1   PSScriptAnalyzer
```

Figure 14.2: Getting the commands in the Script Analyzer module

PowerShell Script Analyzer uses a set of rules that define potential problems with your scripts. In *step 4*, you use `Get-ScriptAnalyzerRule` to examine the types of rules available, with output like this:

```
PS C:\Foo> # 4. Discovering analyzer rules
PS C:\Foo> Get-ScriptAnalyzerRule |
             Group-Object -Property Severity |
               Sort-Object -Property Count -Descending


Count Name           Group
----- ----           -----
   46 Warning        {PSAlignAssignmentStatement, PSAvoidUsingCmdletAliases, PSAvoidAssignment…
   11 Information    {PSAvoidUsingPositionalParameters, PSAvoidTrailingWhitespace, PSAvoidUsin…
    7 Error          {PSAvoidUsingUsernameAndPasswordParams, PSAvoidUsingComputerNameHardcoded…
```

Figure 14.3: Examining Script Analyzer rules

You can view one of the Script Analyzer rules, as shown in *step 5*, with output like this:

```
PS C:\Foo> # 5. Examining a rule
PS C:\Foo> Get-ScriptAnalyzerRule |
             Select-Object -First 1 |
               Format-List

Name        :
Severity    : Warning
Description : Line up assignment statements such that the assignment operator are aligned.
SourceName  : PS
```

Figure 14.4: Examining a Script Analyzer rule

In *step 6*, which generates no console output, you create a script file with issues that Script Analyzer can detect and provide you with problems to resolve. In *step 7*, you check on the newly created script file, with output like this:

```
PS C:\Foo> # 7. Checking the newly created file
PS C:\Foo> Get-ChildItem C:\Foo\Bad.ps1

    Directory: C:\Foo

Mode            LastWriteTime   Length Name
----            -------------   ------ ----
-a---     11/04/2021    15:34      567 Bad.ps1
```

Figure 14.5: Checking the script file

In *step 8*, you use the `Invoke-ScriptAnalyzer` command to check the `C:\Foo\Bad.ps1` file for potential issues. The output from this step looks like this:

```
PS C:\Foo> # 8. Analyzing the script file
PS C:\Foo> Invoke-ScriptAnalyzer -Path C:\Foo\Bad.ps1 |
             Sort-Object -Property Line

RuleName                            Severity     ScriptName Line  Message
--------                            --------     ---------- ----  -------
PSAvoidUsingCmdletAliases           Warning      Bad.ps1    6     'gps' is an alias of 'Get-Process'. Alias can
                                                                  introduce possible problems and make scripts hard
                                                                  to maintain. Pleaseconsider changing alias to its
                                                                  full content.
PSUseDeclaredVarsMoreThanAssignments Warning     Bad.ps1    6     The variable 'Procs' is assigned but never used.
PSUseDeclaredVarsMoreThanAssignments Warning     Bad.ps1    9     The variable 'Services' is assigned but never used.
PSAvoidOverwritingBuiltInCmdlets    Warning      Bad.ps1    15    'Get-ChildItem' is a cmdlet that is included with
                                                                  PowerShell (version core-6.1.0-windows) whose
                                                                  definition should not be overridden
PSAvoidUsingComputerNameHardcoded   Error        Bad.ps1    18    The ComputerName parameter of cmdlet 'Test-Connection'
                                                                  is hardcoded. This will expose sensitive information
                                                                  about the system if the script is shared.
PSAvoidTrailingWhitespace           Information  Bad.ps1    21    Line has trailing whitespace
PSUseDeclaredVarsMoreThanAssignments Warning     Bad.ps1    21    The variable 'foobar' is assigned but never used.
PSAvoidGlobalVars                   Warning      Bad.ps1    24    Found global variable 'Global:foo'.
```

Figure 14.6: Analyzing the script file

Script Analyzer's secondary function is to reformat a script file to improve the script's layout. In *step 9*, you define a simple function that you later reformat. This step generates no console output.

Views on what constitutes a good code layout vary. Script Analyzer allows you to specify settings that govern how you wish the code to be formatted. In *step 10*, you specify some rule settings, which generates no output.

In *step 11*, you invoke the script formatter using the settings you specified in the previous step. The output of this step is as follows:

```
PS:\Foo> # 11. Invoking formatter
PS:\Foo> Invoke-Formatter -ScriptDefinition $Script1 -Settings $Settings
function foo {
    "hello!"
    Get-ChildItem -Path C:\FOO
}
```

Figure 14.7: Invoking the script formatter

In *step 12*, you change the rule's value that places an open brace character on the same line as, or a separate line to, say, the function name. The output of this step looks like this:

```
PS:\Foo> # 12. Changing settings and reformatting
PS:\Foo> $Settings.Rules.PSPlaceOpenBrace.OnSameLine = $False
PS:\Foo> Invoke-Formatter -ScriptDefinition $Script1 -Settings $Settings
function foo
{
    "hello!"
    Get-ChildItem -Path C:\FOO
}
```

Figure 14.8: Changing settings and reformatting the script file

There's more...

In *step 1*, you view details about the `PSScriptAnalyzer` module. Note that the author of this module was a long-time PowerShell team member, Jim Truher. Interestingly, Jim did some of the demonstrations the very first time Microsoft displayed Monad (as PowerShell was then named) at the PDC in the autumn of 2003.

In the final part of this recipe, you use the formatting feature to format a script file. The goal is to create easier-to-read code. That can be very useful for long production scripts. A consistent layout makes it easier to find issues, as well as simplifying any later script maintenance.

Step 12 sets a rule that makes Script Analyzer's formatter put a script block's opening brace on a separate line. Opinion varies as to whether this is a good approach. You always have options. There are other formatting options, such as lining up the = sign in a set of assignment statements, and many more. The documentation on these rules is not particularly helpful, but you can start here: `https://www.powershellgallery.com/packages/PSScriptAnalyzer/1.19.1/Content/Settings%5CCodeFormatting.psd1`.

Using Best Practices Analyzer

One way to avoid needing to perform troubleshooting is to deploy your services in a more trouble-free, or at least trouble-tolerant, manner. There are many ways to deploy and operate your environment, and some methods are demonstrably better than others. For example, having two DCs, two DNS servers with AD-integrated zones, and having two DHCP servers in a failover relationship means you can experience numerous issues in these core services and still deploy a reliable end-user service. While you may still need to do troubleshooting to resolve any issue, your services are running acceptably, with your users unaware that there is an issue.

Along with industry experts, MVPs, and others, Microsoft product teams have identified recommendations for how you should deploy a Windows infrastructure. Some product teams, such as Exchange, publish extensive guidance and have developed a self-contained tool.

The Windows Server **Best Practices Analyzer** (**BPA**) is a built-in Windows Server tool that analyzes your on-premises servers for adherence to best practices. A best practice is a guideline that industry experts agree is the best way to configure your servers. For example, most AD experts recommend you have at least *two* DC for each domain. But for a test environment, that may be overkill. So while best practices are ones to strive for, sometimes they may be inappropriate for your needs. It is, therefore, important to use some judgment when reviewing the results of BPA.

Important note: BPA does not work natively in PowerShell 7 on any supported Windows Server version, including (at the time of writing) Windows Server 2022. There is, however, a way around this that involves using PowerShell Remoting and running BPA in Windows PowerShell, as you can see from this recipe.

BPA with Windows Server 2022 comes with 14 BPA models. Each model is a set of rules that you can use to test your environment. The AD team has built a BPA model for AD, `Microsoft/Windows/DirectoryServices`, that you can run to determine issues with AD on a domain controller.

In this recipe, you create a PowerShell Remoting session with `DC1`. You use the `Invoke-Command` cmdlet to run the BPA cmdlets, allowing you to analyze, in this recipe, the AD model.

Getting ready

This recipe uses `SRV1`, a domain-joined Windows 2022 server in the `Reskit.Org` domain. You also need the domain controllers in Reskit.Org (`DC1` and `DC2`) online for this recipe.

How to do it...

1. Creating a remoting session to Windows PowerShell on DC1

```
$BPAS = New-PSSession -ComputerName DC1
```

2. Discovering the BPA module on DC1

```
$SB1 = {
  Get-Module -Name BestPractices -List |
    Format-Table -AutoSize
}
Invoke-Command -Session $BPAS -ScriptBlock $SB1
```

3. Discovering the commands in the BPA module

```
$SB2 = {
    Get-Command -Module BestPractices  |
      Format-Table -AutoSize
}
Invoke-Command -Session $BPAS -ScriptBlock $SB2
```

4. Discovering all available BPA models on DC1

```
$SB3 = {
  Get-BPAModel  |
    Format-Table -Property Name,Id, LastScanTime -AutoSize
}
Invoke-Command -Session $BPAS -ScriptBlock $SB3
```

5. Running the BPA DS model on DC1

```
$SB4 = {
  Invoke-BpaModel -ModelID Microsoft/Windows/
DirectoryServices -Mode ALL |
    Format-Table -AutoSize
}
Invoke-Command -Session $BPAS -ScriptBlock $SB4
```

6. Getting BPA results from DC1

```
$SB5 = {
    Get-BpaResult -ModelID Microsoft/Windows/DirectoryServices  |
      Where-Object Resolution -ne $null|
        Format-List -Property Problem, Resolution
}
Invoke-Command -Session $BPAS -ScriptBlock $SB5
```

How it works...

In *step 1*, you create a PowerShell Remoting session with your DC, `DC1`. This step creates no output. In *step 2*, you run the `Get-Module` command on `DC1`, using the remoting session. The output of this step looks like this:

```
PS C:\Foo> # 2. Discovering the BPA module on DC1
PS C:\Foo> $SB1 = {
             Get-Module -Name BestPractices -List |
               Format-Table -Autosize
           }
PS C:\Foo> Invoke-Command -Session $BPAS -ScriptBlock $SB1

    Directory: C:\WINDOWS\system32\WindowsPowerShell\v1.0\Modules

ModuleType Version Name          ExportedCommands
---------- ------- ----          ----------------
Manifest   1.0     BestPractices {Get-BpaModel, Get-BpaResult, Invoke-BpaModel, Set-BpaResult}
```

Figure 14.9: Viewing the BPA module on DC1

In *step 3*, you discover the commands contained in the BPA module (on `DC1`), with output like this:

```
PS C:\Foo> # 3. Discovering the commands in the BPA module
PS C:\Foo> $SB2 = {
             Get-Command -Module BestPractices  |
               Format-Table -AutoSize
           }
PS C:\Foo> Invoke-Command -Session $BPAS -ScriptBlock $SB2

CommandType Name            Version Source
----------- ----            ------- ------
Cmdlet      Get-BpaModel    1.0     BestPractices
Cmdlet      Get-BpaResult   1.0     BestPractices
Cmdlet      Invoke-BpaModel 1.0     BestPractices
Cmdlet      Set-BpaResult   1.0     BestPractices
```

Figure 14.10: Discovering the commands inside the BPA module

In *step 4*, you discover the BPA models that are available on `DC1`. The output looks like this:

```
PS C:\Foo> # 4. Discovering all available BPA models on DC1
PS C:\Foo> $SB3 = {
             Get-BPAModel  |
               Format-Table -Property Name,Id, LastScanTime -AutoSize
             }
PS C:\Foo> Invoke-Command -Session $BPAS -ScriptBlock $SB3

Name                                                Id                                                  LastScanTime
----                                                --                                                  ------------
RightsManagementServices                            Microsoft/Windows/ADRMS                             Never
CertificateServices                                 Microsoft/Windows/CertificateServices               Never
Microsoft DHCP Server Configuration Analysis Model  Microsoft/Windows/DHCPServer                        Never
DirectoryServices                                   Microsoft/Windows/DirectoryServices                 Never
Microsoft DNS Server Configuration Analysis Model   Microsoft/Windows/DNSServer                         Never
File Services                                       Microsoft/Windows/FileServices                      Never
Hyper-V                                             Microsoft/Windows/Hyper-V                           Never
LightweightDirectoryServices                        Microsoft/Windows/LightweightDirectoryServices      Never
Network Policy and Access Services (NPAS)           Microsoft/Windows/NPAS                              Never
Microsoft Remote Access Server Configuration Analysis Model Microsoft/Windows/RemoteAccessServer        Never
TerminalServices                                    Microsoft/Windows/TerminalServices                  Never
Windows Server Update Services                      Microsoft/Windows/UpdateServices                    Never
Microsoft Volume Activation Configuration Analysis Model Microsoft/Windows/VolumeActivation             Never
WebServer                                           Microsoft/Windows/WebServer                         Never
```

Figure 14.11: Discovering the BPA models available on DC1

In *step 5*, you use the `Invoke-BpaModel` command to run the `DirectoryServices` BPA model on `DC1`. Invoking the model produces some minimal output, like this:

```
PS C:\Foo> # 5. Running the BPA DS model on DC1
PS C:\Foo> $SB4 = {
             Invoke-BpaModel -ModelID Microsoft/Windows/DirectoryServices -Mode ALL |
               Format-Table -AutoSize
           }
PS C:\Foo> Invoke-Command -Session $BPAS -ScriptBlock $SB4

ModelId                           SubModelId Success ScanTime            ScanTimeUtcOffset Detail
-------                           ---------- ------- --------            ----------------- ------
Microsoft/Windows/DirectoryServices             True 12/04/2021 11:41:32 00:00:00          {DC1, DC1}
```

Figure 14.12: Invoking a BPA scan on DC1

To obtain the detailed results of the BPA scan, you use the `Get-BpaResult` command, as you can see in *step 6*, which produces the following output:

```
PS C:\Foo> # 6. Getting BPA results from DC1
PS C:\Foo> $SB5 = {
             Get-BpaResult -ModelID Microsoft/Windows/DirectoryServices  |
               Where-Object Resolution -ne $null|
                 Format-List -Property Problem, Resolution
           }
PS C:\Foo> Invoke-Command -Session $BPAS -ScriptBlock $SB5


Problem    : The primary domain controller (PDC) emulator operations master in this forest is not configured
             to correctly synchronize time from a valid time source.
Resolution : Set the PDC emulator master in this forest to synchronize time with a reliable external time
             source. If you have not configured a reliable time server (GTIMESERV) in the forest root domain,
             set the PDC emulator master in this forest to synchronize time with a hardware clock that is installed
             on the network (the recommended approach). You can also set the PDC emulator master in this forest to
             synchronize time with an external time server by running the
             w32tm /config /computer:DC1.Reskit.Org /manualpeerlist:time.windows.com /syncfromflags:manual /update command.
             If you have configured a reliable time server (GTIMESERV) in the forest root domain, set the PDC emulator
             master in this forest to synchronize time from the forest root domain hierarchy by running
             w32tm /config /computer:DC1.Reskit.Org /syncfromflags:domhier /update.

Problem    : Some organizational units (OUs) in this domain are not protected from accidental deletion.
Resolution : Make sure that all OUs in this domain are protected from accidental deletion.

Problem    : The directory partition DC=Reskit,DC=Org on the domain controller DC1.Reskit.Org has not been backed up
             within the last 8 days.
Resolution : To ensure that recent system state backups are available to recover Active Directory data that was recently
             added, deleted, or modified, perform daily backups of all directory partitions in your forest or keep the
             time between Active Directory backups to a maximum of 8 days.

Problem    : The directory partition CN=Configuration,DC=Reskit,DC=Org on the domain controller DC1.Reskit.Org has not
             been backed up within the last 8 days.
Resolution : To ensure that recent system state backups are available to recover Active Directory data that was recently
             added, deleted, or modified, perform daily backups of all directory partitions in your forest or keep the
             time between Active Directory backups to a maximum of 8 days.

Problem    : The directory partition CN=Schema,CN=Configuration,DC=Reskit,DC=Org on the domain controller DC1.Reskit.Org
             has not been backed up within the last 8 days.
Resolution : To ensure that recent system state backups are available to recover Active Directory data that was recently
             added, deleted, or modified, perform daily backups of all directory partitions in your forest or keep the
             time between Active Directory backups to a maximum of 8 days.

Problem    : The directory partition DC=DomainDnsZones,DC=Reskit,DC=Org on the domain controller DC1.Reskit.Org has not
             been backed up within the last 8 days.
Resolution : To ensure that recent system state backups are available to recover Active Directory data that was recently
             added, deleted, or modified, perform daily backups of all directory partitions in your forest or keep the
             time between Active Directory backups to a maximum of 8 days.

Problem    : The directory partition DC=ForestDnsZones,DC=Reskit,DC=Org on the domain controller DC1.Reskit.Org has not
             been backed up within the last 8 days.
Resolution : To ensure that recent system state backups are available to recover Active Directory data that was recently
             added, deleted, or modified, perform daily backups of all directory partitions in your forest or keep the
             time between Active Directory backups to a maximum of 8 days.

