4

Managing 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 PS Gallery
* Creating a local PowerShell repository
* Establishing a code signing environment
* Working with Shortcuts and PSShortcut module
* Working with Archive Files

# Introduction

Before you can begin to administer your Windows Server 2019 infrastructure, you need to create an environment in which you can use PowerShell to carry out the administration. That environment includes ensuring you have the tools you need close to hand, and ensuring the environment is as secure as possible.

To manage Windows roles and features as well as manage Windows itself with PowerShell, you need modules of PowerShell commands. You can manage most Windows features using PowerShell using the tools which 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 load the tools separately and manage features remotely. The Remote Server Administration Tools (RSAT) are modules that allow you to manage Windows roles and features. In “Installing RSAT Tools on Windows Server”, 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 modules/commands which you can use as an alternative to PowerShell commands. To manage packages, you need package management which you examine in “Exploring Package Management”. In “Exploring PowerShellGet and PS Gallery” you look at one source of modules and examine how to find and utilize modules contained in the PS Gallery.

With the OneGet module, PowerShell provides general package management within PowerShell. The module enables you to find and add packages to your computer. The PowerShellGet module, which sits on top and makes use of OneGet, enables you to find, download, and use PowerShell resource. You can use the PowerShell gallery or any other repository.

With Microsoft’s Authenticode technology, you can digitally sign an application or PowerShell script. The signature is a cryptographic hash of the executable or script that based on an X.509 code signing certificate. 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.

One issue with Kerberos authentication is known as the double-hop problem. TO understand more about the issue, you can read more at https://techcommunity.microsoft.com/t5/ask-the-directory-services-team/understanding-kerberos-double-hop/ba-p/395463. A solution to this issue is to use the Credential Security Support Provider (CredSSP). There are some risks with this technology, but properly managed CredSSP provides a solution to the double hop issue.

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 short cut to VS Code or PowerShell and place it on the desktop. The second type of short cut, a URL shortcut, is a file (with a .URL extension). So you might place a shortcut on a desktop which points to a specific web site. 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. A

# Installing RSAT Tools

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 so. These tools can include PowerShell cmdlets, functions, and aliases. 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 both Windows Server and the Windows Client (that is Windows 10). This recipe covers RSAT tool installation on Windows Server 2019.

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 Window 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 has fuller details of 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 2019 Data Centre Edition.

## How to do it...

1. Displaying counts of available PowerShell commands

$CommandsBeforeRSAT = Get-Command -Module \*

$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]"

1. Getting command types returned by Get-Command

$CommandsBeforeRSAT |

  Group-Object -Property CommandType

1. Checking the object type details

$CommandsBeforeRSAT |

  Get-Member |

    Select-Object -ExpandProperty TypeName -Unique

1. Getting the collection of PowerShell modules and a count of modules before adding  
   the RSAT tools

$ModulesBefore = Get-Module -ListAvailable

1. Displaying a count of modules available before adding the RSAT tools

$CountOfModulesBeforeRSAT = $ModulesBefore.Count

"$CountOfModulesBeforeRSAT modules available"

1. Getting a count of features available on SRV1

Import-Module -Name ServerManager -WarningAction SilentlyContinue

$Features  = Get-WindowsFeature

$FeaturesI = $Features | Where-Object Installed

| Where-object installed

$RsatF     = $Features |

               Where-Object Name -Match 'RSAT'

$RSATFI    = $RSATF |

              Where-Object Installed

1. 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

1. Adding ALL RSAT tools to SRV1

Get-WindowsFeature -Name \*RSAT\* |

  Install-WindowsFeature

1. Rebooting SRV1 then logging on as the local administrator

Restart-Computer -Force

1. 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'

1. Displaying after effects

"After Installation of RSAT tools on SRV1"

"$($IFSRV1A.count) features installed on SRV1"

"$($RSFSRV1A.count) RSAT features installed on SRV1"

1. Displaying RSAT tools on SRV1

$MODS = "$env:windir\system32\windowspowerShell\v1.0\modules"

$SMMOD = "$MODS\ServerManager"

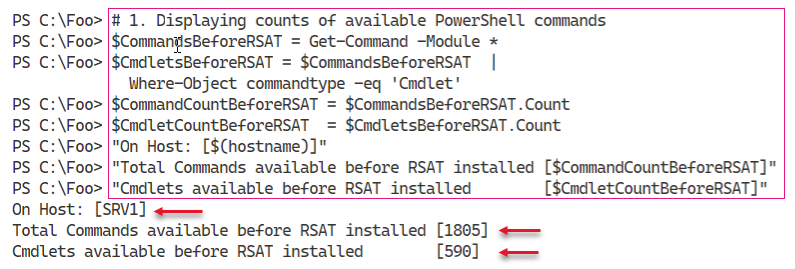
Update-FormatData -PrependPath "$SMMOD\\*.format.ps1xml"

Get-WindowsFeature |

  Where-Object Name -Match 'RSAT'

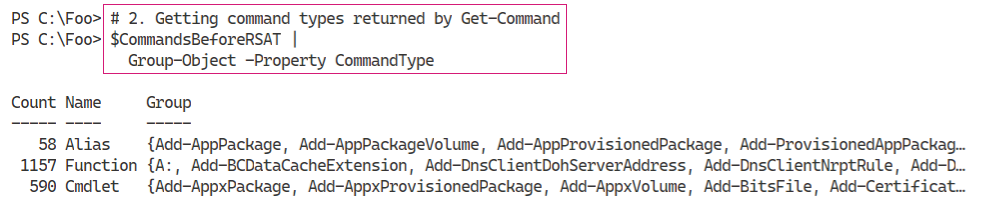
## How it works...

