

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/en-gb/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
```

7. Displaying counts of features installed

8. Adding all RSAT tools to SRV1

9. Rebooting SRV1 (then logging on as the local administrator)

10. Getting details of RSAT tools now installed on SRV1

11. Displaying counts of commands after installing the RSAT tools

12. Displaying RSAT tools on SRV1

5

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-AD RMS	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
**\$PACKAGES = Find-Package
 "Discovered {0:N0} packages" -f \$PACKAGES.Count**
5. Showing the first five packages discovered
**\$PACKAGES |
 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.
NanoServerPackage	For more details and examples refer to our project site at https://github.com/PowerShell/ContainerProvider . A PackageManagement provider to Discover, Save and Install Nano Server Packages on-demand
Chocolatey	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. <ul style="list-style-type: none">- 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/en-us/powershell/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
# MIIFeQYJKoZIhvcNAQcCoIIFajCCBwYCAQEExCzAJBgUrDgMCGGUAMGkGCisGAQQB
# gjcCAQSGwzBZMDQGCisGAQQBgjccCAR4wJgIDAQAABBAfzDtgUUsITrck0sYpfvNR
# AgEAAgEAAgEAAgEAAgEAMCEwCQYFKw4DAhoFAAQUyPNDYU86IzmhHEACQWjqdKz7
# Z9KgggMQMIIDDDCCAfSgAwIBAgIQfGR5i0PPkY1B+bDt+q3dRzANBgkqhkiG9w0B
# AQsFADAeMRwwGgYDVQQDBNSZXNraXQgQ29kZSBTaWduaW5nMB4XDTEwMTAyODA5
# MjM0NDUyODUyODUyODUyODUyODUyODUyODUyODUyODUyODUyODUyODUyODUyODUy
# bmLuZzCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBALCcyu98FneWeHA
# f5eilGj9JolW3hGBZV/ftuY1hd9+WyHw4qQijfK4aUxvFbu8SrPMUUDp8xvFv0/I