Problem    : The Active Directory Domain Services (AD DS) server role on the domain controller DC1.Reskit.Org is installed
             on a virtual machine (VM).
Resolution : Make sure that the domain controller DC1.Reskit.Org complies with the best practice guidelines that are
             described in the Help to avoid performance issues and replication and security failures in the Active
             Directory environment
```

Figure 14.13: Obtaining BPA scan results

There's more...

BPA results include details of successful and unsuccessful tests. The unsuccessful results, where BPA finds that your deployment does not implement a best practice, are the ones you most likely need to review and take action on.

In *step 6*, you retrieve the results of the BPA scan you ran in the previous step. The results show three fundamental issues:

- You have not synchronized the time on the DC holding the PDC emulator FSMO role with some reliable external source. This issue means that the time on your hosts could "wander" from the real-world time, possibly leading to issues later on. See `https://blogs.msmvps.com/acefekay/tag/pdc-emulator-time-configuration/` for more information on how to configure your DCs with a reliable time source.
- You have not backed up your AD environment. Even with multiple DCs, it is a best practice to perform regular backups. See `https://docs.microsoft.com/windows/win32/ad/backing-up-and-restoring-an-active-directory-server` for more information on backing up, and restoring, a DC.
- `DC1` is a DC you are running in a VM. While Microsoft supports such a deployment, there are some best practices you should adhere to to ensure the reliable running of your AD service. See `https://docs.microsoft.com/windows-server/identity/ad-ds/get-started/virtual-dc/virtualized-domain-controllers-hyper-v` for more details on virtualizing DCs using Hyper-V.

For a test environment, these issues are inconsequential for the most part, and you can probably ignore them. If you are using Hyper-V for test VMs, you can configure Hyper-V to update the VMs' local time, at least for the DCs you run in a VM. In a test environment, having a backup of your AD is not needed. And running a domain controller in a Hyper-V, at least for a testing environment, is not an issue with the latest, supported Windows Server versions.

Network troubleshooting

In the upcoming recipe *Checking network connectivity using Get-NetView*, you use the `Get-NetView` command to gather a large amount of network-related information to diagnose and resolve network issues. For some issues, this level of detail is fundamental in helping you to resolve network issues. But in some cases, it can be overkill. Often some simpler steps may help you resolve your more common issues or point you toward a solution.

In this recipe, you carry out some basic troubleshooting on a local `SRV1`, a domain-joined host running Windows Server 2022. A common theory is that any network problem is due to DNS (until you prove otherwise). You start this recipe by getting the host's **fully qualified domain name** (**FQDN**) and the IPv4 address of the DNS server, and then you check whether the DNS server is online.

You then use the configured DNS server to determine the names of the DCs in your domain and ensure you can reach each DC over both port 389 (LDAP) and 445 (for GPOs). Next, you test the default gateway's availability. Finally, you test the ability to reach a remote host over port 80 (HTTP) and port 443 (HTTP over SSL/TLS).

In most cases, the simple tests in this recipe, run on the afflicted host, should help you find some of the more common problems.

Getting ready

This recipe uses SRV1, a domain-joined host running Windows Server 2022. You have installed PowerShell 7 and VS Code on this host.

How to do it...

1. Getting the DNS name of this host

```
$DNSDomain = $Env:USERDNSDOMAIN
$FQDN      = "$Env:COMPUTERNAME.$DNSDomain"
```

2. Getting the DNS server address

```
$DNSHT = @{
  InterfaceAlias = "Ethernet"
  AddressFamily  = 'IPv4'
}
$DNSServers = (Get-DnsClientServerAddress @DNSHT).ServerAddresses
$DNSServers
```

3. Checking if the DNS servers are online

```
Foreach ($DNSServer in $DNSServers) {
  $TestDNS = Test-NetConnection -Port 53 -ComputerName $DNSServer
  $Result  = $TestDNS ? "Available" : ' Not reachable'
  "DNS Server [$DNSServer] is $Result"
}
```

4. Defining a search for DCs in our domain

```
$DNSRRName = "_ldap._tcp." + $DNSDomain
$DNSRRName
```

5. Getting the DC SRV records

```
$DCRRS = Resolve-DnsName -Name $DNSRRName -Type all |
    Where-Object IP4address -ne $null
$DCRRS
```

6. Testing each DC for availability over LDAP

```
ForEach ($DNSRR in $DCRRS){
  $TestDC = Test-NetConnection -Port 389 -ComputerName $DNSRR.IPAddress
  $Result  = $TestDC ? 'DC Available' : 'DC Not reachable'
  "DC [$($DNSRR.Name)]  at [$($DNSRR.IPAddress)]   $Result for LDAP"
}
```

7. Testing DC availability for SMB

```
ForEach ($DNSRR in $DCRRS){
  $TestDC = Test-NetConnection -Port 445 -ComputerName $DNSRR.IPAddress
  $Result  = $TestDC ? 'DC Available' : 'DC Not reachable'
  "DC [$($DNSRR.Name)]  at [$($DNSRR.IPAddress)]   $Result for SMB"
}
```

8. Testing default gateway

```
$NIC    = Get-NetIPConfiguration -InterfaceAlias Ethernet
$DG     = $NIC.IPv4DefaultGateway.NextHop
$TestDG = Test-NetConnection $DG
$Result  = $TestDG.PingSucceeded ? "Reachable" : ' NOT Reachable'
"Default Gateway for [$($NIC.Interfacealias) is [$DG] - $Result"
```

9. Testing a remote website using ICMP

```
$Site = "WWW.Packt.Com"
$TestIP     = Test-NetConnection -ComputerName $Site
$ResultIP   = $TestIP ? "Ping OK" : "Ping FAILED"
"ICMP to $Site - $ResultIP"
```

10. Testing a remote website using port 80

```
$TestPort80 = Test-Connection -ComputerName $Site -TcpPort 80
$Result80    = $TestPort80  ? 'Site Reachable' : 'Site NOT reachable'
"$Site over port 80   : $Result80"
```

11. Testing a remote website using port 443

```
$TestPort443 = Test-Connection -ComputerName $Site -TcpPort 443
$Result443   = $TestPort443  ? 'Site Reachable' : 'Site NOT reachable'
"$Site over port 443  : $Result443"
```

How it works...

In *step 1*, you create a variable to hold the FQDN of the host. This step creates no output.

In *step 2*, you use `Get-DNSClientServerAddress` to get the IP addresses of the DNS servers that you (or DHCP) have configured on the host. The output looks like this:

```
PS C:\Foo> # 2. Getting DNS server address
PS C:\Foo> $DNSHT = @{
             InterfaceAlias = "Ethernet"
             AddressFamily  = 'IPv4'
           }
PS C:\Foo> $DNSServers = (Get-DnsClientServerAddress @DNSHT).ServerAddresses
PS C:\Foo> $DNSServers
10.10.10.10
10.10.10.11
```

Figure 14.14: Obtaining IP addresses configured on SRV1

In *step 3*, you check whether the configured DNS servers are available, with output like this:

```
PS C:\Foo> # 3. Checking if the DNS servers are online
PS C:\Foo> Foreach ($DNSServer in $DNSServers) {
             $TestDNS = Test-NetConnection -Port 53 -ComputerName $DNSServer
             $Result  = $TestDNS ? "Available" : ' Not reachable'
             "DNS Server [$DNSServer] is $Result"
           }
DNS Server [10.10.10.10] is Available
DNS Server [10.10.10.11] is Available
```

Figure 14.15: Checking reachability of each configured DNS server

In *step 4*, you define a DNS **resource record** (**RR**) name for the SRV records registered by active DCs for a given domain. The output looks like this:

```
PS C:\Foo> # 4. Defining a search for DCs in our domain
PS C:\Foo> $DNSRRName = "_ldap._tcp." + $DNSDomain
PS C:\Foo> $DNSRRName
_ldap._tcp.RESKIT.ORG
```

Figure 14.16: Defining an RR name for DC SRV records

In *step 5*, you retrieve the SRV RRs for DCs in your domain. Each RR represents a server that can act as a DC in the Reskit.Org domain. The output of this step looks like this:

```
PS C:\Foo> # 5. Getting the DC SRV records
PS C:\Foo> $DCRRS = Resolve-DnsName -Name $DNSRRName -Type all |
             Where-Object IP4address -ne $null
PS C:\Foo> $DCRRS

Name             Type  TTL   Section    IPAddress
----             ----  ---   -------    ---------
dc1.reskit.org   A     3600  Additional 10.10.10.10
dc2.reskit.org   A     3600  Additional 10.10.10.11
```

Figure 14.17: Querying for DNS RRs for DCs

In *step 6*, you test each discovered DC for LDAP connectivity, with output like this:

```
PS C:\Foo> # 6. Testing each DC for availability over LDAP
PS C:\Foo> ForEach ($DNSRR in $DCRRS){
             $TestDC = Test-NetConnection -Port 389 -ComputerName $DNSRR.IPAddress
             $Result  = $TestDC ? 'DC Available' : 'DC Not reachable'
             "DC [$($DNSRR.Name)]  at [$($DNSRR.IPAddress)]   $Result for LDAP"
           }
DC [dc1.reskit.org]  at [10.10.10.10]   DC Available for LDAP
DC [dc2.reskit.org]  at [10.10.10.11]   DC Available for LDAP
```

Figure 14.18: Testing LDAP connectivity to domain controllers

For each host's Group Policy agent to download GPOs from a DC, the host uses an SMB connection to the SYSVOL share on the DC. In *step 7*, you check connectivity to each DC's SMB port (port 445). The output of this step looks like this:

```
PS C:\Foo> # 7. Testing DC availability for SMB
PS C:\Foo> ForEach ($DNSRR in $DCRRS){
             $TestDC = Test-NetConnection -Port 445 -ComputerName $DNSRR.IPAddress
             $Result  = $TestDC ? 'DC Available' : 'DC Not reachable'
             "DC [$($DNSRR.Name)]  at [$($DNSRR.IPAddress)]   $Result for SMB"
           }
DC [dc1.reskit.org]  at [10.10.10.10]   DC Available for SMB
DC [dc2.reskit.org]  at [10.10.10.11]   DC Available for SMB
```

Figure 14.19: Testing SMB connectivity to domain controllers

In *step 8*, you check whether your host can reach its configured default gateway. The output of this step looks like this:

```
PS C:\Foo> # 8. Testing default gateway
PS C:\Foo> $NIC    = Get-NetIPConfiguration -InterfaceAlias Ethernet
PS C:\Foo> $DG     = $NIC.IPv4DefaultGateway.NextHop
PS C:\Foo> $TestDG = Test-NetConnection $DG
PS C:\Foo> $Result  = $TestDG.PingSucceeded ? "Reachable" : ' NOT Reachable'
PS C:\Foo> "Default Gateway for [$($NIC.Interfacealias) is [$DG] - $Result"
Default Gateway for [Ethernet is [10.10.1.100] - Reachable
```

Figure 14.20: Testing the default gateway

In *step 9*, you check to see if you can reach an external Internet-based host using ICMP (aka ping). The output of this step looks like this:

```
PS C:\Foo> # 9. Testing a remote website using ICMP
PS C:\Foo> $Site = "WWW.Packt.Com"
PS C:\Foo> $TestIP     = Test-NetConnection -ComputerName $Site
PS C:\Foo> $ResultIP   = $TestIP ? "Ping OK" : "Ping FAILED"
PS C:\Foo> "ICMP to $Site - $ResultIP"
ICMP to WWW.Packt.Com - Ping OK
```

Figure 14.21: Pinging an Internet site

In *step 10*, you check to see whether you can reach the same server, via the HTTP port, port `80`, with output like this:

```
PS C:\Foo> # 10. Testing a remote website using port 80
PS C:\Foo> $TestPort80 = Test-Connection -ComputerName $Site -TcpPort 80
PS C:\Foo> $Result80    = $TestPort80  ? 'Site Reachable' : 'Site NOT reachable'
PS C:\Foo> "$Site over port 80   : $Result80"
WWW.Packt.Com over port 80   : Site Reachable
```

Figure 14.22: Testing connectivity over port 80

Finally, in *step 11*, you check to see whether you can reach the same server via HTTP over SSL/TLS, port `443`, with output like this:

```
PS C:\Foo> # 11. Testing a remote website using port 443
PS C:\Foo> $TestPort443 = Test-Connection -ComputerName $Site -TcpPort 443
PS C:\Foo> $Result443   = $TestPort443  ? 'Site Reachable' : 'Site NOT reachable'
PS C:\Foo> "$Site over port 443  : $Result443"
WWW.Packt.Com over port 443  : Site Reachable
```

Figure 14.23: Testing connectivity over port 443

There's more...

In *step 1*, you create a variable to hold the FQDN of the host. You may want to use this to ensure that your host has properly registered your host's FQDN in the DNS server. If DNS misregistration is causing problems, you may wish to adapt this script to check for correct DNS RR registration.

In *step 4*, you create a DNS RR name that you then use, in *step 5*, to query for any SRV records of that name. AD uses DNS as a locator service – each DC registers SRV records to advertise its ability to serve as a DC. The SRV record contains the FQDN name of the advertised DC. The approach taken by these two steps is similar to how any domain client finds a domain controller. The Windows Netlogon service on a DC registers all the appropriate SRV records each time the service starts, or every 24 hours. One troubleshooting technique is to use `Restart-Service` to restart the NetLogon service on each DC.

If you have a large routed network, you may wish to move the default gateway check, performed here in *step 8*, to earlier in your production version of this recipe, possibly before *step 3*. If you can't reach your default gateway and your DNS server and DCs are on different subnetworks, the earlier steps are likely to fail due to a default gateway issue.

In *step 9*, *step 10*, and *step 11*, you test connectivity to a remote host via ICMP and ports `80` and `443`. A host or an intermediate router may drop ICMP traffic, yet allow port `80/443` traffic in many cases. So just because a ping has not succeeded does not necessarily suggest a point of failure at all – it may be a deliberate feature and by design.

In some of the steps in this recipe, you used the PowerShell 7 ternary operator to construct the messages outputted by those steps. These steps provide a good example of using this operator, which you would have read about in *Chapter 2*, *Introducing PowerShell 7* in the *Exploring new operators* recipe.

Checking network connectivity using Get-NetView

`Get-NetView` is a tool that collects details about your network environment that can help you troubleshoot network issues.

The `Get-NetView` module contains a single function, `Get-NetView`. When you run the command, it pulls together a huge range of network details and creates a ZIP file containing a wealth of details about your network. By default, `Get-NetView` creates this output on our desktop.

`Get-NetView` output includes the following details:

- `Get-NetView` metadata
- The host environment (including OS, hardware, domain, and hostname)
- Physical, virtual, and container NICs
- Network configuration (including IP addresses, MAC addresses, neighbors, and IP routes)
- Physical switch configuration, including **Quality of Service** (**QoS**) policies
- Hyper-V VM configuration
- Hyper-V virtual switches, bridges, and NATs
- Windows device drivers
- Performance counters
- System and application events

The output provided by `Get-NetView`, as the above list suggests, is voluminous. To help troubleshoot a given issue, only a very small amount of the information is likely useful to you. However, if there is an issue in your network, this information is going to help you do the troubleshooting.

Getting ready

This recipe uses SRV1, a domain-joined Windows Server 2022 host. You have installed PowerShell 7 and VS Code on this host.

How to do it...

1. Finding the `Get-NetView` module on the PS Gallery

```
Find-Module -Name Get-NetView
```

2. Installing the latest version of `Get-NetView`

```
Install-Module -Name Get-NetView -Force -AllowClobber
```

3. Checking installed version of `Get-NetView`

```
Get-Module -Name Get-NetView -ListAvailable
```

4. Importing the `Get-NetView` module

```
Import-Module -Name Get-NetView -Force
```

5. Creating a new folder

```
$OF = 'C:\NetViewOutput'
New-Item -Path $OF -ItemType directory | Out-Null
```

6. Running `Get-NetView`

```
Get-NetView -OutputDirectory $OF
```

7. Viewing the output folder using `Get-ChildItem`

```
$OFF = Get-ChildItem $OF
$OFF
```

8. Viewing the output folder contents using `Get-ChildItem`

```
$Results = $OFF | Select-Object -First 1
Get-ChildItem -Path $Results
```

9. Viewing IP configuration

```
Get-Content -Path $Results\_ipconfig.txt
```

How it works...

In *step 1*, you find the `Get-NetView` module on the PowerShell Gallery. The output from this step looks like this:

```
PS C:\Foo> # 1. Finding the Get-NetView module on the PS Gallery
PS C:\Foo> Find-Module -Name Get-NetView

Version        Name          Repository   Description
-------        ----          ----------   -----------
2021.3.23.142  Get-NetView   PSGallery    Get-NetView is a tool used to simplify the
                                          collection of network configuration information.
```

Figure 14.24: Finding the Get-NetView module

In *step 2*, you install the latest version of this module, which generates no output. In *step 3*, you check which version (or versions) of the `Get-NetView` module are on `SRV1`, with output like this:

```
PS C:\Foo> # 3. Checking installed version of Get-NetView
PS C:\Foo> Get-Module -Name Get-NetView -ListAvailable

ModuleType Version       PreRelease Name          ExportedCommands
---------- -------       ---------- ----          ----------------
Script     1.0                      Get-NetView   Get-NetView
Script     2021.3.23.142            Get-NetView   Get-NetView
```

Figure 14.25: Checking the installed version(s) of Get-NetView

In *step 4*, you import the `Get-NetView` module. In *step 5*, you create a new folder on the `C:\` drive to hold the output `Get-NetView` generates. These two steps produce no output.

In *step 6*, you run `Get-NetView`. At each step taken, the command logs some network information and outputs a running commentary. This command generates a lot of console output, a subset of which looks like this:

```
PS C:\Foo> # 6. Running Get-NetView
PS C:\Foo> Get-NetView -OutputDirectory $OF
Transcript started, output file is C:\NetViewOutput\msdbg.SRV1\Get-NetView.log
(   669 ms) Get-Service "*" | Sort-Object Name | Format-Table -AutoSize
(   850 ms) Get-Service "*" | Sort-Object Name | Format-Table -Property * -AutoSize
                                    Queueing tasks...
( 2,892 ms) Get-ChildItem HKLM:\SYSTEM\CurrentControlSet\Services\vmsmp -Recurse

... trace details - omitted

Diagnostics Data:
-----------------
Get-NetView
Version: 2021.3.23.142
SHA1:  CBEC2292DF72E2C2BA74FC33EA458ADD6BCEED14