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:



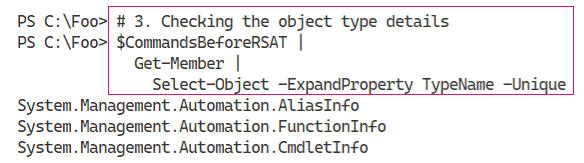
Insert image B42024\_04\_01.png

In step 2, you display a count of the types of commands available thus far on SRV1, which looks like this:



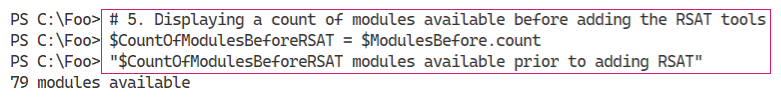
Insert image B42024\_04\_02.png

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:



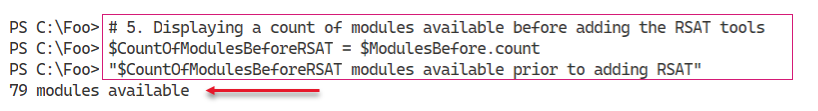
Insert image B42024\_04\_03.png

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, which looks like this:



Insert image B42024\_04\_04.png

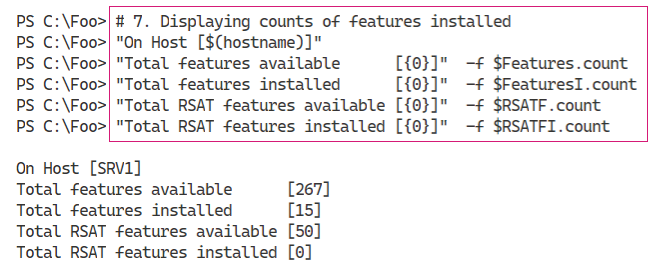
In step 5, you discover there are a total of 79 modules on SRV1 before you add the RSAT tools. The output of this step looks like this:



Insert image B42024\_04\_05.png

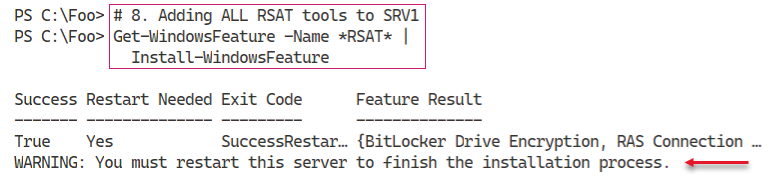
In step 6, you obtain counts of the features and features installed as well as the number of RSAT features available and installed. This step generates no output

In step 7, you display counts of the features available and installed, and the number of RSAT specific features available and installed, which looks like this:



Insert image B42024\_04\_06.png

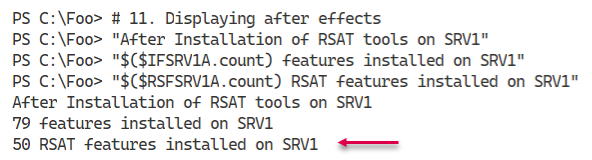
In step 8, you get and install all the RSAT features in Windows Server. This process does take a bit of time, and generates output like this:



Insert image B42024\_04\_07.png

The installation of these tools requires a restart, as you can see in the figure above. Thus, in step 9, you restart the system. After the restart, you log in to SRV1 as an administrator to continue.

Now that you have added all the RSAT-Related Windows Features, you can begin to 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:



Insert image B42024\_04\_08.png

In step 12, you display the RSAT features you installed in an earlier step. The output of this step looks like this



Insert image B42024\_04\_09.png

## There's more...

The output from step 1 shows there are 1805 total commands and 590 Module based cmdlets available on SRV1, before adding the RSAT tools. The actual number may vary, depending on what additional tools, feature, or applications you might have added to SRV1 or the Windows Server version itself.

In step 2 and step 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 PackageMangement PowerShell module provides tools that enable 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 PackageMangement 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 PowerShell Gallery” recipe. The primary function of the PackageMangement 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 (and potentially varying in quality).

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 Data Centre Edition.

## How to do it...

1. Reviewing the cmdlets in the PackageManagement module

Get-Command -Module PackageManagement

1. Reviewing installed providers with Get-PackageProvider

Get-PackageProvider |

  Format-Table -Property Name,

                         Version,

                         SupportedFileExtensions,

                         FromTrustedSource

1. Examining available Package Providers

$PROVIDERS = Find-PackageProvider

$PROVIDERS |

    Select-Object -Property Name,Summary |

      Format-Table -AutoSize -Wrap

1. Discovering and counting available packages

$PAGKAGES = Find-Package

"Discovered {0:N0} packages" -f $PAGKAGES.count

1. Showing the first 5 packages discovered

$PAGKAGES  |

    Select-Object -First 5 |

      Format-Table -AutoSize -Wrap

1. Installing Chocolatier provider

Install-PackageProvider -Name Chocolatier -Force

1. Verifying Chocolatier is in the list of installed providers

Get-PackageProvider |

  Select-Object -Property Name,Version

1. Discovering Packages from Chocolatier

$Start = Get-Date

$CPackages = Find-Package -ProviderName Chocolatier -Name \*

"$($CPackages.Count) packages available from Chocolatey"