# aqlsisJXw+TxwEpBTheiStgafDDx/nEyUi0PpRiHFNCq7zFITnrE/10NtLxSYFK
# cXLx3qjMstgoezxVqtvg8DhTmzPK3Gw5DurCoQ+I4NJknw5gf1wh/XyCptTJMSah
# +7Fcj4fRk/SCQKCFP10a7dtiXrI4v5VAEvg5faxNeLno2JELDoGjONgQsIQbBLH7
# WQjWjWtCtBXyXwYi2DZnhxfBmzCx0c/JC+oY6060r6Qpsed0zXVMrc58U6zrv1vU
# +OtWUY0CAwEAAaNGMEQwDgYDVR0PAQH/BAQDAgeAMBGA1UdJQQMMAoGCCSGAQUF
# BwMDMB0GA1UdDgQWBBSKINUNq+xNYwNo3U+8zc2ILQZGVTANBgkqhkiG9w0BAQsF
# AAOCAQEAHoe9W8Gq4dqHhS0jk+WvzdgrKy2D0rq10rVn1wuBmL5CcseNhTwdqDV
# xnsQn3TKRoKYc0Inpf4cVwuzuAoFN8l8g+TkYNdE+sz360mVYXFGWxj87sGPCADa
# qj6MtUadacG+Sa82GVD5uponQLQ6La2FplwzPqvaRKJFq7LhbLEJvotDB2Eiz9IX
# GVC8cUy8H/qM/bu2Xqwh40aGF15Bwbxeev2Ye1e8lrtMhAS/KBJnl9qi9Pg6tzc
# Uw+hI5uQ5LDuhbHxs3AXSZDoFBjKMAAa3YcxVpv0Dm3+85nX7fsjHcOnTxwYSI0W
# fJ51PjUgFaymmYdsUq+fTer+s9o3VTGCAdMwggHPAgEBMDIwHjEcMBoGA1UEAwwT
# UmVza2l0IENvZGUyODUyODUyODUyODUyODUyODUyODUyODUyODUyODUyODUyODUy
# oHgwGAYKKwYBBAGCNwIBDDEKMAigAoAaQAADAZBgkqhkiG9w0BQCQxMxDAYKKwYB
# BAGCNwIBBDAcBgorBgEEAYI3AgELMQ4wDAYKKwYBBAGCNwIBFTAjBgkqhkiG9w0B
# CQxMxMxMxMxMxMxMxMxMxMxMxMxMxMxMxMxMxMxMxMxMxMxMxMxMxMxMxMxMxMx
# 3qhygbk7bbDM97V9U6z03H8WxuxgPvpvahpzS8SZ0Ekt+Kc9LxYFcKsmKv6YT5Dh
# +bku3dYK0gctw39dBQb+ti0W18kTB7PLJKWaYd/b0/BtV5DbMTEKR3tBnm7eS00W
# aYDDUsQifx0nHlcpFY8aUjoJP4AcA456qnqe6dSu+SgCcNU3anAT60+Dv9viL7sU
# gl+Q9QIFLSg2t824Kh0eMjNEmJDitRY/ekCIk02JEE6D2S09wIyJhv0hEmHENpqx
# TFrqwBxwxkNzeaxEp3NrcAbKKuYGnzueogk4xqPnLkeEoPRl0z2c6LD03A38wJj
# 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]
                        7C64798B43CF918D41F9B0EDFAADD47

                        [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

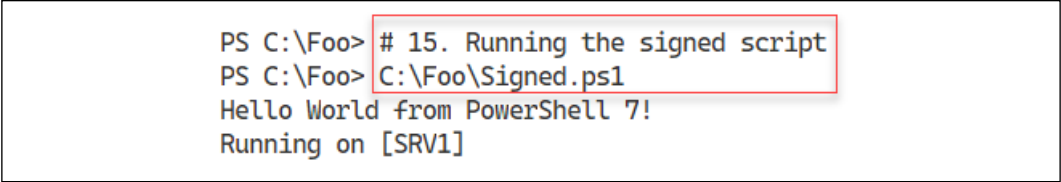
In step 13, you attempt to run the script again, but with a different result, like this:

```
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:

A screenshot of a PowerShell terminal window. The prompt is 'PS C:\Foo>'. The first command is '# 15. Running the signed script'. The second command is 'C:\Foo\Signed.ps1'. The output is 'Hello World from PowerShell 7!' followed by 'Running on [SRV1]'. A red rectangular box highlights the command 'C:\Foo\Signed.ps1' and its output 'Hello World from PowerShell 7!'.

```
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.co.uk/suppliers/digicert/products#code-signing-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/en-us/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:

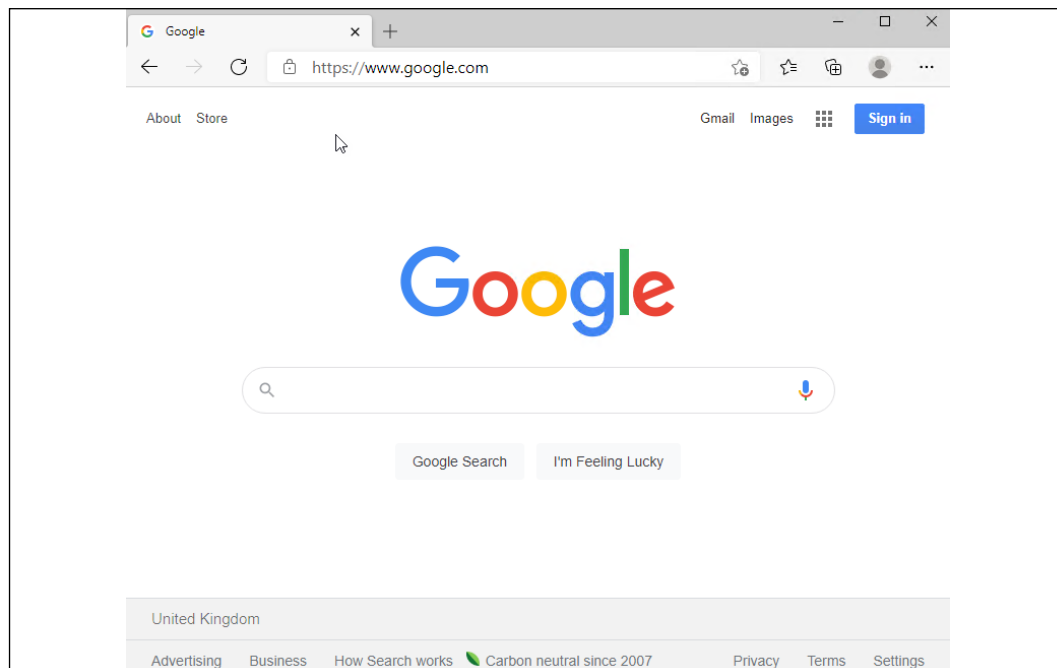


Figure 4.50: Executing the shortcut to Google.com

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:

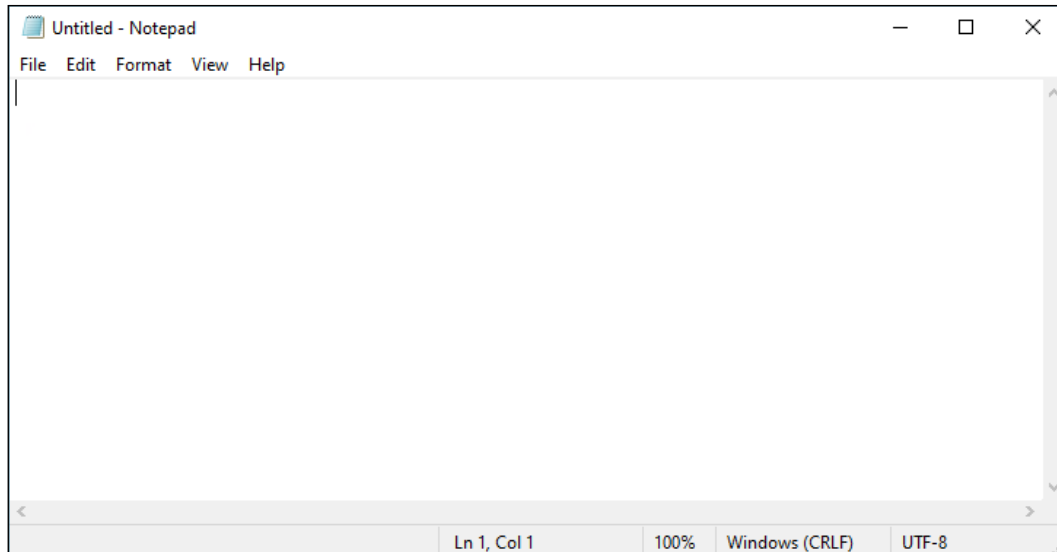


Figure 4.51: Executing the shortcut to Notepad

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 find more information about the ZIP file format at [https://en.wikipedia.org/wiki/Zip_\(file_format\)](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

```
$AFI1 = 'C:\Foo\Archive1.zip'
Compress-Archive -Path $Files -DestinationPath "$AFI1"
```

7. Compressing a folder containing files

```
$AFI2 = 'C:\Foo\Archive2.zip'
Compress-Archive -Path "C:\Foo\Archive" -DestinationPath $AFI2
```

8. Viewing the archive files

```
Get-ChildItem -Path $AFI1, $AFI2
```

9. Viewing archive content with Windows Explorer