C:\NetViewOutput\msdbg.SRV1
Size:    487.39 MB
Dirs:    105
Files:   910

Execution Time:
---------------
1 Min 46 Sec

Transcript stopped, output file is C:\NetViewOutput\msdbg.SRV1\Get-NetView.log
C:\NetViewOutput\msdbg.SRV1-2021.04.13_12.28.53.zip
Size:    36.87 MB
```

Figure 14.26: Running Get-NetView

In *step 7*, you view the output folder to view the files created by `Get-NetView`, with output like this:

```
PS C:\Foo> # 7. Viewing the output folder using Get-ChildItem
PS C:\Foo> $OFF = Get-ChildItem $OF
PS C:\Foo> $OFF

    Directory: C:\NetViewOutput

Mode          LastWriteTime     Length Name
----          -------------     ------ ----
d----   13/04/2021    16:24            msdbg.SRV1
-a---   13/04/2021    16:24   38597360 msdbg.SRV1-2021.04.13_04.20.33.zip
```

Figure 14.27: Viewing the Get-NetView output folder

In *step 8*, you view the detailed information created by `Get-NetView`, with output like this:

```
PS C:\Foo> # 8. Viewing the output folder contents using Get-ChildItem
PS C:\Foo> $Results = $OFF | Select-Object -First 1
PS C:\Foo> Get-ChildItem -Path $Results

    Directory: C:\NetViewOutput\msdbg.SRV1

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d----          13/04/2021    16:21                _Localhost
d----          13/04/2021    16:24                _Logs
d----          13/04/2021    16:23                802.1X
d----          13/04/2021    16:22                Counters
d----          13/04/2021    16:23                NetIp
d----          13/04/2021    16:23                NetNat
d----          13/04/2021    16:23                NetQoS
d----          13/04/2021    16:22                NetSetup
d----          13/04/2021    16:24                Netsh
d----          13/04/2021    16:22                NIC.14.Local Area Connection.TAP-Windows Adapter V9
d----          13/04/2021    16:22                NIC.6.Ethernet 2.Microsoft Hyper-V Network Adapter #2
d----          13/04/2021    16:22                NIC.7.Ethernet.Microsoft Hyper-V Network Adapter
d----          13/04/2021    16:21                NIC.Hidden
d----          13/04/2021    16:24                SMB
d----          13/04/2021    16:22                VMSwitch.Detail
-a---          13/04/2021    16:22           2233 _advfirewall.txt
-a---          13/04/2021    16:21           1963 _arp.txt
-a---          13/04/2021    16:21           3949 _ipconfig.txt
-a---          13/04/2021    16:22          13813 _netstat.txt
-a---          13/04/2021    16:22             69 _nmbind.txt
-a---          13/04/2021    16:21           7417 Environment.txt
-a---          13/04/2021    16:21          12897 Get-ComputerInfo.txt
-a---          13/04/2021    16:21          49611 Get-NetAdapter.txt
-a---          13/04/2021    16:21           4417 Get-NetAdapterStatistics.txt
-a---          13/04/2021    16:21          12269 Get-NetIpAddress.txt
-a---          13/04/2021    16:21            310 Get-NetLbfoTeam.txt
-a---          13/04/2021    16:21           1162 Get-NetOffloadGlobalSetting.txt
-a---          13/04/2021    16:21            944 Get-VMNetworkAdapter.txt
-a---          13/04/2021    16:21            832 Get-VMSwitch.txt
-a---          13/04/2021    16:21            354 Powercfg.txt
-a---          13/04/2021    16:21           1312 Verifier.txt
```

Figure 14.28: Viewing the Get-NetView output folder contents

In *step 9*, you examine the IP configuration generated by `Get-NetView`, which looks like this:

```
PS C:\Foo> # 9. Viewing IP configuration
PS C:\Foo> Get-Content -Path $Results\_ipconfig.txt

Administrator @ SRV1:
PS C:\NetViewOutput> ipconfig

Windows IP Configuration

Ethernet adapter Ethernet:

   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . : fe80::286e:7f7f:a77c:79c1%7
   IPv4 Address. . . . . . . . . . . : 10.10.10.50
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 10.10.10.254

  Unknown adapter Local Area Connection:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :
   Description . . . . . . . . . . . : TAP-Windows Adapter V9
   Physical Address. . . . . . . . . : 00-FF-B6-68-E1-5D
   DHCP Enabled. . . . . . . . . . . : Yes
   Autoconfiguration Enabled . . . . : Yes
```

Figure 14.29: Viewing IP configuration

There's more...

In *step 3*, you check the version(s) of the `Get-NetView` module on your system. In this case, you can see version 1.0, which shipped with the edition of Windows Server (running on `SRV1`) as well as a later version you obtained from the PowerShell Gallery. As always, you may see a later version of this module on the PowerShell Gallery. In this step, you install the module rather than updating it. You cannot upgrade the inbox versions of any module unless and until you install it first explicitly. Then, going forward, you can use `Update-Module` to obtain any updates.

In *step 7*, you view the files outputted by `Get-NetView`. As you can see, there is a folder and a ZIP archive file in the output folder. `Get-NetView` adds all the network information to separate files in the output folder and then compresses all that information into a single archive file you can send to a network technician for resolution.

In *step 9*, you view one of the many bits of information created by `Get-NetView`. In this case, you look at the IP configuration information, including the IP address, subnet mask, and default gateway. Interestingly, the developers used `ipconfig.exe` to create this information without using the `/all` switch. This means you cannot see from this output the configured DNS server IP addresses.

Exploring PowerShell script debugging

PowerShell, both Windows PowerShell and PowerShell 7, contains some great debugging features. Using these features makes it easier to find and remove errors in your scripts. You can set breakpoints in a script – when you run your script, PowerShell stops execution at the breakpoint. For example, you can set a breakpoint to stop at a particular line, any time your script writes to a particular variable, or any time PowerShell calls a particular cmdlet.

When PowerShell encounters a breakpoint, it suspends processing and presents you with a debugging prompt, as you see in this recipe. You can then examine the results so far and run additional commands to ensure your script produces the output and results you expect. If your script adds a user to the AD and then performs an action on that user (adding the user to a group, for example), you could stop the script just after the `Add-ADUser` command completes. At that point, you could use `Get-ADUser` or other commands to check whether your script has added the user as you expected. You can then use the `continue` statement to resume your script. PowerShell then resumes running your script until it either completes or hits another breakpoint.

Getting ready

This recipe uses `SRV1`, a domain-joined host controller in the Reskit.Org domain. You have installed PowerShell 7 and VS Code on this host.

How to do it...

1. Creating a script to debug

```
$SCR = @'
# Script to illustrate breakpoints
Function Get-Foo1 {
  param ($J)
  $K = $J*2           # NB: line 4
  $M = $K             # NB: $m written to
  $M
  $BIOS = Get-CimInstance -Class Win32_Bios
}
Function Get-Foo {
  param ($I)
  (Get-Foo1 $I)      # Uses Get-Foo1
}
Function Get-Bar {
  Get-Foo (21)}
# Start of ACTUAL script
```

```
"In Breakpoint.ps1"
"Calculating Bar as [{0}]" -f (Get-Bar)
'@
```

2. Saving the script

```
$ScrFile = 'C:\Foo\Breakpoint.ps1'
$SCR | Out-File -FilePath $ScrFile
```

3. Executing the script

```
& $ScrFile
```

4. Adding a breakpoint at a line in the script

```
Set-PSBreakpoint -Script $ScrFile -Line 4 |  # breaks at line 4
    Out-Null
```

5. Adding a breakpoint on the script whenever the script uses a specific command

```
Set-PSBreakpoint -Script $SCRFile -Command "Get-CimInstance" |
  Out-Null
```

6. Adding a breakpoint when the script writes to `$M`

```
Set-PSBreakpoint -Script $SCRFile -Variable M -Mode Write |
  Out-Null
```

7. Viewing the breakpoints set in this session

```
Get-PSBreakpoint | Format-Table -AutoSize
```

8. Running the script – until the first breakpoint is hit

```
& $ScrFile
```

9. Viewing the value of `$J` from the debug console

```
$J
```

10. Viewing the value of `$K` from the debug console

```
$K
```

11. Continuing script execution from the DBG prompt until the next breakpoint

```
continue
```

12. Continuing script execution from the DBG prompt until the execution of `Get-CimInstance`

```
continue
```

13. Continuing script execution from the DBG prompt until the end

```
continue
```

How it works...

In *step 1*, you create a script to allow you to examine PowerShell script debugging. In *step 2*, you save this file to the `C:` drive. These steps create no output.

In *step 3*, you execute the script, which produces some output to the console, like this:

```
PS C:\Foo> # 3. Executing the script
PS C:\Foo> & $ScrFile
In Breakpoint.ps1
Calculating Bar as [42]
```

Figure 14.30: Executing the script

In *step 4*, you set a breakpoint in the script at a specific line. In *step 5*, you set another breakpoint, this time whenever your script calls a specific command (`Get-CimInstance`). In *step 6*, you set a breakpoint to stop whenever you write to a specific variable (`$M`). Setting these three breakpoints produces no output (since you piped the command output to `Out-Null`).

In *step 7*, you view the breakpoints you have set thus far in the current PowerShell session. The output looks like this:

```
PS C:\Foo> # 7. Viewing the breakpoints set in this session
PS C:\Foo> Get-PSBreakpoint | Format-Table -AutoSize

ID Script         Line Command         Variable Action
-- ------         ---- -------         -------- ------
 1 Breakpoint.ps1    4
 2 Breakpoint.ps1      Get-CimInstance
 3 Breakpoint.ps1                      M
```

Figure 14.31: Viewing breakpoints

Having set three breakpoints in the script, in *step 8,* you run the script. PowerShell stops execution when it reaches the first breakpoint (in *line 4* of the script), with output like this:

```
PS C:\Foo> # 8. Running the script - until the first breakpoint is hit
PS C:\Foo> & $ScrFile
In Breakpoint.ps1
Entering debug mode. Use h or ? for help.

Hit Line breakpoint on 'C:\Foo\Breakpoint.ps1:4'

At C:\Foo\Breakpoint.ps1:4 char:3
+   $K = $J*2           # NB: line 4
+   ~~~~~~~~~
[DBG]: PS C:\Foo>>
```

Figure 14.32: Running the script until PowerShell hits the first breakpoint

From the DBG prompt, you can enter any PowerShell command, for example, to view the value of `$J`, which you do in *step 9*. It looks like this:

```
[DBG]: PS C:\Foo>> # 9. Viewing the value of $J from the debug console
[DBG]: PS C:\Foo>> $j
21
```

Figure 14.33: Viewing the value of the $J variable

In *step 10*, you view the value of the `$K` variable. Since PowerShell stopped execution before it could write a value to this variable, this step displays no value, as you can see here:

```
[DBG]: PS C:\Foo>> # 10. Viewing the value of $K from the debug console
[DBG]: PS C:\Foo>> $K
[DBG]: PS C:\Foo>>
```

Figure 14.34: Viewing the value of the $K variable

To continue the execution of the script, in *step 11*, you type `continue`, which generates output like this:

```
[DBG]: PS C:\Foo>> # 11. Continuing script execution from the DBG prompt until the next breakpoint
[DBG]: PS C:\Foo>> continue
Hit Variable breakpoint on 'C:\Foo\Breakpoint.ps1:$M' (Write access)

At C:\Foo\Breakpoint.ps1:5 char:3
+   $M = $K             # NB: $m written to
+   ~~~~~~~~
[DBG]: PS C:\Foo>>
```

Figure 14.35: Running the script until PowerShell hits the second breakpoint

As in the previous step, you can examine the script's actions thus far. You continue the script, in *step 12*, by typing `continue` like this:

```
[DBG]: PS C:\Foo>> # 12. Continuing script execution from the DBG prompt until the execution of Get-CimInstance
[DBG]: PS C:\Foo>> continue
Hit Command breakpoint on 'C:\Foo\Breakpoint.ps1:Get-CimInstance'

At C:\Foo\Breakpoint.ps1:7 char:3
+   $bios = Get-CimInstance -Class Win32_Bios
+   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
[DBG]: PS C:\Foo>>
```

Figure 14.36: Running the script until PowerShell hits the third and final breakpoint

In *step 13*, you continue running the script, which now completes, with output like this:

```
[DBG]: PS C:\Foo>> # 13. Continuing script execution from the DBG prompt until the end
[DBG]: PS C:\Foo>> continue
Calculating Bar as [42]
PS C:\Foo>
```

Figure 14.37: Running the script until the end of the script

There's more...

In *step 4*, you set a line breakpoint, instructing PowerShell to stop execution once it reaches a specific line (and column) in our script. In *step 5*, you set a command breakpoint, telling PowerShell to break whenever the script invokes a specific command, in this case `Get-CimInstance`. In *step 6*, you set a variable breakpoint – you tell PowerShell to stop whenever your script reads from or writes to a specific variable.

In *step 8*, you run this instrumented script, which breaks at the first breakpoint. From the debug console, as you see in *step 9*, you can view the value of any variable, such as `$J`. When debugging, you have to ask yourself if the value you see is the value you expected to see. In *step 10*, you also view the value of `$K`. Since PowerShell has not yet processed the assignment stage, this variable has no value.

In *step 11*, you continue execution until PowerShell hits the second breakpoint. As before, you could examine the values of key variables.

After continuing again, your script hits the final breakpoint, just before PowerShell invokes `Get-CimInstance` and assigns the output to `$BIOS`. From the debug console, you could invoke the cmdlet to check what the output would be.

Finally, in *step 13*, you continue to complete the execution of the script. Note that you now see the normal PowerShell prompt.

If you have an especially complex script, you could set the breakpoints using another script similar to this recipe. You would use `Set-PSBreakpoint` whenever your script executes important commands, writes to specific variables, or reaches a key line in the script. You could later use that script file when you perform debugging, after making changes to the underlying script.

15

Managing with Windows Management Instrumentation

In this chapter, we cover the following recipes:

- Exploring WMI in Windows
- Exploring WMI namespaces
- Exploring WMI classes
- Obtaining local and remote WMI objects
- Using WMI methods
- Managing WMI events
- Implementing permanent WMI eventing

Introduction

Windows Management Instrumentation (**WMI**) is a Windows component you use to help manage Windows systems. WMI is Microsoft's proprietary implementation of the **Web-Based Enterprise Management** (**WBEM**) standard. WBEM is an open standard promulgated by the **Distributed Management Task Force** (**DMTF**) that aims to unify the management of distributed computing environments by utilizing standards-based internet technologies.

In addition to WMI for Windows, there are other implementations of WBEM, including OpenWBEM. You can read more about the DMTF and WBEM at `https://www.dmtf.org/about/faq/wbem_faq`, and check out OpenWBEM over at `http://openwbem.sourceforge.net/`.

Microsoft first introduced WMI as an add-on component for Windows NT 4. They later integrated WMI as an essential component of the Windows client, from Windows XP onward, and Windows Server versions since Windows Server 2000. Subsequently, several feature teams inside the Windows group made heavy use of WMI. Both the storage and networking stacks within Windows, for example, use WMI. Many of the cmdlets, such as `Get-NetAdapter`, are based directly on WMI.

The WBEM standards originally specified that the components of WMI communicate using HTTP. To improve performance, Microsoft instead implemented the components of WMI to communicate using the **Common Object Model** (**COM**), which was a popular development technology at that time. WMI itself remains based on COM. The PowerShell cmdlets use .NET to access the features of WMI.

Windows PowerShell 1.0 came with a set of cmdlets that you could use to access WMI. These were basic cmdlets that work in all subsequent versions of Windows PowerShell. However, there is no support for these cmdlets directly in PowerShell 7. You can invoke older WMI cmdlet-based scripts on a machine using PowerShell remoting if you really need to.

With PowerShell V3, Microsoft did some significant work on WMI, resulting in a new module, `CimCmdlets`. There were several great reasons behind both the new module and the associated updates to some of the WMI internals to assist developers. In this chapter, you make use of the `CimCmdlets` module to access the features of WMI. You can read more about why the team decided to build new cmdlets in a blog post at `https://devblogs.microsoft.com/powershell/introduction-to-cim-cmdlets/`. If you have scripts that use the older WMI cmdlets, consider upgrading them to use the later `CimCmdlets` module instead. These newer cmdlets are faster, which is always a nice benefit.

WMI architecture

WMI is a complex subject. Before looking at `CimCmdlets` and what you can do with them, it is useful to understand the WMI architecture within Windows. The runtime architecture of WMI is the same in Windows 10 and Windows Server 2022. The following diagram shows the WMI conceptual architecture:

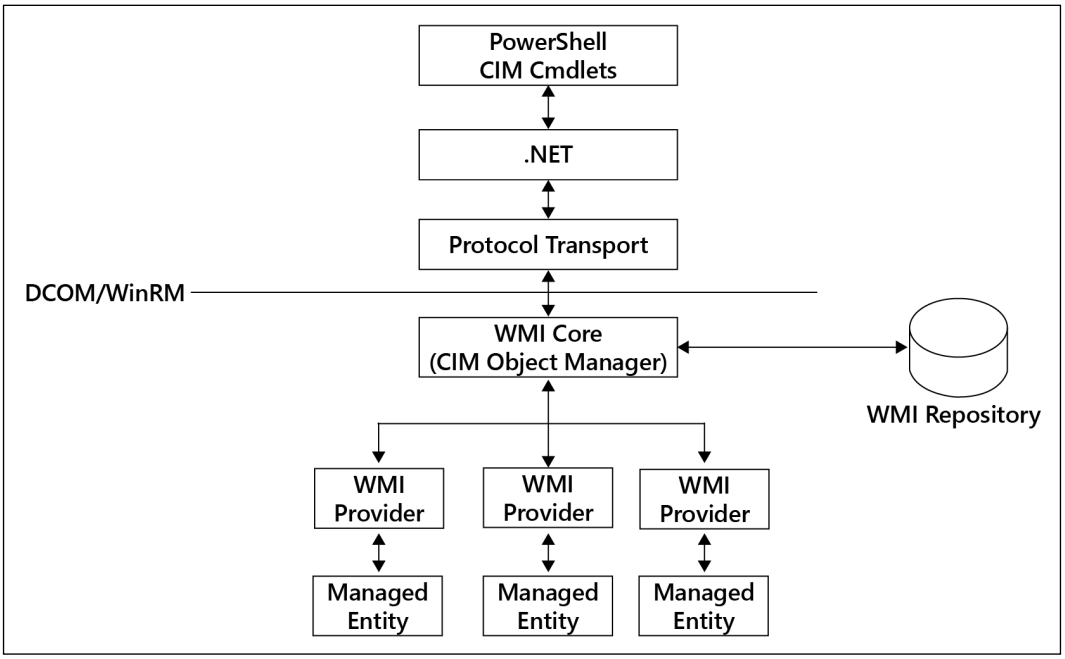


Figure 15.1: WMI architecture

As an IT pro, you use the `CimCmdlets` module's **cmdlets** to access WMI. These cmdlets use **.NET** to communicate, via the **protocol transport**, with the **WMI core** and the **CIM Object manager** on a local or remote host. The transport protocols include DCOM and WSMan. The core components of WMI, particularly the CIM Object Manager (CIMOM), are COM components you find on every Windows host.

The CIMOM stores information in the **WMI repository**, sometimes referred to as the **Common Information Model** (**CIM**) or the **CIM database**. This database is, in effect, a subset of an ANSI-SQL database. The CIM cmdlets enable you to access the information within the database. The CIM database organizes the information into namespaces of classes. .NET also uses namespaces to organize classes. However, .NET classes include the namespace name as part of the class name. For example, you can create a new email message using a .NET class, `System.Net.Mail.Mailmessage`, where the namespace is `System.Net.Mail`. Thus, the namespace is included in the full class name. With WMI, namespace names and class names are separate and supplied to the CIM cmdlets using different parameters.

WMI classes contain data instances that hold relevant management data. For example, the WMI namespace `Root\CimV2` contains the class `Win32_Share`. Each instance within this class represents one of the SMB shares within your host. With PowerShell, you would normally use the SMB cmdlets to manage SMB shares. There is useful information contained in other WMI classes for which there is no cmdlet support.

When you retrieve instances of, for example, the `Win32_Share` class, .NET gets the instance information and returns it in a .NET wrapper object. Strictly speaking, a WMI object instance is an instance of a specialized .NET class with data returned in WMI. For this reason, you treat WMI objects using the same methods you employ with other .NET objects.

Many WMI classes have methods you can invoke that perform some operation on either a given WMI instance or statically based on the class. The `Win32_Share` class, for example, has a static `Create()` method you can use to create a new share. Each instance of that class has a dynamic `Delete()` method, which deletes the SMB share.

An important architectural feature of WMI is the **WMI provider**. A WMI provider is an add-on to WMI that implements WMI classes inside a given host. The Win32 WMI provider, for example, implements hundreds of WMI classes, including `Win32_Share` and `Win32_Bios`. A provider also implements class methods and class events. For example, the Win32 provider is responsible for performing both the `Delete()` method to delete an SMB share and the `Create()` method to create a new SMB share.

In production, you are more likely to manage SMB shares using the SMB cmdlets and less likely to use WMI directly. Since SMB shares should be very familiar, they make a great example to help you understand more about WMI, and this chapter's recipes make use of the class.

WMI and WMI providers offer a rich eventing system. WMI and WMI provider classes can implement events to which you can subscribe. When the event occurs, the eventing system notifies you, and you can take some action to handle the event occurrence. For example, you can register for a WMI event that occurs when someone changes the membership of an AD group. When this happens, WMI eventing allows you to take some actions, such as emailing a security administrator informing them of the group's membership change. WMI also implements permanent WMI events. This feature allows you to configure WMI to trap and handle events with no active PowerShell session running. Permanent events even survive a reboot of the host, extremely powerful in lights-off environments.

There is a lot more detail about WMI than can fit in this chapter. For more details about WMI and how you can interact with it in more detail, consult with Richard Siddaway's *PowerShell and WMI* book (© Manning, Aug 2012 — `https://www.manning.com/books/powershell-and-wmi`). Richard's book goes into great detail about WMI, but all the code samples use the older WMI cmdlets. You should be able to translate the samples to use the CIM cmdlets. A key value of the book is the discussion of WMI features and how they work. The basic functioning of WMI has not changed significantly since that book was published.

Exploring WMI in Windows

Windows installs WMI during the installation of the OS. The installation process puts most of the WMI components, including the repository, tools, and the providers, into a folder, `C:\Windows\System32\WBEM`.

Inside a running Windows host, WMI runs as a service, the `winmgmt` service (`winmgmt.exe`). Windows runs this service inside a shared service process (`svchost.exe`). In the early versions of WMI in Windows, WMI loaded all the WMI providers into the `winmgmt` service. The failure of a single provider could cause the entire WMI service to fail. Later, with Windows XP and beyond, Microsoft improved WMI to load providers in a separate process, `WmiPrvSE.exe`.

In this recipe, you examine the contents of the WBEM folder, the WMI service, and runtime components of WMI.

Getting ready

This recipe uses `SRV1`, a domain-joined host. You have installed PowerShell 7 and VS Code on this host.

How to do it...

1. Viewing the WBEM folder