$End = Get-Date

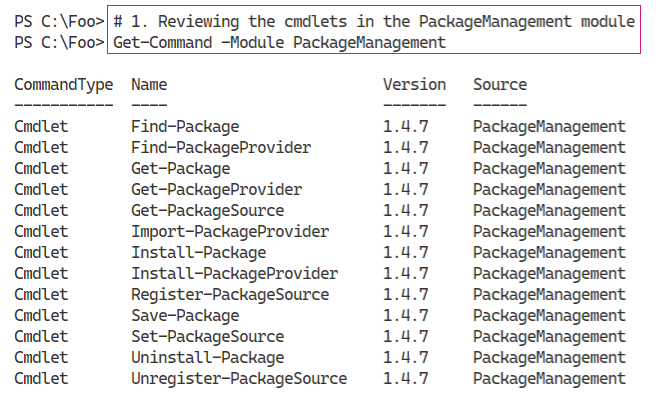
1. Displaying how long it took for Chocolatier

$Elapsed = $End - $Start

"Took {0:n3} seconds" -f $Elapsed.TotalSeconds

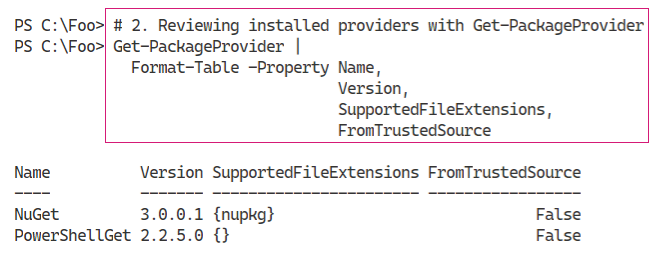
## How it works...

In step 1, you use Get-Command to view the commands provided by the PackageManagement module, which looks like this:



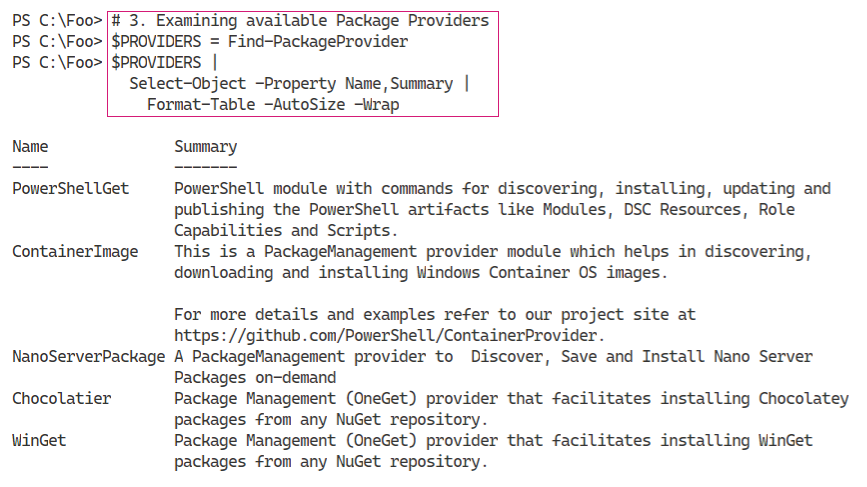
Insert image B42024\_04\_10.png

In step 2, you use the Get-PackageProvider cmdlet to discover the installed package providers, which looks like this



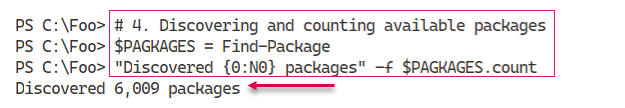
Insert image B42024\_04\_11.png

In step 3, you use the Find-PackageProvider to discover any other providers you can use. The output looks like this:



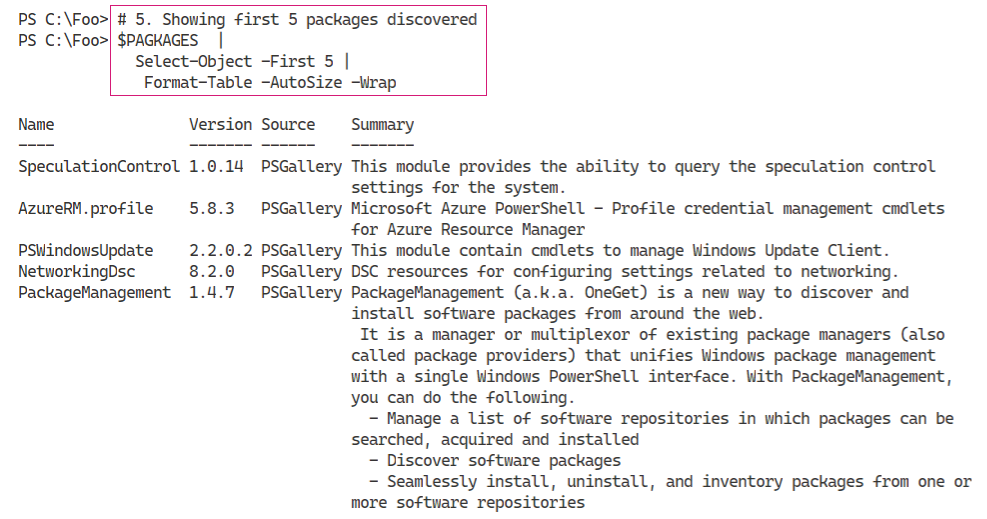
Insert image B42024\_04\_12.png

In step 4, you discover and count the packages you can find using Find-Packages, with output like this:



Insert image B42024\_04\_13.png

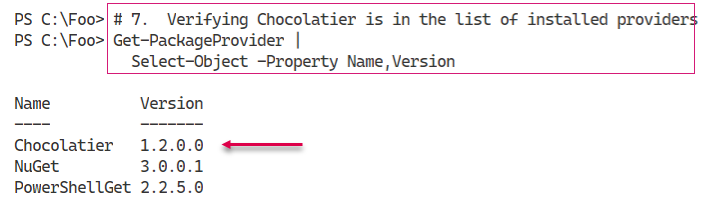
To illustrate some of the packages you just discovered, in step 5, you view the first 5 packages, which looks like this:



Insert image B42024\_04\_14.png

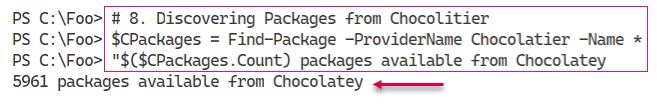
In step 6, you install the Chocolatier package provider, which gives you access via the package management to the Chocolately repository. Chocolately is a third-party application repository, although not directly supported by Microsoft. For more information about the Chocolately repository, see https://chocolatey.org/.

With step 7, you review the packaged providers now available on SRV1 to ensure Chocolatier is included, which looks like this:



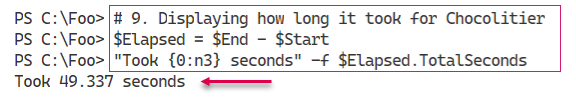
Insert image B42024\_04\_15.png

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 likes this:



Insert image B42024\_04\_16.png

In step 9, you display the time taken to find the packages on Chocolately, which looks like this:



Insert image B42024\_04\_17.png

## There's more...

In step 3, you obtain a list of packages and then, in step 4, display them. 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 retriever 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 Chocolately. 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 Chocolately repository. Chocolately provides you 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.

# Exploring PowerShellGet and 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 were 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 Data Centre Edition.

## How to do it...

1. Reviewing the commands available in the PowerShellGet module

Get-Command -Module PowerShellGet

1. Discovering Find-\* cmdlets in PowerShellGet module

Get-Command -Module PowerShellGet -Verb Find

1. Getting all commands, modules, DSC resources and scripts

$COM = Find-Command

$MOD = Find-Module

$DSC = Find-DscResource

$SCR = Find-Script

1. 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

1. Discovering NTFS Related modules

$MOD |

  Where-Object Name -match NTFS

1. Installing the NTFSSecurity module

Install-Module -Name NTFSSecurity -Force

1. Reviewing module commands

Get-Command -Module NTFSSecurity

1. Testing Get-NTFSAccess cmdlet

Get-NTFSAccess -Path C:\Foo

1. Creating a download folder

$DLFLDR = 'C:\Foo\DownloadedModules'

$NIHT = @{

  ItemType = 'Directory'

  Path     = $DLFLDR

  ErrorAction = 'SilentlyContinue'

}

New-Item @NIHT | Out-Null

1. Downloading the CountriesPS module

Save-Module -Name CountriesPS -Path $DLFLDR

1. Checking downloaded module

Get-ChildItem -Path $DLFLDR -Recurse |

  Format-Table -Property Fullname

1. Importing the CounriesPS module

$ModuleFolder = "$DLFLDR\CountriesPS"

Get-ChildItem -Path $ModuleFolder -Filter \*.psm1 -Recurse |

    Select-Object -ExpandProperty FullName -First 1 |

        Import-Module -Verbose

1. Checking commands in the module

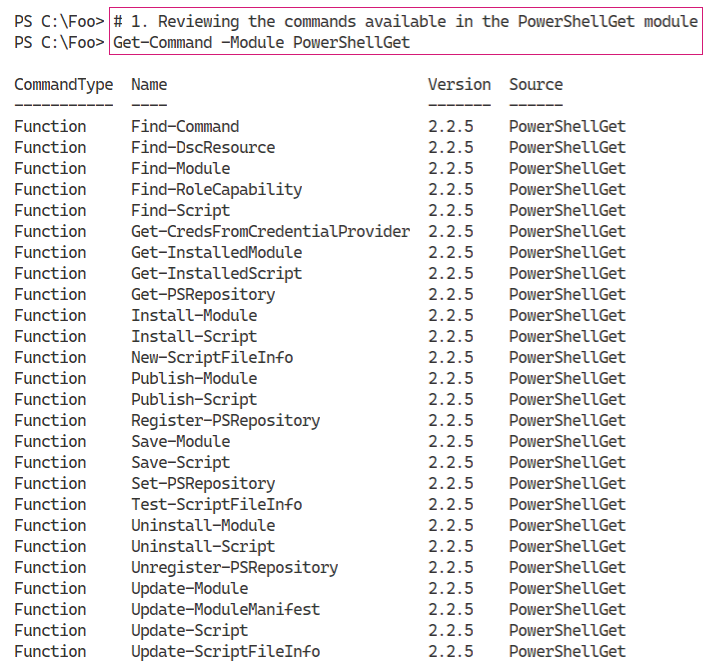
Get-Command -Module CountriesPSc

1. Using the Get-Country command

Get-Country -Name “United Kingdom”

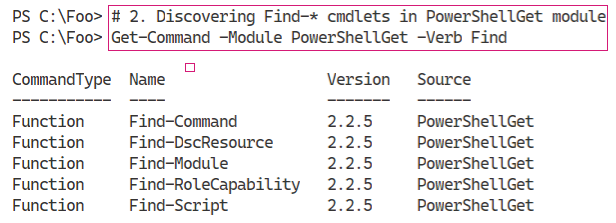
## How it works...