```
explorer.exe $AFI1
```

10. Viewing the second archive with Windows Explorer

```
explorer.exe $AFI2
```

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 $FILE1 -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:

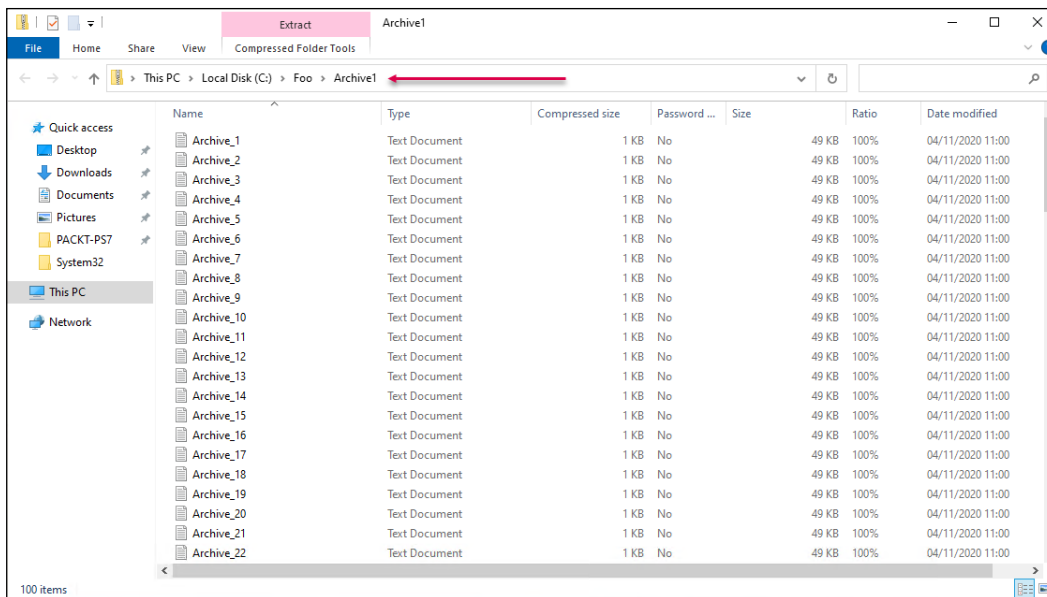


Figure 4.56: Using Windows Explorer to view the first archive

In step 10, you use Windows Explorer to view the second archive file, which looks like this:

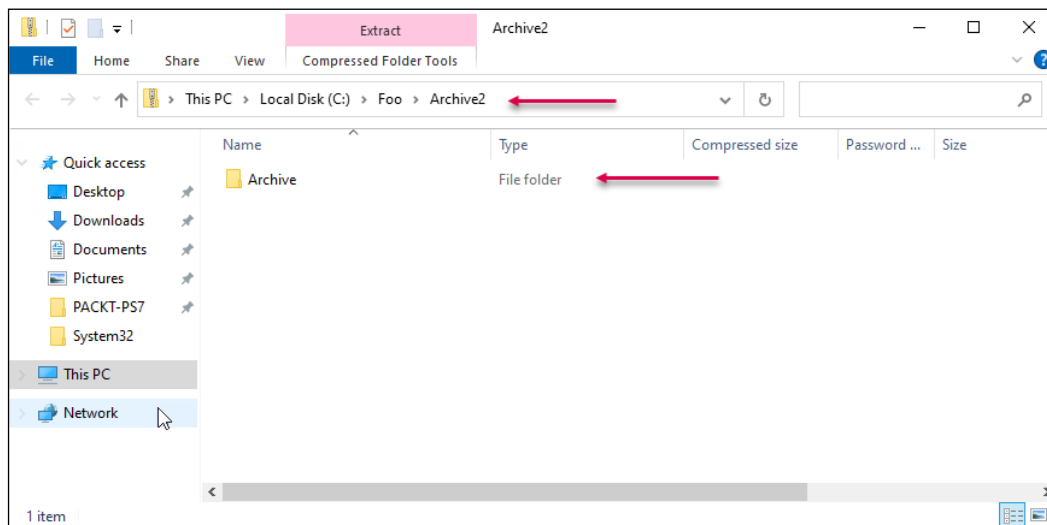


Figure 4.57: Using Windows Explorer to view the second archive

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.