```
$WBEMFOLDER = "$Env:windir\system32\wbem"
Get-ChildItem -Path $WBEMFOLDER |
  Select-Object -First 20
```

2. Viewing the WMI repository folder

```
Get-ChildItem -Path $WBEMFOLDER\Repository
```

3. Viewing the WMI service details

```
Get-Service -Name Winmgmt  |
  Format-List -Property *
```

4. Getting process details

```
$S = tasklist.exe /svc /fi "SERVICES eq winmgmt" |
       Select-Object -Last 1
$P = [int] ($S.Substring(30,4))
Get-Process -Id $P
```

5. Examining DLLs loaded by the WMI service process

```
Get-Process -Id $P |
  Select-Object -ExpandProperty modules |
    Where-Object ModuleName -match 'wmi' |
      Format-Table -Property FileName, Description, FileVersion
```

6. Discovering WMI providers

```
Get-ChildItem -Path $WBEMFOLDER\*.dll |
  Select-Object -ExpandProperty Versioninfo |
    Where-Object FileDescription -match 'prov' |
      Format-Table -Property Internalname,
                             FileDescription,
                             ProductVersion
```

7. Examining the `WmiPrvSE` process

```
Get-Process -Name WmiPrvSE
```

8. Finding the WMI event log

```
$Log = Get-WinEvent -ListLog *wmi*
$Log
```

9. Looking at the event types in the WMI log

```
$Events = Get-WinEvent -LogName $Log.LogName
$Events | Group-Object -Property LevelDisplayName
```

10. Examining WMI event log entries

```
$Events |
  Select-Object -First 5 |
    Format-Table -Wrap
```

11. Viewing executable programs in WBEM folder

```
$Files = Get-ChildItem -Path $WBEMFOLDER\*.exe
"{0,15}  {1,-40}" -f 'File Name','Description'
Foreach ($File in $Files){
 $Name = $File.Name
 $Desc = ($File |
          Select-Object -ExpandProperty VersionInfo).FileDescription
"{0,15}  {1,-40}" -f $Name,$Desc
}
```

12. Examining the `CimCmdlets` module

```
Get-Module -Name CimCmdlets |
  Select-Object -ExcludeProperty Exported*
    Format-List -Property *
```

13. Finding cmdlets in the `CimCmdlets` module

```
Get-Command -Module CimCmdlets
```

14. Examining the .NET type returned from `Get-CimInstance`

```
Get-CimInstance -ClassName Win32_Share | Get-Member
```

How it works...

The WMI service and related files are in the Windows installation folder's `System32\WBEM` folder. In *step 1*, you view part of the contents of that folder, with output like this:

```
PS C:\Foo> # 1. Viewing the WBEM folder
PS C:\Foo> $WBEMFOLDER = "$Env:windir\system32\wbem"
PS C:\Foo> Get-ChildItem -Path $WBEMFOLDER |
             Select-Object -First 20


    Directory: C:\Windows\System32\wbem


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        11/04/2021    06:42                AutoRecover
d-----        27/02/2021    11:26                en
d-----        18/03/2021    13:42                en-US
d-----        27/02/2021    10:15                Logs
d-----        17/03/2021    18:15                MOF
d-----        17/04/2021    10:40                Performance
d-----        17/04/2021    10:33                Repository
d-----        27/02/2021    10:15                tmf
d-----        27/02/2021    10:15                xml
-a----        27/02/2021    10:09           2852 aeinv.mof
-a----        27/02/2021    11:26          17510 AgentWmi.mof
-a----        27/02/2021    10:10            693 AgentWmiUninstall.mof
-a----        27/02/2021    10:08            852 appbackgroundtask_uninstall.mof
-a----        27/02/2021    10:08          65536 appbackgroundtask.dll
-a----        27/02/2021    10:08           2902 appbackgroundtask.mof
-a----        27/02/2021    10:10           1112 AttestationWmiProvider_Uninstall.mof
-a----        27/02/2021    10:10           3376 AttestationWmiProvider.mof
-a----        27/02/2021    10:09           1724 AuditRsop.mof
-a----        27/02/2021    10:09           1092 authfwcfg.mof
-a----        27/02/2021    10:09          12120 bcd.mof
```

Figure 15.2: Examining the WBEM folder

WMI stores the CIM repository in a separate folder. In *step 2*, you examine the files that make up the database, with output like this:

```
PS C:\Foo> # 2. Viewing the WMI repository folder
PS C:\Foo> Get-ChildItem -Path $WBEMFOLDER\Repository

    Directory: C:\Windows\System32\wbem\Repository

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a---          17/04/2021    10:38        5881856 INDEX.BTR
-a---          16/04/2021    10:41         106032 MAPPING1.MAP
-a---          19/04/2021    05:00         106032 MAPPING2.MAP
-a---          15/04/2021    02:48         106032 MAPPING3.MAP
-a---          17/04/2021    10:38       31891456 OBJECTS.DATA
```

Figure 15.3: Examining the files making up the CIM repository

In *step 3*, you use `Get-Service` to examine the WMI service, with output that looks like this:

```
PS C:\Foo> # 3. Viewing the WMI service details
PS C:\Foo> Get-Service -Name Winmgmt  |
             Format-List -Property *

UserName            : localSystem
Description         : Provides a common interface and object model to access management information about operating
                      system, devices, applications and services. If this service is stopped, most Windows-based
                      software will not function properly. If this service is disabled, any services that explicitly
                      depend on it will fail to start.
DelayedAutoStart    : False
BinaryPathName      : C:\Windows\system32\svchost.exe -k netsvcs -p
StartupType         : Automatic
Name                : Winmgmt
RequiredServices    : {RPCSS}
CanPauseAndContinue : True
CanShutdown         : True
CanStop             : True
DisplayName         : Windows Management Instrumentation
DependentServices   : {UALSVC, HgClientService}
MachineName         : .
ServiceName         : Winmgmt
ServicesDependedOn  : {RPCSS}
StartType           : Automatic
ServiceHandle       : Microsoft.Win32.SafeHandles.SafeServiceHandle
Status              : Running
ServiceType         : Win32OwnProcess, Win32ShareProcess
Site                :
Container           :
```

Figure 15.4: Viewing the WMI service

In *step 4*, you examine the Windows process that runs the WMI service, with output like this:

```
PS C:\Foo> # 4. Getting process details
PS C:\Foo> $S = tasklist.exe /svc /fi "SERVICES eq winmgmt" |
                Select-Object -Last 1
PS C:\Foo> $P = [int] ($S.Substring(30,4))
PS C:\Foo> Get-Process -Id $P

 NPM(K)    PM(M)      WS(M)     CPU(s)      Id  SI ProcessName
 ------    -----      -----     ------      --  -- -----------
     15     8.43      17.91      10.97    3528   0 svchost
```

Figure 15.5: Viewing the WMI service

In *step 5*, you look at the DLLs loaded by the WMI service process, with the following output:

```
PS C:\Foo> # 5. Examining DLLs loaded by the WMI service process
PS C:\Foo> Get-Process -Id $P |
             Select-Object -ExpandProperty modules |
               Where-Object ModuleName -match 'wmi' |
                 Format-Table -Property FileName, Description, FileVersion

FileName                              Description     FileVersion
--------                              -----------     -----------
c:\windows\system32\wbem\wmisvc.dll   WMI             10.0.20303.1 (WinBuild.160101.0800)
C:\Windows\SYSTEM32\WMICLNT.dll       WMI Client API  10.0.20303.1 (WinBuild.160101.0800)
C:\Windows\system32\wbem\wmiutils.dll WMI             10.0.20303.1 (WinBuild.160101.0800)
C:\Windows\system32\wbem\wmiprvsd.dll WMI             10.0.20303.1 (WinBuild.160101.0800)
```

Figure 15.6: Viewing the DLLs loaded by the WMI service process

Each WMI provider is a DLL that the WMI service can use. In *step 6*, you look at the WMI providers on SRV1, with output like this:

```
PS C:\Foo> # 6. Discovering WMI providers
PS C:\Foo> Get-ChildItem -Path $WBEMFOLDER\*.dll |
             Select-Object -ExpandProperty Versioninfo |
               Where-Object FileDescription -match 'prov' |
                 Format-Table -Property Internalname,
                                        FileDescription,
                                        ProductVersion

InternalName                FileDescription                                    ProductVersion
------------                ---------------                                    --------------
cimwin32.dll                WMI Win32 Provider                                 10.0.20303.1
DhcpServerPsProvider.dll    DHCP WMIv2 Provider                                10.0.20303.1
DMWmiBridgeProv.dll         DM WMI Bridge Provider                             10.0.20303.1
DMWmiBridgeProv1.dll        DM WMI Bridge Provider                             10.0.20303.1
DnsClientPsProvider.dll     DNS Client WMIv2 Provider                          10.0.20303.1
DnsServerPsProvider.dll     DNS WMIv2 Provider                                 10.0.20303.1
dsprov.dll                  WMI DS Provider                                    10.0.20303.1
"EventTracingManagement.dll" WMI Provider for ETW                              10.0.20303.1
ipmiprr.dll                 IPMI Provider Resource                             10.0.20303.1
IPMIPRV.dll                 WMI IPMI PROVIDER                                  10.0.20303.1
MDMAppProv.dll              MDM Application Provider                           10.0.20303.1
MDMSettingsProv.dll         MDM Settings Provider                              10.0.20303.1
mgmtprovider.dll            Server Manager Managment Provider                  10.0.20303.1
mistreamprov.dll            SIL Composable Streams Provider                    10.0.20303.1
DtcCIMProvider.dll          DTC WMIv2 Provider                                 10.0.20303.1
msiprov.dll                 WMI MSI Provider                                   10.0.20303.1
MspsProv.exe                Msps Prov                                          10.0.20303.1
mttmprov.dll                Mangement Tools Task Manager CIM Provider          10.0.20303.1
NCObjAPI                    Non-COM WMI Event Provision APIs                   10.0.20303.1
ndisimplatwmi.DLL           NDIS IM Platform WMI Provider                      10.0.20303.1
NetAdapter.dll              Network Adapter WMI Provider                       10.0.20303.1
netdacim.dll                Microsoft Direct Access WMI Provider               10.0.20303.1
NetEventPacketCapture.dll   NetEvent Packet Capture Provider                   10.0.20303.1
NetNat.dll                  Windows NAT WMI Provider                           10.0.20303.1
netnccim.dll                DA Network Connectivity WMI Provider               10.0.20303.1
NetPeerDistCim.dll          BranchCache WMI Provider                           10.0.20303.1
netswitchteamcim.DLL        VM Switch Teaming WMI Provider                     10.0.20303.1
NetTCPIP.dll                TCPIP WMI Provider                                 10.0.20303.1
nlmcim.dll                  Network Connection Profiles WMI Provider           10.0.20303.1
ntevt.dll                   WMI Event Log Provider                             10.0.20303.1
PLATID.DLL                  WMIv2 Provider to Retrieve Platform Identifiers    10.0.20303.1
PrintManagementProvider.DLL Print WMI Provider                                 10.0.20303.1
RacWmiProv.dll              Reliability Metrics WMI Provider                   10.0.20303.1
RAMgmtPSProvider.dll        RAMgmtPSProvider                                   10.0.20303.1
regprov.dll                 Registry Provider                                  10.0.20303.1
schedprov.dll               Task Scheduler WMIv2 Provider                      10.0.20303.1
SDNDiagnosticsProvider.dll  SDNDiagnosticsProvider                             10.0.20303.1
servercompprov.dll          Windows Server Feature WMI Provider                10.0.20303.1
sllprovider.dll             Server License Logging CIM Provider                10.0.20303.1
vdswmi.dll                  WMI Provider for VDS                               10.0.20303.1
viewprov.dll                WMI View Provider                                  10.0.20303.1
VpnClientPsProvider.dll     VPN Client WMIv2 Provider                          10.0.20303.1
VSSPROV.DLL                 WMI Provider for VSS                               10.0.20303.1
WdacWmiProv.dll             WDAC WMI Providers                                 10.0.20303.1
Win32_EncryptableVolume.DLL Win32_Encryptable Volume Provider                  10.0.20303.1
Win32_Tpm.DLL               TPM WMI Provider                                   10.0.20303.1
WMIPCIMA.dll                WMI Win32Ex Provider                               10.0.20303.1
wmipdfs.dll                 WMI DFS Provider                                   10.0.20303.1
WMIPDskQ.dll                WMI Provider for Disk Quota Information            10.0.20303.1
WbemPerfClass.dll           WbemPerf V2 Class Provider                         10.0.20303.1
WbemPerfInst.dll            WbemPerf V2 Instance Provider                      10.0.20303.1
wmipicmp.dll                WMI ICMP Echo Provider                             10.0.20303.1
WMIPIpRt.dll                WBEM Provider for IP4 Routes                       10.0.20303.1
wmipjobj.dll                WMI Windows Job Object Provider                    10.0.20303.1
WMIPSess.dll                WMI Provider for Sessions and Connections          10.0.20303.1
```

Figure 15.7: Viewing WMI provider DLLs

In *step 7*, you examine the `WmiPrvSE` process, with output like this:

```
PS C:\Foo> # 7. Examining the WmiPrvSE process
PS C:\Foo> Get-Process -Name WmiPrvSE

 NPM(K)  PM(M)   WS(M)   CPU(s)     Id  SI ProcessName
 ------  -----   -----   ------     --  -- -----------
     18   7.29   15.71     3.45   1876   0 WmiPrvSE
```

Figure 15.8: Viewing the WmiPrvSE process

Like other Windows services, WMI logs events to an event log, which can help troubleshoot WMI issues. In *step 8*, you look for any WMI-related event logs with output like this:

```
PS C:\Foo> # 8. Finding the WMI event log
PS C:\Foo> $Log = Get-WinEvent -ListLog *wmi*
PS C:\Foo> $Log

LogMode   MaximumSizeInBytes RecordCount LogName
-------   ------------------ ----------- -------
Circular            1052672         982 Microsoft-Windows-WMI-Activity/Operational
```

Figure 15.9: Viewing WMI-related event logs

In *step 9*, you get the events from the log to view the different log levels, with output like this:

```
PS C:\Foo> # 9. Looking at the event types in the WMI log
PS C:\Foo> $Events = Get-WinEvent -LogName $Log.LogName
PS C:\Foo> $Events | Group-Object -Property LevelDisplayName

Count Name                      Group
----- ----                      -----
   22 Error                     {System.Diagnostics.Eventing.Reader.EventLogRecord,System.Diagno…
  960 Information               {System.Diagnostics.Eventing.Reader.EventLogRecord,System.Diagno…
```

Figure 15.10: Discovering WMI event types

In *step 10*, you view the first five WMI event log entries on SRV1. The output looks like this:

```
PS C:\Foo> # 10. Examining WMI event log entries
PS C:\Foo> $Events |
             Select-Object -First 5 |
               Format-Table -Wrap

   ProviderName: Microsoft-Windows-WMI-Activity

TimeCreated           Id LevelDisplayName Message
-----------           -- ---------------- -------
19/04/2021 15:39:52 5857 Information      CIMWin32a provider started with result code 0x0. HostProcess =
                                          wmiprvse.exe; ProcessID = 6716; ProviderPath =
                                          %systemroot%\system32\wbem\wmipcima.dll
19/04/2021 15:39:52 5857 Information      CIMWin32 provider started with result code 0x0. HostProcess =
                                          wmiprvse.exe; ProcessID = 6716; ProviderPath =
                                          %systemroot%\system32\wbem\cimwin32.dll
19/04/2021 15:38:01 5857 Information      CIMWin32 provider started with result code 0x0. HostProcess =
                                          wmiprvse.exe; ProcessID = 4932; ProviderPath =
                                          %systemroot%\system32\wbem\cimwin32.dll
19/04/2021 15:30:16 5857 Information      deploymentprovider provider started with result code 0x0. HostProcess
                                          = wmiprvse.exe; ProcessID = 5764; ProviderPath =
                                          %systemroot%\system32\wbem\ServerManager.DeploymentProvider.dll
19/04/2021 15:30:14 5857 Information      nettcpip provider started with result code 0x0. HostProcess =
                                          wmiprvse.exe; ProcessID = 7152; ProviderPath =
                                          %systemroot%\system32\wbem\NetTCPIP.dll
```

Figure 15.11: Viewing WMI event log entries

In *step 11*, you view the executable programs in the WBEM folder, with output like this:

```
PS C:\Foo> # 11.Viewing executable programs in WBEM folder
PS C:\Foo> $Files = Get-ChildItem -Path $WBEMFOLDER\*.exe
PS C:\Foo> "{0,15}  {1,-40}" -f 'File Name','Description'
PS C:\Foo> Foreach ($File in $Files){
             $Name = $File.Name
             $Desc = ($File |
                      Select-Object -ExpandProperty VersionInfo).FileDescription
            "{0,15}  {1,-40}" -f $Name,$Desc
            }

       File Name  Description
      mofcomp.exe  The Managed Object Format (MOF) Compiler
      scrcons.exe  WMI Standard Event Consumer - scripting
    unsecapp.exe  Sink to receive asynchronous callbacks for WMI client application
    wbemtest.exe  WMI Test Tool
     WinMgmt.exe  WMI Service Control Utility
     WMIADAP.exe  WMI Reverse Performance Adapter Maintenance Utility
    WmiApSrv.exe  WMI Performance Reverse Adapter
        WMIC.exe  WMI Commandline Utility
    WmiPrvSE.exe  WMI Provider Host
```

Figure 15.12: Viewing the executable programs in the WBEM folder

With PowerShell 7 (and optionally with Windows PowerShell), you access WMI's functionality using the cmdlets in the `CimCmdlets` module. You installed this module as part of installing PowerShell 7. The Windows installation program installed a version of this module when you installed the host OS. In *step 12*, you examine the properties of this module, with output like this:

```
PS C:\Foo> # 12. Examining the CimCmdlets module
PS C:\Foo> Get-Module -Name CimCmdlets |
             Select-Object -ExcludeProperty Exported* |
               Format-List -Property *

LogPipelineExecutionDetails : False
Name                        : CimCmdlets
Path                        : C:\Program Files\PowerShell\7\Microsoft.Management.Infrastructure.CimCmdlets.dll
ImplementingAssembly        : Microsoft.Management.Infrastructure.CimCmdlets, Version=7.1.3.0, Culture=neutral,
                              PublicKeyToken=31bf3856ad364e35
Definition                  :
Description                 :
Guid                        : fb6cc51d-c096-4b38-b78d-0fed6277096a
HelpInfoUri                 : https://aka.ms/powershell71-help
ModuleBase                  : C:\program files\powershell\7\Modules\CimCmdlets
PrivateData                 :
ExperimentalFeatures        : {}
Tags                        : {}
ProjectUri                  :
IconUri                     :
LicenseUri                  :
ReleaseNotes                :
RepositorySourceLocation    :
Version                     : 7.0.0.0
ModuleType                  : Binary
Author                      : PowerShell
AccessMode                  : ReadWrite
ClrVersion                  :
CompanyName                 : Microsoft Corporation
Copyright                   : Copyright (c) Microsoft Corporation.
DotNetFrameworkVersion      :
Prefix                      :
FileList                    : {}
CompatiblePSEditions        : {Core}
ModuleList                  : {}
NestedModules               : {}
PowerShellHostName          :
PowerShellHostVersion       :
PowerShellVersion           : 3.0
ProcessorArchitecture       : None
Scripts                     : {}
RequiredAssemblies          : {Microsoft.Management.Infrastructure.CimCmdlets.dll,
                               Microsoft.Management.Infrastructure.Dll}
RequiredModules             : {}
RootModule                  : Microsoft.Management.Infrastructure.CimCmdlets
SessionState                : System.Management.Automation.SessionState
OnRemove                    :
```

Figure 15.13: Viewing the CimCmdlets module details

In *step 13*, you use `Get-Command` to discover the cmdlets within the `CimCmdlets` module, which looks like this:

```
PS C:\Foo> # 13. Finding cmdlets in the CimCmdlets module
PS C:\Foo> Get-Command -Module CimCmdlets

CommandType     Name                               Version    Source
-----------     ----                               -------    ------
Cmdlet          Get-CimAssociatedInstance          7.0.0.0    CimCmdlets
Cmdlet          Get-CimClass                       7.0.0.0    CimCmdlets
Cmdlet          Get-CimInstance                    7.0.0.0    CimCmdlets
Cmdlet          Get-CimSession                     7.0.0.0    CimCmdlets
Cmdlet          Invoke-CimMethod                   7.0.0.0    CimCmdlets
Cmdlet          New-CimInstance                    7.0.0.0    CimCmdlets
Cmdlet          New-CimSession                     7.0.0.0    CimCmdlets
Cmdlet          New-CimSessionOption               7.0.0.0    CimCmdlets
Cmdlet          Register-CimIndicationEvent        7.0.0.0    CimCmdlets
Cmdlet          Remove-CimInstance                 7.0.0.0    CimCmdlets
Cmdlet          Remove-CimSession                  7.0.0.0    CimCmdlets
Cmdlet          Set-CimInstance                    7.0.0.0    CimCmdlets
```

Figure 15.14: Viewing the cmdlets in the CimCmdlets module

In *step 14*, you examine the properties of an object returned from WMI after using the `Get-CimInstance` command. The output from this step looks like this:

```
PS C:\Foo> # 14. Examining the .NET type returned from Get-CimInstance
PS C:\Foo> Get-CimInstance -ClassName Win32_Share | Get-Member

   TypeName: Microsoft.Management.Infrastructure.CimInstance#root/cimv2/Win32_Share

Name                      MemberType  Definition
----                      ----------  ----------
Dispose                   Method      void Dispose(), void IDisposable.Dispose()
Equals                    Method      bool Equals(System.Object obj)
GetCimSessionComputerName Method      string GetCimSessionComputerName()
GetCimSessionInstanceId   Method      guid GetCimSessionInstanceId()
GetHashCode               Method      int GetHashCode()
GetType                   Method      type GetType()
ToString                  Method      string ToString()
AccessMask                Property    uint AccessMask {get;}
AllowMaximum              Property    bool AllowMaximum {get;}
Caption                   Property    string Caption {get;}
Description               Property    string Description {get;}
InstallDate               Property    CimInstance#DateTime InstallDate {get;}
MaximumAllowed            Property    uint MaximumAllowed {get;}
Name                      Property    string Name {get;}
Path                      Property    string Path {get;}
PSComputerName            Property    string PSComputerName {get;}
Status                    Property    string Status {get;}
Type                      Property    uint Type {get;}
PSStatus                  PropertySet PSStatus {Status, Type, Name}
```

Figure 15.15: Examining the output from Get-CimInstance

There's more...

In *step 1*, you view the first 20 files/folders in the WBEM folder. There are a lot more files than you see in the figure. These include the DLL files for the WMI providers available on your system.

In *step 7*, you view the `WmiPrvSE` process. This process hosts WMI providers. Depending on the actions WMI is currently doing, you may see zero, one, or more occurrences of this process on your hosts.

In *step 9* and *step 10,* you discover and examine the WMI system event log. On `SRV1`, you can see there are both `Error` and `Information` event log entries. As you can see in *step 10*, the `Information` entries are mostly indications that WMI has loaded and invoked a particular provider. In most cases, the `Error` event messages you see are transient or benign.

In *step 14*, you look at the data returned by `Get-CimInstance`. As you can see from the output, the cmdlet returns the data obtained from the WMI class. This data is wrapped in a .NET object and has a class of `Microsoft.Management.Infrastructure.CimInstance`, with a suffix indicating the path to the WMI class, in this case, the `Win32_Share` class in the `ROOT/CIMV2` namespace. The returned object and its contents differ from that returned from `Get-WMIObject`. This could be an issue if you are upgrading Windows PowerShell scripts that previously used the older WMI cmdlets.

Exploring WMI namespaces

The PowerShell CIM cmdlets enable you to retrieve, update, and remove information from the CIM database and subscribe to and handle WMI events. The CIM database organizes its information into sets of classes within a hierarchical set of namespaces. A **namespace** is, in effect, a container holding WMI classes and WMI namespaces.

The name of the root WMI namespace is `ROOT`, although WMI is not overly consistent with regard to capitalization, as you may notice. A namespace can contain classes as well as additional child namespaces. For example, the root namespace has a child namespace, `CIMV2`, which you refer to as `ROOT\CIMV2`. This namespace also has child namespaces.

Every namespace in the CIM DB, including `ROOT`, has a special system class called `__NAMESPACE`. This class contains the names of child namespaces within the current namespaces. Thus, in the namespace `ROOT`, the `__NAMESPACE` class contains an instance for the `CIMV2` child namespace. Since this class exists inside every namespace, it is straightforward to discover all the namespaces on your system.

There are many namespaces and classes within WMI on any given system. The specific namespaces and classes depend on what applications and Windows features you run on a host. Additionally, not all the namespaces or classes are useful. The `ROOT\CimV2` namespace, implemented by the Win32 WMI provider, contains WMI classes that are useful to IT professionals. Other classes or namespaces may also contain useful classes, but most of the rest are typically only useful for developers implementing WMI components or WMI providers.

Another less commonly used namespace is `ROOT\directory\LDAP`, which contains classes related to Active Directory. While you perform most of the AD management using the AD cmdlets, there are features of this namespace, specifically eventing, that are not available with the AD cmdlets and that you may find useful.

Getting ready

This recipe uses `SRV1`, a domain-joined host. You have installed PowerShell 7 and VS Code on this host.

How to do it...

1. Viewing WMI classes in the root namespace

```
Get-CimClass -Namespace 'ROOT' |
  Select-Object -First 10
```

2. Viewing the `__NAMESPACE` class in ROOT

```
Get-CimInstance -Namespace 'ROOT' -ClassName __NAMESPACE |
  Sort-Object -Property Name
```

3. Getting and counting classes in `ROOT\CIMV2`

```
$Classes = Get-CimClass -Namespace 'ROOT\CIMV2'
"There are $($Classes.Count) classes in ROOT\CIMV2"
```

4. Discovering all the namespaces on `SRV1`

```
Function Get-WMINamespaceEnum {
  [CmdletBinding()]
  Param($NS)
  Write-Output $NS
  Get-CimInstance "__Namespace" -Namespace $NS @EAHT |
  ForEach-Object { Get-WMINamespaceEnum "$ns\$($_.name)"   }
}  # End of function
$Namespaces = Get-WMINamespaceEnum 'ROOT' |
  Sort-Object
"There are $($Namespaces.Count) WMI namespaces on SRV1"
```

5. Viewing the first 25 namespaces on SRV1

```
$Namespaces |
  Select-Object -First 25
```

6. Creating a script block to count namespaces and classes

```
$SB = {
 Function Get-WMINamespaceEnum {
   [CmdletBinding()]
   Param(
     $NS
    )
   Write-Output $NS
   $EAHT = @{ErrorAction = 'SilentlyContinue'}
   Get-CimInstance "__Namespace" -Namespace $NS @EAHT |
     ForEach-Object { Get-WMINamespaceEnum "$NS\$($_.Name)"   }
   }  # End of function
   $Namespaces = Get-WMINamespaceEnum 'ROOT' | Sort-Object
   $WMIClasses = @()
   Foreach ($Namespace in $Namespaces) {
   $WMIClasses += Get-CimClass -Namespace $Namespace
  }
 "There are $($Namespaces.Count) WMI namespaces on $(hostname)"
 "There are $($WMIClasses.Count) classes on $(hostname)"
}
```

7. Running the script block locally on SRV1

```
Invoke-Command -ComputerName SRV1 -ScriptBlock $SB
```

8. Running the script block on SRV2

```
Invoke-Command -ComputerName SRV2 -ScriptBlock $SB
```

9. Running the script block on DC1

```
Invoke-Command -ComputerName DC1 -ScriptBlock $SB
```

How it works...

In *step 1*, you view the WMI classes in the WMI root namespace on SRV1, with output like this:

```
PS C:\Foo> # 1. Viewing WMI classes in the root namespace
PS C:\Foo> Get-CimClass -Namespace 'ROOT' |
             Select-Object -First 10

   NameSpace: ROOT

CimClassName                        CimClassMethods      CimClassProperties
------------                        ---------------      ------------------
__SystemClass                       {}                   {}
__thisNAMESPACE                     {}                   {SECURITY_DESCRIPTOR}
__CacheControl                      {}                   {}
__EventConsumerProviderCacheControl {}                   {ClearAfter}
__EventProviderCacheControl         {}                   {ClearAfter}
__EventSinkCacheControl             {}                   {ClearAfter}
__ObjectProviderCacheControl        {}                   {ClearAfter}
__PropertyProviderCacheControl      {}                   {ClearAfter}
__NAMESPACE                         {}                   {Name}
__ArbitratorConfiguration           {}                   {OutstandingTasksPerUser, OutstandingTasksTotal, Per…
```

Figure 15.16: Examining WMI classes in the root namespace

In *step 2*, you examine the instances of the __NAMESPACE class in the root WMI namespace. The output looks like this:

```
PS C:\Foo> # 2. Viewing the __NAMESPACE class in ROOT
PS C:\Foo> Get-CimInstance -Namespace 'ROOT' -ClassName __NAMESPACE |
             Sort-Object -Property Name

Name             PSComputerName
----             --------------
AccessLogging
Appv
CIMV2
Cli
DEFAULT
directory
Hardware
Interop
InventoryLogging
Microsoft
msdtc
MSPS
PEH
Policy
RSOP
SECURITY
ServiceModel
StandardCimv2
subscription
WMI
```

Figure 15.17: Examining the __NAMESPACE class in the root namespace

In *step 3*, you get and count the classes in the `ROOT\CIMV2` namespace, with output like this:

```
PS C:\Foo> # 3. Getting and counting classes in ROOT\CIMV2
PS C:\Foo> $Classes = Get-CimClass -Namespace 'ROOT\CIMV2'
PS C:\Foo> "There are $($Classes.Count) classes in ROOT\CIMV2"
There are 1202 classes in ROOT\CIMV2
```

Figure 15.18: Counting the classes in the ROOT\CIMV2 namespace

In *step 4*, you define and then use a function to discover all the namespaces in WMI on this host, sorted alphabetically. The output of this step looks like this:

```
PS C:\Foo> # 4. Discovering all the namespaces on SRV1
PS C:\Foo> Function Get-WMINamespaceEnum {
             [CmdletBinding()]
             Param($NS)
             Write-Output $NS
             Get-CimInstance "__Namespace" -Namespace $NS -ErrorAction SilentlyContinue |
             ForEach-Object { Get-WMINamespaceEnum "$ns\$($_.name)"   }
           }  # End of function
PS C:\Foo> $Namespaces = Get-WMINamespaceEnum 'ROOT' |
             Sort-Object
PS C:\Foo> "There are $($Namespaces.Count) WMI namespaces on SRV1"
There are 128 WMI namespaces on SRV1
```

Figure 15.19: Discovering all the WMI namespaces on SRV1

In *step 5*, you view the first 25 namespace names, with output like this:

```
PS C:\Foo> # 5. Viewing first 25 namespaces on SRV1
PS C:\Foo> $Namespaces |
             Select-Object -First 25
ROOT
ROOT\AccessLogging
ROOT\Appv
ROOT\CIMV2
ROOT\CIMV2\mdm
ROOT\CIMV2\mdm\dmmap
ROOT\CIMV2\mdm\MS_409
ROOT\CIMV2\ms_409
ROOT\CIMV2\power
ROOT\CIMV2\power\ms_409
ROOT\CIMV2\Security
ROOT\CIMV2\Security\MicrosoftTpm
ROOT\CIMV2\Security\MicrosoftVolumeEncryption
ROOT\CIMV2\TerminalServices
ROOT\CIMV2\TerminalServices\ms_409
ROOT\Cli
ROOT\Cli\MS_409
ROOT\DEFAULT
ROOT\DEFAULT\ms_409
ROOT\directory
ROOT\directory\LDAP
ROOT\directory\LDAP\ms_409
ROOT\Hardware
ROOT\Hardware\ms_409
ROOT\Interop
```

Figure 15.20: Listing the first 25 namespaces in WMI on SRV1

In *step 6*, you create a script block that counts WMI namespaces and classes. This step generates no console output. In *step 7*, you run this function against `SRV1` with output like this:

```
PS C:\Foo> # 7. Running the script block locally on SRV1
PS C:\Foo> Invoke-Command -ComputerName SRV1 -ScriptBlock $SB
There are 128 WMI namespaces on SRV1
There are 17384 classes on SRV1
```

Figure 15.21: Counting WMI namespaces and classes on SRV1

In *step 8*, you run the script block on `SRV2`, with output like this:

```
PS C:\Foo> # 8. Running the script block on SRV2
PS C:\Foo> Invoke-Command -ComputerName SRV2 -ScriptBlock $SB
There are 99 WMI namespaces on SRV2
There are 13679 classes on SRV2
```

Figure 15.22: Counting WMI namespaces and classes on SRV2

In the final step, *step 9*, you run the script block on a domain controller, `DC1`. The output of this step is as follows:

```
PS C:\Foo> # 9. Running the script block on DC1
PS C:\Foo> Invoke-Command -ComputerName DC1 -ScriptBlock $SB
There are 107 WMI namespaces on DC1
There are 15521 classes on DC1
```

Figure 15.23: Counting WMI namespaces and classes on DC1

There's more...

In *step 2*, you determine the child namespaces of the root namespaces. Each instance contains a string with the child's namespace name. The first entry is `AccessLogging`. Therefore, the namespace name of this child namespace is `ROOT\AccessLogging`.

In *step 3*, you count the classes in `ROOT\CIMV2`. As mentioned before, not all of these classes are useful to an IT pro, although many are. You can use your search engine to find classes that might be useful.

In *step 4*, you define a recursive function. When you call this function, specifying the `ROOT` namespace, the function retrieves the child namespace names from the `__NAMESPACE` class in the root namespace. Then, for each child's namespace of the root, the function calls itself with a child namespace name. Eventually, this function returns the names of every namespace in WMI. You then sort this alphabetically by namespace name. Note that you can sort the output of `GET-WMINamespaceEnum` without specifying a property – you are sorting on the contents of the strings returned from the function.

In *step 5*, you view some of the namespaces in WMI on `SRV1`. Two important namespaces are `ROOT\CIMV2` and `ROOT\directory\LDAP`. The former contains classes provided by the Win32 WMI provider, containing details about software and hardware on your system, including the BIOS, the OS, files, and a lot more.

In *step 6*, you define a recursive function that enumerates all the WMI namespaces. The function starts at the root WMI namespace (or any namespace you call the function with) and obtains all the child namespaces. For each of those, the function returns the child namespaces, and so on. For each namespace, the function counts the number of classes in each namespace.

Step 7, *step 8*, and *step 9* run the function (defined in *step 6*) remotely. These steps count and display a count of the namespaces and classes on all three systems. For this reason, you should expect the number of classes and namespaces to differ.

Exploring WMI classes

A WMI class defines a WMI-managed object. All WMI classes live within a namespace and contain members that include properties, methods, and events. An example class is `Win32_Share`, which you find in the `ROOT\CIMV2` namespace. This class defines an SMB share on a Windows host. Within WMI, the Win32 provider implements this class (along with multiple other OS and host-related classes).

As mentioned, you typically use the SMB cmdlets to manage SMB shares (as discussed in *Chapter 10*, *Managing Shared Data*, including the *Creating and securing SMB shares* recipe). Likewise, you carry out most AD management activities using AD cmdlets rather than accessing the information via WMI. Nevertheless, you can do things with WMI, such as event handling, that can be very useful to the IT professional.

A WMI class contains one or more properties that are attributes of an occurrence of a WMI class. Classes can also include methods that act on a WMI occurrence. For example, the `Win32_Share` class contains a `Name` property that holds the share name for that share. Each WMI property has a data type, such as integer or string. The `Win32_Share` class also has a `Create()` method to create a new SMB share and a `Delete()` method to remove a specific share. A WMI method can be **dynamic** (instance-based) or **static** (class-related). The `Win32_Share`'s `Delete()` method is a dynamic method you use to delete a particular SMB share. The `Create()` method is a static method that the class can perform to create a new SMB share.

In this recipe, you use the CIM cmdlets to discover information about classes, and what a class can contain. You first examine a class within the default `ROOT\CIMV2` namespace. You also examine a class in a non-default namespace and discover the objects contained in a class.

Getting ready

This recipe uses `SRV1`, a domain-joined host. You have installed PowerShell 7 and VS Code on this host.

How to do it...

1. Viewing the `Win32_Share` class

```
Get-CimClass -ClassName Win32_Share
```

2. Viewing `Win32_Share` class properties

```
Get-CimClass -ClassName Win32_Share |
  Select-Object -ExpandProperty CimClassProperties |
    Sort-Object -Property Name |
      Format-Table -Property Name, CimType
```

3. Getting methods of the `Win32_Share` class

```
Get-CimClass -ClassName Win32_Share |
  Select-Object -ExpandProperty CimClassMethods
```

4. Getting classes in a non-default namespace

```
Get-CimClass -Namespace root\directory\LDAP |
  Where-Object CimClassName -match '^ds_group'
```

5. Viewing the instances of the `ds_group` class

```
Get-CimInstance -Namespace root\directory\LDAP -Classname 'DS_Group' |
  Format-Table -Property DS_name, DS_Member
```

How it works...

In *step 1*, you view a specific class, the `Win32_Share` class, with output like this:

```
PS C:\Foo> # 1. Viewing Win32_Share class
PS C:\Foo> Get-CimClass -ClassName Win32_Share

   NameSpace: ROOT/cimv2

CimClassName  CimClassMethods      CimClassProperties
------------  ----------------     ------------------
Win32_Share   {Create, SetShareIn… {Caption, Description, InstallDate, Name,
                                    Status, AccessMask, AllowMaximum, MaximumAllowed,
                                    Path, Type}
```

Figure 15.24: Viewing the Win32_Share WMI class

In *step 2*, you view the properties of the `Win32_Share` class. The output of this step looks like this:

```
PS C:\Foo> # 2. Viewing Win32_Share class properties
PS C:\Foo> Get-CimClass -ClassName Win32_Share |
             Select-Object -ExpandProperty CimClassProperties |
               Sort-Object -Property Name |
                 Format-Table -Property Name, CimType

Name            CimType
----            -------
AccessMask       UInt32
AllowMaximum    Boolean
Caption          String
Description      String
InstallDate    DateTime
MaximumAllowed   UInt32
Name             String
Path             String
Status           String
Type             UInt32
```

Figure 15.25: Viewing the Win32_Share class properties

In *step 3*, you use the `Get-CimClass` cmdlet to view the methods available with the `Win32_Share` class, with output like this:

```
PS C:\Foo> # 3. Getting methods of Win32_Share class
PS C:\Foo> Get-CimClass -ClassName Win32_Share |
             Select-Object -ExpandProperty CimClassMethods

Name          ReturnType Parameters                                  Qualifiers
----          ---------- ----------                                  ----------
Create            UInt32 {Access, Description, MaximumAllowed,     {Constructor, Implemented, MappingStrings, Static}
                          Name, Password, Path, Type}
SetShareInfo      UInt32 {Access, Description, MaximumAllowed}     {Implemented, MappingStrings}
GetAccessMask     UInt32 {}                                        {Implemented, MappingStrings}
Delete            UInt32 {}                                        {Destructor, Implemented, MappingStrings}
```

Figure 15.26: Viewing methods in the Win32_Share class

In *step 4*, you get group-related classes in the `ROOT\directory\LDAP` namespace. The step returns just those classes that have the name `ds_group`. As you can see, this matches a few of the classes in this namespace, as follows:

```
PS C:\Foo> # 4. Getting classes in a non-default namespace
PS C:\Foo> Get-CimClass -Namespace root\directory\LDAP |
             Where-Object CimClassName -match '^ds_group'

   NameSpace: ROOT/directory/LDAP

CimClassName               CimClassMethods   CimClassProperties
------------               ---------------   ------------------
ds_grouppolicycontainer    {}                {ADSIPath, DS_adminDescription, DS_allowedChildClasses, DS_allowedCh…
ds_group                   {}                {ADSIPath, DS_adminDescription, DS_allowedChildClasses, DS_allowedCh…
ds_groupofnames            {}                {ADSIPath, DS_adminDescription, DS_allowedChildClasses, DS_allowedCh…
ds_groupofuniquenames      {}                {ADSIPath, DS_adminDescription, DS_allowedChildClasses, DS_allowedCh…
```

Figure 15.27: Finding AD group-related classes in the LDAP namespace

In *step 5*, you get the instances of the `ds_group` WMI class. The output, shown here, includes both the AD group's name and the current members of each AD group. The output of this step looks like this:

```
PS C:\Foo> # 5. Viewing the instances of the ds_group class
PS C:\Foo> Get-CimInstance -Namespace root\directory\LDAP -Classname 'ds_group' |
             Format-Table -Property DS_name, DS_Member

DS_name                                  DS_Member
-------                                  ---------
Administrators                           {CN=Domain Admins,CN=Users,DC=Reskit,DC=Org,
CN=Enterprise                             CN=Enterprise Admins,CN=Users,DC=Reskit,DC=Org,
                                          CN=Administrator,CN=Users,DC=Reskit,DC=Org}
Users                                    {CN=Domain Users,CN=Users,DC=Reskit,DC=Org,
                                          CN=S-1-5-11,CN=ForeignSecurityPrincipals,DC=Reskit,  DC=Org,
                                          CN=S-1-5-4,CN=ForeignSecurityPrincipals,DC=Reskit,DC=Org}
Guests                                   {CN=Domain Guests,CN=Users,DC=Reskit,DC=Org,
                                          CN=Guest,CN=Users,DC=Reskit,DC=Org}
Print Operators
Backup Operators
Replicator
Remote Desktop Users
Network Configuration Operators
Performance Monitor Users
Performance Log Users
```

Figure 15.28: Finding AD groups and members

There's more...

In *step 3*, you view the methods in the `Win32_Share` class using `Get-CimClass`. Since you did not specify a namespace, the WMI cmdlet assumes you are interested in the `ROOT\CIV2` namespace. Note that, in this step, the `Create()` method has two important qualifiers: constructor and status. The constructor qualifier tells you that you use the static `Create()` method to construct a new instance of this class (and a new SMB share). Likewise, you use the `Delete()` method to remove an instance of the class.

In *step 5*, you view the first instances in the `ds_group` class. This class contains an instance for every group in the Reskit.Org domain. It contains more information for each group returned by your use of the `Get-ADGroup` cmdlet.

Obtaining local and remote WMI objects

In the *Exploring WMI classes* recipe, you discovered that WMI provides a large number (over 100) of namespaces on each host along with thousands of WMI classes. You use the `Get-CimInstance` cmdlet to return the instances of a WMI class on either the local or a remote host, as you can see in the recipe. This cmdlet returns the WMI instances for a specified WMI class wrapped in a .NET object.

With WMI, you have three ways you can use `Get-CimInstance`:

- The first way is to use the cmdlet to return all class occurrences and return all class properties.
- The second way is to use the `-Filter` parameter to specify a WMI filter. The WMI filter instructs the `Get-CimInstance` command to return some, and not all, instances of the desired class.
- The third method uses a WMI query using the **WMI Query Language** (**WQL**). A WQL query is, in effect, a SQL statement that instructs WMI to return some (or all) properties of some (or all) occurrences of the specified WMI class.

When you use WMI across the network, specifying a filter or a full WMI query can reduce the amount of data transiting the wire and improve performance.

As in previous recipes, you use the `Get-CimInstance` cmdlet to retrieve WMI class instances using each of these three approaches.

Getting ready

This recipe uses `SRV1`, a domain-joined host. You have installed PowerShell 7 and VS Code on this host.

How to do it...

1. Using `Get-CimInstance` in the default namespace

```
Get-CimInstance -ClassName Win32_Share
```

2. Getting WMI objects from a non-default namespace

```
$GCIMHT1 = @{
    Namespace = 'ROOT\directory\LDAP'
    ClassName = 'ds_group'
}
Get-CimInstance @GCIMHT1 |
  Sort-Object -Property Name |
    Select-Object -First 10 |
      Format-Table -Property DS_name, DS_distinguishedName
```

3. Using a WMI filter

```
$Filter = "ds_Name LIKE '%operator%' "
Get-CimInstance @GCIMHT1  -Filter $Filter |
  Format-Table -Property DS_Name
```

4. Using a WMI query

```
$Q = @"
  SELECT * from ds_group
    WHERE ds_Name like '%operator%'
"@
Get-CimInstance -Query $q -Namespace 'root\directory\LDAP' |
  Format-Table DS_Name
```

5. Getting a WMI object from a remote system (`DC1`)

```
Get-CimInstance -CimSession DC1 -ClassName Win32_ComputerSystem |
  Format-Table -AutoSize
```

How it works...

In *step 1*, you use `Get-CimInstance` to return all the instances of the `Win32_Share` class on `SRV1`, with output like this:

```
PS C:\Foo> # 1. Using Get-CimInstance in the default namespace
PS C:\Foo> Get-CimInstance -ClassName Win32_Share

Name   Path          Description
----   ----          -----------
ADMIN$ C:\Windows    Remote Admin
C$     C:\           Default share
E$     E:\           Default share
H$     H:\           Default share
IPC$                 Remote IPC
RKRepo C:\RKRepo     Reskit Repository
S$     S:\           Default share
screen C:\FileScreen
T$     T:\           Default share
W$     W:\           Default share
X$     X:\           Default share
Y$     Y:\           Default share
Z$     Z:\           Default share
```

Figure 15.29: Retrieving Win32_Share class instances on SRV1

In *step 2*, you use `Get-CimInstance` to retrieve instances of a class in a non-default namespace that you name explicitly, with output like this:

```
PS C:\Foo> # 2. Getting WMI objects from a non-default namespace
PS C:\Foo> $GCIMHT1 = @{
             Namespace = 'root\directory\LDAP'
             ClassName = 'ds_group'
           }
PS C:\Foo> Get-CimInstance @GCIMHT1 |
             Sort-Object -Property Name |
               Select-Object -First 10 |
                 Format-Table -Property DS_name, DS_distinguishedName

DS_name                            DS_distinguishedName
-------                            --------------------
Administrators                     CN=Administrators,CN=Builtin,DC=Reskit,DC=Org
Domain Guests                      CN=Domain Guests,CN=Users,DC=Reskit,DC=Org
Group Policy Creator Owners        CN=Group Policy Creator Owners,CN=Users,DC=Reskit,DC=Org
RAS and IAS Servers                CN=RAS and IAS Servers,CN=Users,DC=Reskit,DC=Org
Server Operators                   CN=Server Operators,CN=Builtin,DC=Reskit,DC=Org
Account Operators                  CN=Account Operators,CN=Builtin,DC=Reskit,DC=Org
Pre-Windows 2000 Compatible Access CN=Pre-Windows 2000 Compatible Access,CN=Builtin,DC=Reskit,DC=Org
Incoming Forest Trust Builders     CN=Incoming Forest Trust Builders,CN=Builtin,DC=Reskit,DC=Org
Windows Authorization Access Group CN=Windows Authorization Access Group,CN=Builtin,DC=Reskit,DC=Org
Terminal Server License Servers    CN=Terminal Server License Servers,CN=Builtin,DC=Reskit,DC=Org
```

Figure 15.30: Retrieving WMI objects in a non-default namespace

In *step 3*, you use a WMI filter, specified with the `-Filter` parameter to the `Get-CimInstance` cmdlet. The output looks like this:

```
PS C:\Foo> # 3. Using a WMI filter
PS C:\Foo> $Filter = "ds_Name LIKE '%operator%' "
PS C:\Foo> Get-CimInstance @GCIMHT1  -Filter $Filter |
             Format-Table -Property DS_Name

DS_Name
-------
Print Operators
Backup Operators
Network Configuration Operators
Cryptographic Operators
Access Control Assistance Operators
Server Operators
Account Operators
```

Figure 15.31: Retrieving WMI objects using a WMI filter

In *step 4*, you use a full WMI query that contains the namespace/class you wish to retrieve and details of which properties and which instances WMI should return, like the following:

```
PS C:\Foo> # 4. Using a WMI query
PS C:\Foo> $Q = @"
             SELECT * from ds_group
               WHERE ds_Name like '%operator%'
           "@
PS C:\Foo> Get-CimInstance -Query $q -Namespace 'root\directory\LDAP' |
             Format-Table DS_Name

DS_Name
-------
Print Operators
Backup Operators
Network Configuration Operators
Cryptographic Operators
Access Control Assistance Operators
Server Operators
Account Operators
```

Figure 15.32: Retrieving WMI objects using a WMI filter

In *step 5*, you retrieve a WMI object from a remote host, `DC1`. The class retrieved by this step, `Win32_ComputerSystem`, holds details of the host, such as hostname, domain name, and total physical memory, as you can see in the following output:

```
PS C:\Foo> # 5. Getting a WMI object from a remote system (DC1)
PS C:\Foo> Get-CimInstance -CimSession DC1 -ClassName Win32_ComputerSystem |
             Format-Table -AutoSize

Name PrimaryOwnerName   Domain     TotalPhysicalMemory Model           Manufacturer          PSComputerName
---- ----------------   ------     ------------------- -----           ------------          --------------
DC1  Packt book readers Reskit.Org 4294496256          Virtual Machine Microsoft Corporation DC1
```

Figure 15.33: Retrieving WMI information from DC1

There's more...

In *step 4*, you create a WMI query. This query returns all properties on any instance of the class whose name contains the characters "operator", using WMI's wildcard syntax. This query returns all properties in the groups that include printer operators and server operators, as you can see in the output from this step.

In *step 5*, you return details of the `DC1` host. You used `Get-CimInstance` to return the single occurrence of the `Win32_ComputerSystem` class. The output shows that the `DC1` host has 4 GB of memory. If you are using virtualization to implement this host, you may see a different value depending on how you configured the VM.

Using WMI methods

In many object-oriented programming languages, a method is some action that an object can carry out. WMI also provides class methods. For example, the `Win32_Share` class has a `Delete()` method to delete a given SMB share. The class also has the `Create()` static method that creates a new SMB share.

In many cases, WMI methods duplicate what you can do with other PowerShell cmdlets. You could, for example, use the `New-SMBShare` cmdlet to create a new SMB share rather than using the `Create()` static method of the `Win32_Share` class.

As mentioned previously, WMI methods include instance methods and static methods. A dynamic or instance method operates on a specific instance – for example, deleting a specific SMB share. Classes also provide static methods, and these do not need a reference to any existing class instances. For example, you can use the `Create()` static method to create a new SMB share (and a new occurrence in the `Win32_Share` class).

Getting ready

This recipe uses `SRV1`, a domain-joined host. You have installed PowerShell 7 and VS Code on this host.

How to do it...

1. Reviewing methods of `Win32_Share` class on `SRV1`

```
Get-CimClass -ClassName Win32_Share |
  Select-Object -ExpandProperty CimClassMethods
```

2. Reviewing properties of `Win32_Share` class

```
Get-CimClass -ClassName Win32_Share |
```

```
  Select-Object -ExpandProperty CimClassProperties |
    Format-Table -Property Name, CimType
```

3. Creating a new SMB share using the `Create()` static method

```
$NSHT = @{
  Name        = 'TestShare1'
  Path        = 'C:\Foo'
  Description = 'Test Share'
  Type        = [uint32] 0 # disk
}
Invoke-CimMethod -ClassName Win32_
Share -MethodName Create -Arguments $NSHT
```

4. Viewing the new SMB share

```
Get-SMBShare -Name 'TestShare1'
```

5. Viewing the new SMB share using `Get-CimInstance`

```
Get-CimInstance -Class Win32_Share -Filter "Name = 'TestShare1'"
```

6. Removing the share

```
Get-CimInstance -Class Win32_Share -Filter "Name = 'TestShare1'" |
  Invoke-CimMethod -MethodName Delete
```

How it works...

In *step 1*, you use `Get-CimClass` to retrieve and display the methods provided by the `Win32_Share` WMI class. The output is as follows:

```
PS C:\Foo> # 1. Reviewing methods of Win32_Share class on SRV1
PS C:\Foo> Get-CimClass -ClassName Win32_Share |
             Select-Object -ExpandProperty CimClassMethods

Name          ReturnType Parameters                                  Qualifiers
----          ---------- ----------                                  ----------
Create            UInt32 {Access, Description, MaximumAllowed,     {Constructor, Implemented, MappingStrings, Static}
                          Name, Password, Path, Type}
SetShareInfo      UInt32 {Access, Description, MaximumAllowed}     {Implemented, MappingStrings}
GetAccessMask     UInt32 {}                                        {Implemented, MappingStrings}
Delete            UInt32 {}                                        {Destructor, Implemented, MappingStrings}
```

Figure 15.34: Reviewing the methods contained in the Win32_Share WMI class

In *step 2*, you use the `Get-CimClass` cmdlet to get the properties of each instance of the `Win32_Share` class, producing the following:

```
PS C:\Foo> # 2. Reviewing properties of Win32_Share class
PS C:\Foo> Get-CimClass -ClassName Win32_Share |
             Select-Object -ExpandProperty CimClassProperties |
               Format-Table -Property Name, CimType

Name            CimType
----            -------
Caption          String
Description      String
InstallDate    DateTime
Name             String
Status           String
AccessMask       UInt32
AllowMaximum    Boolean
MaximumAllowed   UInt32
Path             String
Type             UInt32
```

Figure 15.35: Reviewing the properties of an instance of the Win32_Share class

With *step 3*, you use the `Invoke-CimMethod` cmdlet to invoke the `Create()` method of the `Win32_Share` class and create a new SMB share on `SRV1`, with output like this:

```
PS C:\Foo> # 3. Creating a new SMB share using the Create() static method
PS C:\Foo> $NSHT = @{
             Name        = 'TestShare1'
             Path        = 'C:\Foo'
             Description = 'Test Share'
             Type        = [uint32] 0      # disk type share
           }
PS C:\Foo> Invoke-CimMethod -ClassName Win32_Share -MethodName Create -Arguments $NSHT

ReturnValue PSComputerName
----------- --------------
          0
```

Figure 15.36: Creating a new SMB share using WMI

In *step 4*, you use the `Get-SMBShare` cmdlet to get the SMB share information for the share you created in the previous step, producing output like this:

```
PS C:\Foo> # 4. Viewing the new SMB share
PS C:\Foo> Get-SMBShare -Name 'TestShare1'

Name       ScopeName Path   Description
----       --------- ----   -----------
TestShare1 *         C:\Foo Test Share
```

Figure 15.37: Viewing the newly created SMB share using Get-SMBShare

In *step 5*, you use `Get-CimInstance` to view the details of the share via WMI. This step produces the following output:

```
PS C:\Foo> # 5. Viewing the new SMB share using Get-CimInstance
PS C:\Foo> Get-CimInstance -Class Win32_Share -Filter "Name = 'TestShare1'"

Name       Path   Description
----       ----   -----------
TestShare1 C:\Foo Test Share
```

Figure 15.38: Viewing the newly created SMB share using Get-CimInstance

In the final step in this recipe, *step 6*, you use `Invoke-CimMethod` to delete a specific share (the one you created in *step 3*).

There's more...

In *step 3*, you create a new SMB share using the `Invoke-CimMethod` cmdlet. This cmdlet takes a hash table containing the properties and property values for the new SMB share. The cmdlet returns an object containing a `ReturnCode` property. A return code of 0 indicates success – in this case, WMI created the new share. For other values of the return code, you need to consult the documentation. For the `Win32_Share` class, you can find more online documentation at `https://docs.microsoft.com/en-us/windows/win32/cimwin32prov/create-method-in-class-win32-share`. This page shows the return codes that the `Create()` method generates and what those return codes indicate. For example, a return code of 8 would indicate that you attempted to create a share whose name already exists. If you plan to use WMI in production scripting, consider testing for non-zero return codes and handling common errors gracefully.

In *step 6*, you use a WMI method, `Delete()`, to delete the previously created SMB share. You delete this share by first using `Get-CimInstance` with a WMI filter to retrieve the share(s) to be deleted. You then pipe these share objects to the `Invoke-CimMethod` cmdlet and invoke the `Delete()` method on the instance passed in the pipeline. The approach taken in *step 6* is a common way to remove WMI class instances of this class and, for that matter, any WMI class.

Managing WMI events

A key feature of WMI is its event handling. There are thousands of events that can occur within a Windows system that might be of interest. For example, you might want to know if someone adds a new member to a high-privilege AD group such as Enterprise Admins. You can tell WMI to notify you when such an event occurs, and then take whatever action is appropriate. For example, you might just print out an updated list of group members when group membership changes occur. You could also check a list of users who should be members of the group and take some action if the user added is not authorized.

Events are handled both by WMI itself and by WMI providers. WMI itself can signal an event should a change be detected in a CIM class – that is, any new, updated, or deleted class instance. You can detect changes, too, to entire classes or namespaces. WMI calls these events **intrinsic** events. One common intrinsic event would occur when you (or Windows) start a new process and, by doing so, WMI adds a new instance to the `Win32_Process` class (contained in the `ROOT/CIMV2` namespace).

WMI providers can also implement events. These are known as **extrinsic** WMI events. The AD WMI provider, for example, implements an event that fires any time the membership of an AD group changes. The Windows Registry provider also provides an extrinsic event that detects changes to the registry, such as a new registry key or an updated registry value.

To make use of WMI event management, you first create an event subscription. The event subscription tells WMI which event you want it to track. Additionally, you can define an event handler that tells WMI what you want to do if the event occurs. For example, if a new process starts, you may wish to display the event's details. If an AD group membership changes, you might want to check to see if any group members are not authorized and report the fact or possibly even delete the invalid group member.

For more details on how you can receive WMI events, see `https://docs.microsoft.com//windows/win32/wmisdk/receiving-a-wmi-event`.

For information about the types of events to receive, see `https://docs.microsoft.com/windows/win32/wmisdk/determining-the-type-of-event-to-receive`.

There are two types of WMI eventing you can utilize. In this recipe, you create and handle temporary WMI events that work within a PowerShell session. If you close a session, WMI stops tracking all the events you registered for in that session. In the *Implementing permanent WMI eventing* recipe, you look at creating event subscriptions independent of the current PowerShell session.

When you register for a temporary event, you can provide WMI with a script block that you want WMI to execute when the event occurs. WMI executes this script block in the background, inside a PowerShell job.

When you register for a WMI event, PowerShell creates this job, in which it runs the action script. As with all PowerShell jobs, you use `Receive-Job` to view any output generated by the script. If your script block contains `Write-Host` statements, PowerShell sends any output directly to the console (and not the background job). You can also register for a WMI event without specifying an action block. In that case, WMI queues the events, and you can use `Get-WinEvent` to retrieve the event details.

When WMI detects an event, it generates an event record that contains the details of the event. These event records can be useful in helping you with more details about the event, but they are not a complete snapshot of the event. You can register for a WMI event should the membership of an AD group change and receive details such as the new member. However, the event record does not contain details of the user who modified the group's membership, or the IP address of the host used to effect the unauthorized change.

Getting ready

This recipe uses SRV1, a domain-joined host. You have installed PowerShell 7 and VS Code on this host.

How to do it...

1. Registering an intrinsic event

```
$Query1 = "SELECT * FROM __InstanceCreationEvent WITHIN 2
          WHERE TargetInstance ISA 'Win32_Process'"
$CEHT = @{
  Query             = $Query1
  SourceIdentifier = 'NewProcessEvent'
}
Register-CimIndicationEvent @CEHT
```

2. Running Notepad to trigger the event

```
notepad.exe
```

3. Getting the new process event

```
$Event = Get-Event -SourceIdentifier 'NewProcessEvent' |
           Select-Object -Last 1
```

4. Displaying event details

```
$Event.SourceEventArgs.NewEvent.TargetInstance
```

5. Unregistering the event

```
Unregister-Event -SourceIdentifier 'NewProcessEvent'
```

6. Registering an event query based on the registry provider

```
New-Item -Path 'HKLM:\SOFTWARE\Packt' | Out-Null
$Query2 = "SELECT * FROM RegistryValueChangeEvent
            WHERE Hive='HKEY_LOCAL_MACHINE'
              AND KeyPath='SOFTWARE\\Packt' AND ValueName='MOLTUAE'"
$Action2 = {
  Write-Host -Object "Registry Value Change Event Occurred"
```

```
  $Global:RegEvent = $Event
}
Register-
CimIndicationEvent -Query $Query2 -Action $Action2 -Source RegChange
```

7. Creating a new registry key and setting a value entry

```
$Q2HT = [ordered] @{
  Type  = 'DWord'
  Name  = 'MOLTUAE'
  Path  = 'HKLM:\Software\Packt'
  Value = 42
}
Set-ItemProperty @Q2HT
Get-ItemProperty -Path HKLM:\SOFTWARE\Packt
```

8. Unregistering the event

```
Unregister-Event -SourceIdentifier 'RegChange'
```

9. Examining event details

```
$RegEvent.SourceEventArgs.NewEvent
```

10. Creating a WQL event query

```
$Group = 'Enterprise Admins'
$Query1 = @"
  Select * From __InstanceModificationEvent Within 5
   Where TargetInstance ISA 'ds_group' AND
         TargetInstance.ds_name = '$Group'
"@
```

11. Creating a temporary WMI event registration

```
$Event = @{
  Namespace =  'ROOT\directory\LDAP'
  SourceID  = 'DSGroupChange'
  Query     = $Query1
  Action    = {
    $Global:ADEvent = $Event
    Write-Host 'We have a group change'
  }
}
Register-CimIndicationEvent @Event
```

12. Adding a user to the Enterprise Admins group

```
Add-ADGroupMember -Identity 'Enterprise Admins' -Members Malcolm
```

13. Viewing the newly added user

```
$ADEvent.SourceEventArgs.NewEvent.TargetInstance |
  Format-Table -Property DS_sAMAccountName,DS_Member
```

14. Unregistering the event

```
Unregister-Event -SourceIdentifier 'DSGroupChange'
```

How it works...

In *step 1*, you register for an intrinsic event that occurs whenever Windows starts a process. The registration does not include an action block. In *step 2*, you run `Notepad.exe` to trigger the event. In *step 3*, you use `Get-Event` to retrieve details of the event. These three steps produce no console output.

In *step 4*, you view details of the process start up event, with output like this:

```
PS C:\Foo> # 4. Displaying event details
PS C:\Foo> $Event.SourceEventArgs.NewEvent.TargetInstance

ProcessId Name        HandleCount WorkingSetSize VirtualSize
--------- ----        ----------- -------------- -----------
1344      notepad.exe 219         18137088       2203487641600
```

Figure 15.39: Displaying event details

In *step 5*, you remove the registration for the process start event. This step generates no output. In *step 6*, you register a new event subscription using a WMI query that targets the WMI provider, with output like this:

```
PS C:\Foo> # 6. Registering an event query based on the registry provider
PS C:\Foo> New-Item -Path 'HKLM:\SOFTWARE\Packt' | Out-Null
PS C:\Foo> $Query2 = "SELECT * FROM RegistryValueChangeEvent
                      WHERE Hive='HKEY_LOCAL_MACHINE'
                      AND KeyPath='SOFTWARE\\Packt' AND ValueName='MOLTUAE'"
PS C:\Foo> $Action2 = {
             Write-Host -Object "Registry Value Change Event Occurred"
             $Global:RegEvent = $Event
           }
PS C:\Foo> Register-CimIndicationEvent -Query $Query2 -Action $Action2 -Source RegChange

Id     Name            PSJobTypeName   State         HasMoreData     Location             Command
--     ----            -------------   -----         -----------     --------             -------
11     RegChange                       NotStarted    False
```

Figure 15.40: Registering for a registry provider-based event

With the event registration complete, in *step 7*, you create a new registry key and set a registry key value to test the event subscription. The output of this step looks like this:

```
PS C:\Foo> # 7. Creating a new registry key and setting a value entry
PS C:\Foo> $Q2HT = [ordered] @{
             Type  = 'DWord'
             Name  = 'MOLTUAE'
             Path  = 'HKLM:\Software\Packt'
             Value = 42
           }
PS C:\Foo> Set-ItemProperty @Q2HT
Registry Value Change Event Occurred

PS C:\Foo> Get-ItemProperty -Path HKLM:\SOFTWARE\Packt
MOLTUAE      : 42
PSPath       : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Packt
PSParentPath : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE
PSChildName  : Packt
PSDrive      : HKLM
PSProvider   : Microsoft.PowerShell.Core\Registry
```

Figure 15.41: Invoking the WMI registry event

In *step 8*, you un-register the registry event to avoid more event handling and event output. In *step 9*, you examine the output WMI generated based on the registry changes you made in *step 7*. The event details look like this:

```
PS C:\Foo> # 9. Examining event details
PS C:\Foo> $RegEvent.SourceEventArgs.NewEvent