In step 1, you examine the commands within the PowerShellGet module, which looks like this:



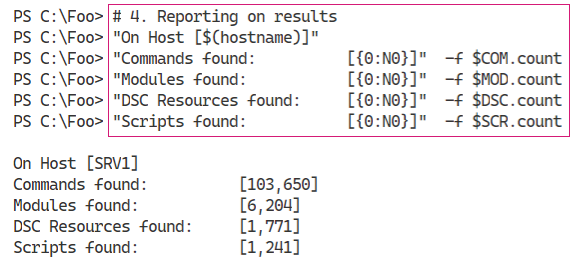
Insert image B42024\_04\_18.png

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:



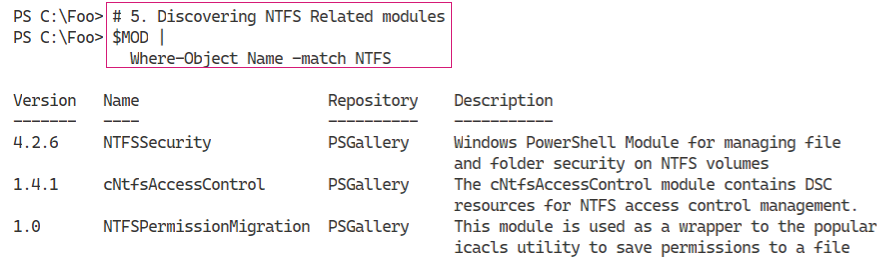
Insert image B42024\_04\_19.png

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:



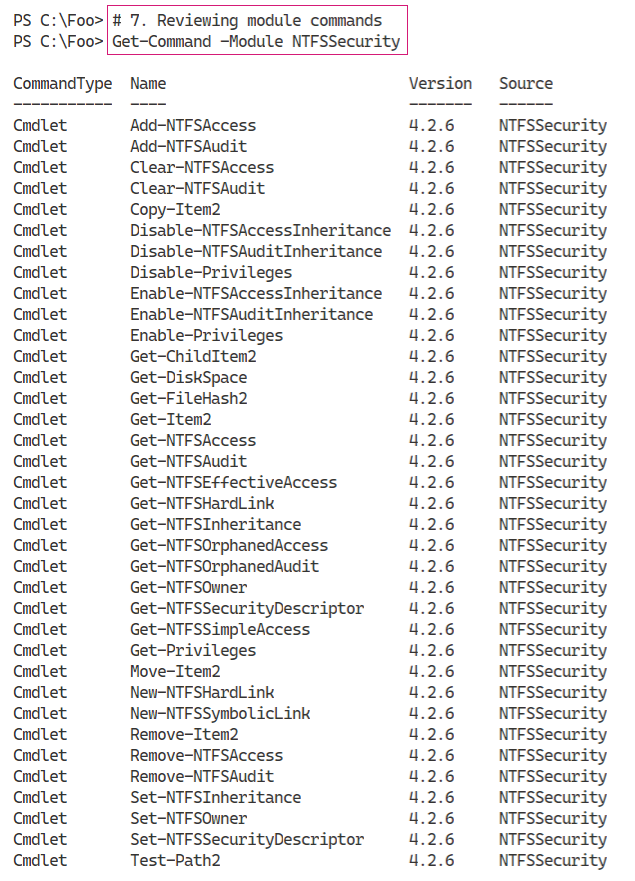
Insert image B42024\_04\_20.png

In step 5, you use the returned list of modules to discover any NTFS related modules, like this:



Insert image B42024\_04\_21.png

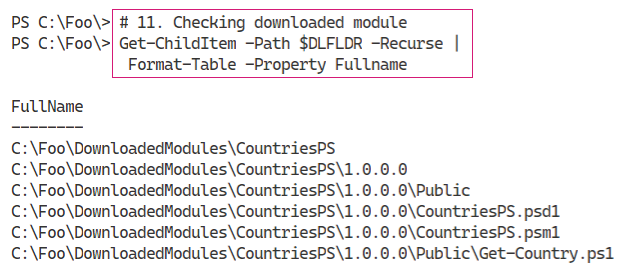
In step 6, you install the NTFSSecurity module, which produces no output. In step 7, you review the commands in the NTFSSecurity module, which produces output like this:



Insert image B42024\_04\_22.png

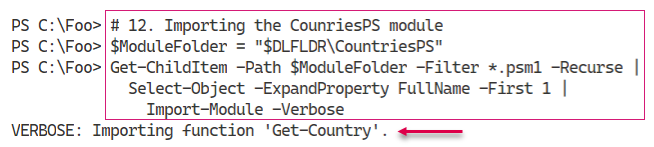
In preparation of downloading another module, in step 9, you create a new folder to hold the downloaded module. In step 10, you download the CountriesPS module. Neither of these steps generates output.

In step 11, you examine the files that make up the module, which looks like this:



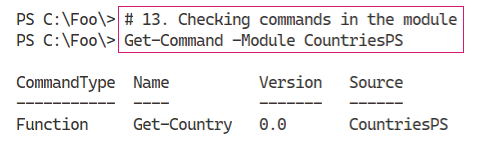
Insert image B42024\_04\_23.png

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:



Insert image B42024\_04\_24.png

In step 13, you use Get-Command to check the commands available in the CountriesPS module, which looks like this:



Insert image B42024\_04\_25.png

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:



Insert image B42024\_04\_26.png

## 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 5 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 might5 contain a command.
* Module - these are PowerShell modules where 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 aspect was created, but for the most part, is not used.

In step 3 and step 4, you discover the number of commands, modules, DSC Resources land 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 includes 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 step 6 through step 8, you make use of the NTFSSecurity module in the PowerShell gallery. This module, which you use in later chapters in this book, allows you to manage NTFS Access Control Lists. This module is an excellent example of a useful set of commands that the PowerShell development team could have, but did not, include inside Windows PowerShell or PowerShell 7. But with the PowerShellGet module, you can find, download, and leverage the modules in the PowerShell Gallery.

In step 9 through step 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 countries.eu web site. 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 re-invent the wheel. In some cases, you may develop a particularly useful module and then publish it to the PSGallery to share with others. See https://docs.microsoft.com/en-us/powershell/gallery/concepts/publishing-guidelines for guidelines regarding publishing to the PSGallery. 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 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.

As an alternative to using a public repository is to create you’re 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 you can use to set 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 setup 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 Datacentre Edition.

## How to do it...

1. Creating a repository folder

$LPATH = 'C:\RKRepo'

New-Item -Path $LPATH -ItemType Directory | Out-Null

1. Sharing the folder

$SMBHT = @{

  Name        = 'RKRepo'

  Path        = $LPATH

  Description = 'Reskit Repository.'

  FullAccess  = 'Everyone'

}

New-SmbShare @SMBHT

1. Registering the repository as trusted (on SRV1)

$Path = '\\SRV1\RKRepo'

$REPOHT = @{

  Name               = 'RKRepo'

  SourceLocation     = $Path

  PublishLocation    = $Path

  InstallationPolicy = 'Trusted'

}

Register-PSRepository @REPOHT

1. Viewing configured repositories

Get-PSRepository

1. Creating an HW module folder

$HWDIR = 'C:\HW'

New-Item -Path $HWDIR -ItemType Directory | Out-Null

1. Creating an elementary module

$HS = @"

Function Get-HelloWorld {'Hello World'}

Set-Alias GHW Get-HelloWorld

"@

$HS | Out-File $HWDIR\HW.psm1

1. Testing the module locally

Import-Module -Name $HWDIR\HW.PSM1 -Verbose

GHW

1. Creating a 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

1. Publishing the module

Publish-Module -Path $HWDIR -Repository RKRepo -Force

1. Viewing the results of publishing

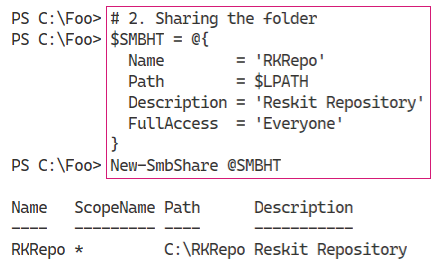
Find-Module -Repository RKRepo

1. Checking the repository's home folder

Get-ChildItem -Path $LPATH

## How it works...

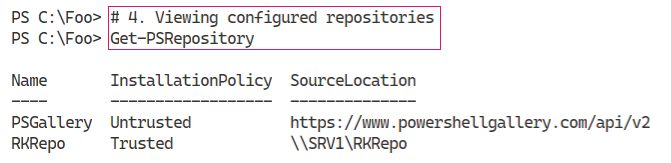
In step 1, you create a folder on SRV1 that you plan to use to hold the repository. There is no output from this step. In step 2, you create a new SMB share, RKREPO on SRV1, which looks like t his:



Insert image B42024\_04\_27.png

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.

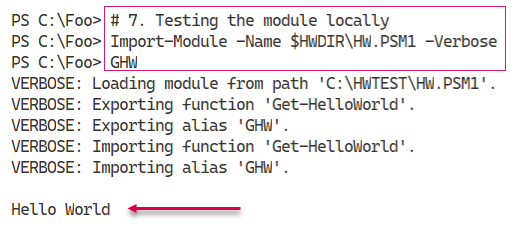
In step 4, you use the Get-PSRepository to view the repositories you have available, which looks like this:



Insert image B42024\_04\_28.png

To illustrate how you can utilize a provide 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 Import-Module takes upon importing your new module, you specify the -Verbose switch. The output of this step looks like this:



Insert image B42024\_04\_29.png

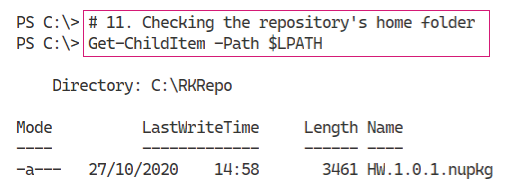
Before you can publish a module to your repository, you must create a module manifest to accompany the script module file. In step 8, you use the New-ModuleManifest to create a manifest for this module. With step 9, you publish your output. Both steps produce no output.