SECURITY_DESCRIPTOR :
TIME_CREATED        : 132634791372033548
Hive                : HKEY_LOCAL_MACHINE
KeyPath             : SOFTWARE\Packt
ValueName           : MOLTUAE
PSComputerName      :
```

Figure 15.42: Examining a registry change WMI event

In *step 10*, you create a WQL query that captures changes to the Enterprise Admins AD group, generating no output. In *step 11*, you use the query to create a temporary WMI event that fires when the group membership changes. The output looks like this:

```
PS C:\Foo> # 11. Creating a temporary WMI event registration
PS C:\Foo> $Event = @{
             Namespace =  'root\directory\LDAP'
             SourceID  = 'DSGroupChange'
             Query     = $Query1
             Action    = {
               $Global:ADEvent = $Event
               Write-Host 'We have a group change'
             }
           }
PS C:\Foo> Register-CimIndicationEvent @Event

Id  Name           PSJobTypeName  State      HasMoreData  Location  Command
--  ----           -------------  -----      -----------  --------  -------
12  DSGroupChange                 NotStarted  False
```

Figure 15.43: Creating a temporary WMI event registration

To test this new directory change event, in *step 12*, you add a user to the Enterprise Admins AD group, generating output like this:

```
PS C:\Foo> # 12. Adding a user to the Enterprise Admins group
PS C:\Foo> Add-ADGroupMember -Identity 'Enterprise Admins' -Members Malcolm
PS C:\Foo> We have a group change
```

Figure 15.44: Triggering an AD group membership change event

In *step 13*, you examine the details of the event you generated in the previous step, with output like this:

```
PS C:\Foo> # 13. Viewing the newly added user
PS C:\Foo> $ADEvent.SourceEventArgs.NewEvent.TargetInstance |
             Format-Table -Property DS_sAMAccountName,DS_Member

DS_sAMAccountName DS_Member
----------------- ---------
Enterprise Admins {CN=Malcolm,OU=IT,DC=Reskit,DC=Org,
                   CN=Jerry Garcia,OU=IT,DC=Reskit,DC=Org,
                   CN=Administrator,CN=Users,DC=Reskit,DC=Org}
```

Figure 15.45: Examining AD group membership change event details

In the final step in this recipe, *step 14*, you unregister for the AD group membership change. This step generates no output.

There's more...

In *step 7*, you test the registry event handling, which includes an event action script block that you want PowerShell to execute when the event occurs. Since the script block you specified in *step 6* contains a `Write-Host` statement, you see the output on the PowerShell console.

In *step 9*, you examine the details WMI generated when the WMI registry change event occurred. As with other events, the event details omit potentially critical information. For example, the event does not tell you which user made this change or provide the new value of the registry value. You can examine the Windows Security event log to discover the user logged on that system (and therefore the user who made the change). And you can use `Get-ItemProperty` to determine the new value for this registry key property.

In *step 14*, you explicitly remove the registration for the AD change event. As an alternative, you could have closed the current PowerShell session, removing the event subscriptions.

Implementing permanent WMI eventing

In the *Managing WMI events* recipe, you used PowerShell's WMI event handling capability and used temporary event handling. The event handlers were active only as long as the PowerShell session was active and a user was logged in to the host. In that recipe, you created an event subscription and handled the events as your system generated them. This temporary event handling is a great troubleshooting tool that works well as long as you are logged in and are running PowerShell.

WMI also provides permanent event handling. You configure WMI to subscribe and handle events as they occur without using an active and open session. With permanent event handling, you configure WMI to subscribe to a specific event, for instance, adding a new member to a high-privilege AD group such as Enterprise Admins. You can also configure WMI to perform a predefined action when that event occurs, such as creating a report or sending an email message to report on the event if/when it occurs.

WMI in Windows defines several different types of permanent event consumers you can use to set a permanent event:

- **Active Script Consumer**: You use this to run a specific VBS script.
- **Log File Consumer**: This handler writes details of events to event log files.
- **NT Event Log Consumer**: This consumer writes event details into the Windows event log.
- **SMTP Event Consumer**: You can use this consumer to send an SMTP email message when an event occurs.

- **Command Line Consumer**: With this consumer, you can run a program, such as PowerShell 7, and pass a script filename. When the event occurs, the script has access to the event details and can do pretty much anything you can do in PowerShell.

Microsoft developed the Active Script consumer in the days of Visual Basic and VBS scripts. Unfortunately, this consumer does not support PowerShell scripts. The command-line WMI permanent event handler, on the other hand, enables you to run any programs you wish when the event handler detects an event occurrence. In this recipe, you ask WMI to run `pwsh.exe` and to run a specific script file when the event fires.

Managing permanent event handling is similar to the temporary WMI events you explored in the *Managing WMI events* recipe. You tell WMI which event to trap and what to do when that event occurs. You add a WMI class occurrence to three WMI classes, as you see in the recipe, to implement a permanent event handler as follows:

- Define an event filter. The event filter specifies the specific event that WMI should handle. You do this by adding a new instance to the specific event class you want WMI to detect. This event filter is essentially the same as in the previous recipe, *Managing WMI events*.
- Define an event consumer. In this step, you define the action you want WMI to take when the event occurs.
- Bind the event filter and event consumer. With this step, you add a new occurrence to an event binding class. This occurrence directs WMI to take some action (invoke the event consumer) whenever WMI detects that a specific WMI event (specified in the event filter) has occurred.

The AD WMI provider implements a wide range of AD-related events to which you can subscribe. WMI namespaces typically contain specific event classes that can detect when anything changes within the namespace. The namespace `ROOT/Directory/LDAP` has a system class named `__InstanceModificationEvent`. For a permanent event handler, you add an occurrence to this class.

A small word of caution is appropriate. You need to be very careful when working with WMI permanent event handling. A best practice is to understand how you remove the objects related to the permanent event handler. Note that unless you remove these records explicitly, WMI continues to monitor your host for the events, which can unnecessarily consume host resources.

This recipe also demonstrates a useful approach: creating PowerShell functions to display the event subscription and then to remove the subscription fully. Finally, be careful when changing an event filter's refresh time (specified in the WMI event filter). Decreasing the event refresh time can consume additional CPU and memory. For the most part, a refresh rate of once per second, or even every 5 seconds, is possibly overly excessive. Checking every 10 seconds is more than adequate for most WMI events.

Getting ready

This recipe uses SRV1, a domain-joined host. You have installed PowerShell 7 and VS Code on this host.

Also, ensure that the user Malcolm is not a member of the Enterprise Admins AD group.

How to do it...

1. Creating a list of valid users for the Enterprise Admins AD group

```
$OKUsersFile = 'C:\Foo\OKUsers.Txt'
$OKUsers  =  @'
Administrator
JerryG
'@
$OKUsers |
  Out-File -FilePath $OKUsersFile
```

2. Defining helper functions to get/remove permanent events

```
Function Get-WMIPE {
  '*** Event Filters Defined ***'
  Get-CimInstance -Namespace root\subscription
-ClassName __EventFilter  |
    Where-Object Name -eq "EventFilter1" |
     Format-Table Name, Query
  '***Consumer Defined ***'
  $NS = 'ROOT\subscription'
  $CN = 'CommandLineEventConsumer'
  Get-CimInstance -Namespace $ns -Classname  $CN |
    Where-Object {$_.name -eq "EventConsumer1"}  |
     Format-Table Name, Commandlinetemplate
  '***Bindings Defined ***'
  Get-CimInstance -Namespace root\subscription
-ClassName __FilterToConsumerBinding |
    Where-Object -FilterScript {$_.Filter.Name -eq "EventFilter1"} |
```

```
      Format-Table Filter, Consumer
}
Function Remove-WMIPE {
  Get-CimInstance -Namespace root\subscription __EventFilter |
    Where-Object Name -eq "EventFilter1" |
      Remove-CimInstance
  Get-CimInstance -Namespace root\subscription CommandLineEventConsumer |
    Where-Object Name -eq 'EventConsumer1' |
      Remove-CimInstance
  Get-CimInstance
-Namespace root\subscription __FilterToConsumerBinding  |
    Where-Object -FilterScript {$_.Filter.Name -eq 'EventFilter1'}   |
      Remove-CimInstance
}
```

3. Creating an event filter query

```
$Group = 'Enterprise Admins'
$Query = @"
  SELECT * From __InstanceModificationEvent Within 10
   WHERE TargetInstance ISA 'ds_group' AND
         TargetInstance.ds_name = '$Group'
"@
```

4. Creating the event filter

```
$Param = @{
  QueryLanguage =  'WQL'
  Query          =  $Query
  Name           =  "EventFilter1"
  EventNameSpace =  "root/directory/LDAP"
}
$IHT = @{
  ClassName = '__EventFilter'
  Namespace = 'root/subscription'
  Property  = $Param
}
$InstanceFilter = New-CimInstance @IHT
```

5. Creating the `Monitor.ps1` script run when the WMI event occurs

```
$MONITOR = @'
$LogFile   = 'C:\Foo\Grouplog.Txt'
$Group     = 'Enterprise Admins'
"On:  [$(Get-Date)]  Group [$Group] was changed" |
  Out-File -Force $LogFile -Append -Encoding Ascii
$ADGM = Get-ADGroupMember -Identity $Group
```

```
# Display who's in the group
"Group Membership"
$ADGM | Format-Table Name, DistinguishedName |
  Out-File -Force $LogFile -Append  -Encoding Ascii
$OKUsers = Get-Content -Path C:\Foo\OKUsers.txt
# Look at who is not authorized
foreach ($User in $ADGM) {
  if ($User.SamAccountName -notin $OKUsers) {
    "Unauthorized user [$($User.SamAccountName)] added to $Group"  |
      Out-File -Force $LogFile -Append  -Encoding Ascii
  }
}
"*********************************`n`n" |
Out-File -Force $LogFile -Append -Encoding Ascii
'@
$MONITOR | Out-File -Path C:\Foo\Monitor.ps1
```

6. Creating a WMI event consumer

```
# The consumer runs PowerShell 7 to execute C:\Foo\Monitor.ps1
$CLT = 'Pwsh.exe -File C:\Foo\Monitor.ps1'
$Param =[ordered] @{
  Name                = 'EventConsumer1'
  CommandLineTemplate = $CLT
}
$ECHT = @{
  Namespace = 'root/subscription'
  ClassName = "CommandLineEventConsumer"
  Property  = $param
}
$InstanceConsumer = New-CimInstance @ECHT
```

7. Binding the filter and consumer

```
$Param = @{
  Filter   = [ref]$InstanceFilter
  Consumer = [ref]$InstanceConsumer
}
$IBHT = @{
  Namespace = 'root/subscription'
  ClassName = '__FilterToConsumerBinding'
  Property  = $Param
}
$InstanceBinding = New-CimInstance   @IBHT
```

8. Viewing the event registration details

```
Get-WMIPE
```

9. Adding a user to the Enterprise Admins group

```
Add-ADGroupMember -Identity 'Enterprise admins' -Members Malcolm
```

10. Viewing the `Grouplog.txt` file

```
Get-Content -Path C:\Foo\Grouplog.txt
```

11. Tidying up

```
Remove-WMIPE   # invoke this function you defined above
$RGMHT = @{
 Identity = 'Enterprise admins'
 Member   = 'Malcolm'
 Confirm  = $false
}
Remove-ADGroupMember @RGMHT
Get-WMIPE      # ensure you have removed the event handling
```

How it works...

In *step 1*, you create a text file containing the `sAMAccountName` of users you have specified should be members of the Enterprise Admins group. In *step 2*, you create two helper functions, `Get-WMIPE` and `Remove-WMIPE`, to view and delete the WMI class instances that handle the event. In *step 3*, you create an event filter query which, in *step 4*, you add to WMI. These steps produce no output.

When the permanent WMI event occurs and the group membership changes, you want WMI to run a specific PowerShell script. In *step 5*, you create a file, `C:\Foo\Monitor.ps1`, producing no console output.

In *step 6*, you create a new event consumer, telling WMI to run the monitor script to detect the event. Then in *step 7,* you bind the event consumer and the event filter to complete setting up a permanent event handler. These two steps also produce no output.

In *step 8*, you use the `Get-WMIPE` function you defined in *step 2* to view the event filter details. The output of this step is as follows:

```
PS C:\Foo> # 8. Viewing the event registration details
PS C:\Foo> Get-WMIPE
*** Event Filters Defined ***

Name         Query
----         -----
EventFilter1   SELECT * From __InstanceModificationEvent Within 10
                WHERE TargetInstance ISA 'ds_group' AND
                      TargetInstance.ds_name = 'Enterprise Admins'

***Consumer Defined ***

Name           Commandlinetemplate
----           -------------------
EventConsumer1 Pwsh.exe -File C:\Foo\Monitor.ps1

***Bindings Defined ***

Filter                                Consumer
------                                --------
__EventFilter (Name = "EventFilter1") CommandLineEventConsumer (Name = "EventConsumer1")
```

Figure 15.46: Examining details of an AD group membership change event

In *step 9*, you test the permanent event handling by adding a new user (Malcolm) to the Enterprise Admins group. This step does not generate console output because you added no `Write-Host` statements to `Monitor.ps1`.

In *step 10*, you view the `Grouplog.txt` file with output like this:

```
PS C:\Foo> # 10. Viewing Grouplog.txt file
PS C:\Foo> Get-Content -Path C:\Foo\Grouplog.txt
On:  [04/20/2021 15:41:49]  Group [Enterprise Admins] was changed

Name          DistinguishedName
----          -----------------
Malcolm       CN=Malcolm,OU=IT,DC=Reskit,DC=Org
Jerry Garcia  CN=Jerry Garcia,OU=IT,DC=Reskit,DC=Org
Administrator CN=Administrator,CN=Users,DC=Reskit,DC=Org

Unauthorized user [Malcolm] added to Enterprise Admins
**********
```

Figure 15.47: Viewing Grouplog.txt (generated by Monitor.ps1)

In the final step in this recipe, *step 11*, you tidy up and call the `Remove-WMIPE` function you defined in *step 2* to remove the event details from WMI. At the end of this step, you run the `Get-WMIPE` function to ensure you have deleted all the event subscription class instances. The output of this step looks like this:

```
PS C:\Foo> # 11. Tidying up
PS C:\Foo> Remove-WMIPE    # invoke this function you defined above
PS C:\Foo> $RGMHT = @{
             Identity = 'Enterprise admins'
             Member   = 'Malcolm'
             Confirm  = $false
           }
PS C:\Foo> Remove-ADGroupMember @RGMHT
PS C:\Foo> Get-WMIPE       # ensure you have removed the event handling
*** Event Filters Defined ***
***Consumer Defined ***
***Bindings Defined **
```

Figure 15.48: Tidying up

There's more...

In *step 5*, you create a script that you want WMI to run any time the membership of the Enterprise Admins group changes. This script writes details to a text file (`Grouplog.txt`) containing the time the event occurred, the new membership, and whether this group now contains any unauthorized users. You could add to `Monitor.ps1` to send an email to an administrative mailbox or just remove the unauthorized user. You could also look in the Windows Security event log to find the most recent user to log on to the server.

In *step 10*, you view the output generated by the `Monitor.ps1` script. Note that it can take a few seconds for the permanent event handler to run the script to completion.

In this recipe, you create two helper functions, `Get-WMIPE` and `Remove-WMIPE`, to view and delete the permanent event handling details. You also call these functions at the end of the recipe to ensure you have not left the permanent event fully or partly configured.

packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

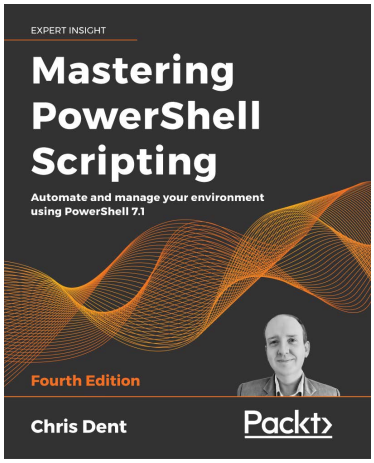
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Learn better with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.Packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.Packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

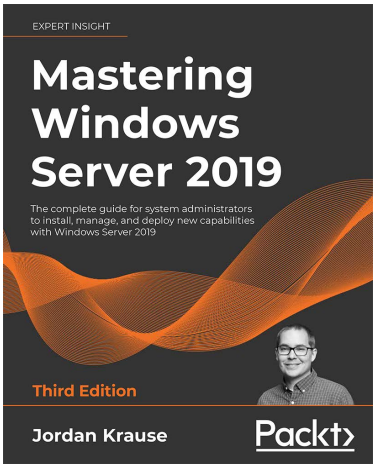


Mastering PowerShell Scripting – Fourth Edition

Chris Dent

ISBN: 978-1-80020-654-0

- Optimize code with functions, switches, and looping structures
- Work with objects and operators to test and manipulate data
- Parse and manipulate different data types
- Create scripts and functions using PowerShell
- Use jobs, runspaces, and runspace pools to run code asynchronously
- Write .NET classes with ease within PowerShell
- Create and implement regular expressions in PowerShell scripts
- Make use of advanced techniques to define and restrict the behavior of parameters



Mastering Windows Server 2019, Third Edition

Jordan Krause

ISBN: 978-1-80107-831-3

- Work with Server Core and Windows Admin Center
- Secure your network and data with modern technologies in Windows Server 2019
- Understand containers and understand when to use Nano Server
- Discover new ways to integrate your datacenter with Microsoft Azure
- Reinforce and secure your Windows Server
- Virtualize your datacenter with Hyper-V
- Explore Server Manager, PowerShell, and Windows Admin Center
- Centralize your information and services using Active Directory and Group Policy

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *Windows Server Automation with PowerShell Cookbook, Fourth Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, `please click here to go straight to the Amazon review page` for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

Symbols

A

B

C

D

E

F

G

H

P

Q

R

S

T

U

V

W