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:



Insert image B42024\_04\_30.png

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:



Insert image B42024\_04\_31.png

## 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 published 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 manifest, etc. 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 the Windows’ Authenticode Digital Signatures. 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 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 (you set the execution policy to AllSigned) or only for scripts you downloaded from a remote site (you set 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, 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 Certificate Authority 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 are 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. 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 Datacentre 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

1. Displaying the newly created certificate

$Cert = Get-ChildItem -Path Cert:\CurrentUser\my -CodeSigningCert

$Cert |

  Where-Object {$\_.Subjectname.Name -match $CHT.Subject}

1. 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

1. Signing your new script

$SHT = @{

  Certificate = $cert

  FilePath    = 'C:\Foo\Signed.ps1'

}

Set-AuthenticodeSignature @SHT

1. Checking script after signing

Get-ChildItem -Path C:\Foo\Signed.ps1

1. Viewing the signed script

Get-Content -Path C:\Foo\Signed.ps1.

1. Testing the signature

Get-AuthenticodeSignature -FilePath C:\Foo\Signed.ps1 |

  Format-List

1. Ensuring certificate is trusted

$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()

1. Checking the cert

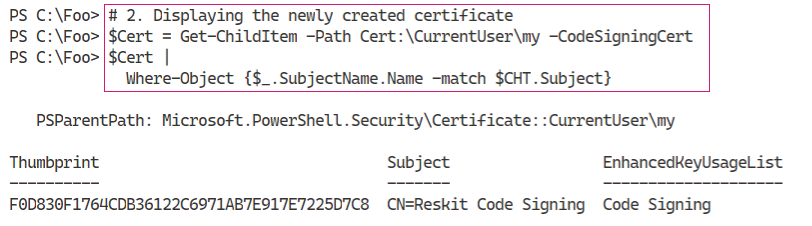
Get-AuthenticodeSignature -FilePath C:\Foo\Signed.ps1 |

  Format-List

## How it works...

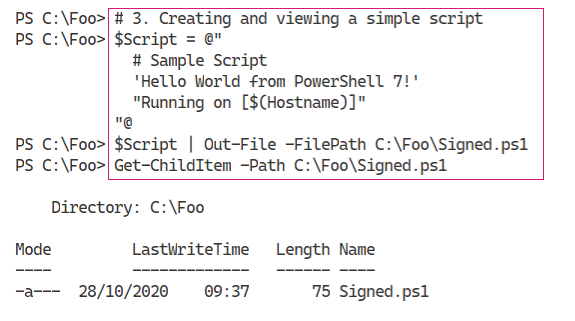
In step 1, you create a new self-signed code signing certificate and store the certificate in the current user My certificate store. Because you pipe the output from New-SelfSignedCertificate to Out-Null, this step produces no output.

In step 2, you retrieve the code signing certificate from the current user’s certificate store, then view the certificate, which looks like this:



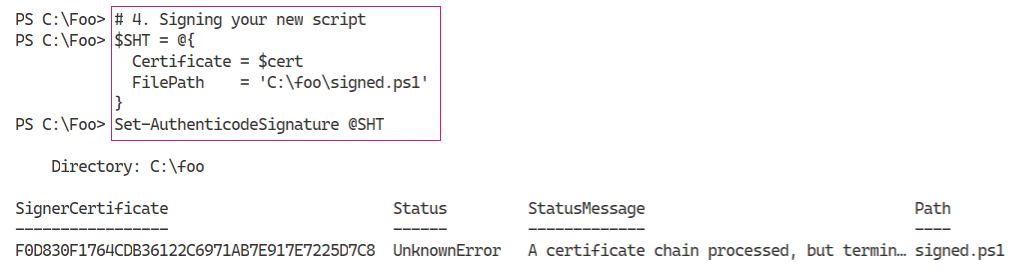
Insert image B42024\_04\_32.png

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:



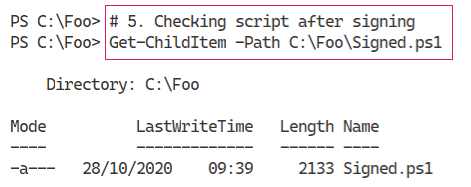
Insert image B42024\_04\_33.png

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.



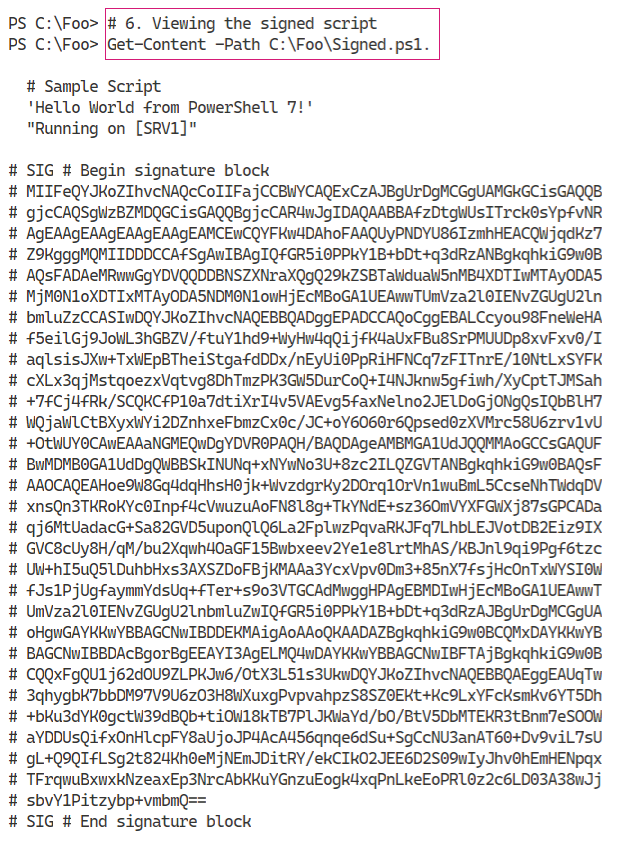
Insert image B42024\_04\_34.png

In step 5, you view the signed script, which looks like this:



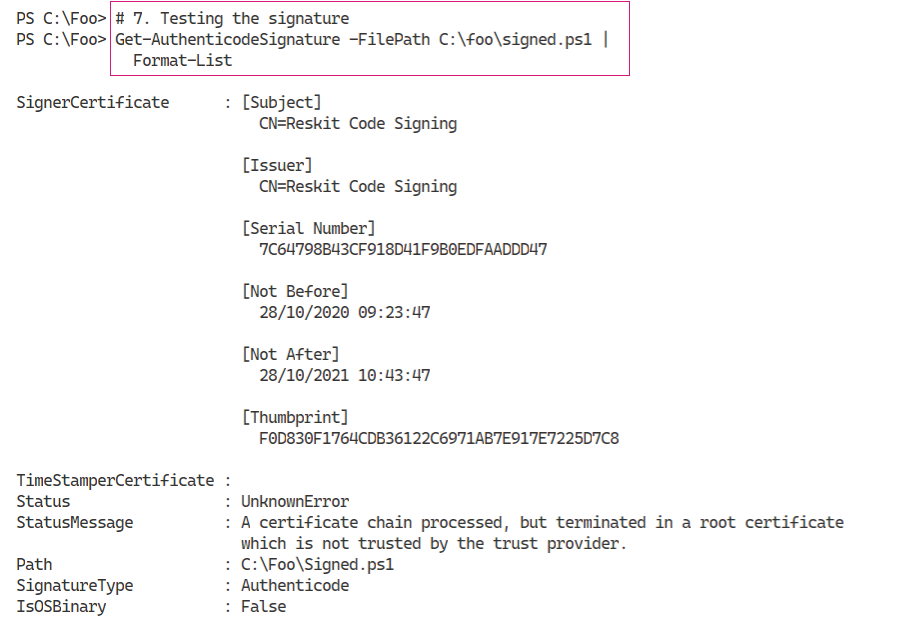
Insert image B42024\_04\_35.png

In step 6, you view the script, including the script’s digital signature, which looks like this:



Insert image B42024\_04\_36.png

In step 7, you use the Get-AuthenticodeSignature cmdlet to test the digital signature, which looks like this:



Insert image B42024\_04\_37.png

In step 8, you use the System.Security.Cryptography.X509Certificates.X509Store class to copy the signing certificate to the current user’s Trusted Publisher certificate store. Once you copy the certificate, in step 9, you re-check the Authenticode signature which looks like this:



Insert image B42024\_04\_38.png

## 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 Trusted Publisher store, which has the effect of making the code-signing certificate trusted.

In a production environment, you would obtain a code-signing from a trusted Certificate authority and manage the trusted certificate stores carefully. For enterprise environments, you could set up a certificate authority and ensure users auto-enrol for Root CA certificates. With auto-enrollment, 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 PSShortcut module

A shortcut is a file which 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 which 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 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 short cuts 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 Data Centre Edition.

## How to do it...

1. Finding the PSShortcut module

Find-Module -Name '\*shortcut'

1. Installing PSShortcut module

Install-Module -Name PSShortcut -Force

1. Reviewing PSShortcut module

Get-Module -Name PSShortCut -ListAvailable |

  Format-List

1. Discovering commands in PSShortcut module

Get-Command -Module PSShortcut

1. Discovering all shortcuts on SRV1

$SHORTCUTS = Get-Shortcut

"Shortcuts found on $(hostname): [{0}]" -f $SHORTCUTS.Count

1. Discovering PWSH shortcuts

$SHORTCUTS | Where-Object Name -match '^PWSH'

1. Discovering URL shortcut

$URLSC = Get-Shortcut -FilePath \*.url

$URLSC

1. Viewing the content of the shortcut

$URLSC | Get-Content

1. 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

1. Using the URL Shortcut

& $NEWURLSC

1. Creating a file shortcut

$CMD  = Get-Command -Name notepad.exe

$NP   = $CMD.Source

$NPSC = 'C:\Foo\NotePad.lnk'

New-Item -Path $NPSC | Out-Null

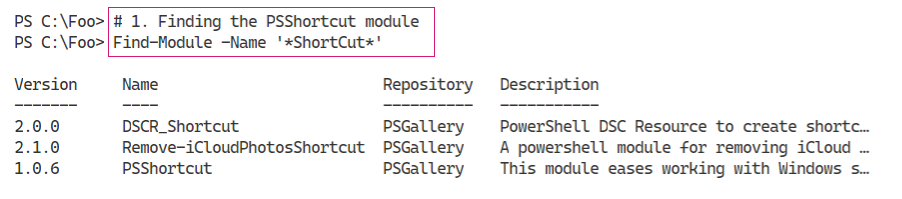
Set-Shortcut -FilePath $NPSC -TargetPath $NP

1. Using the shortcut

& $NPSC

## How it works...

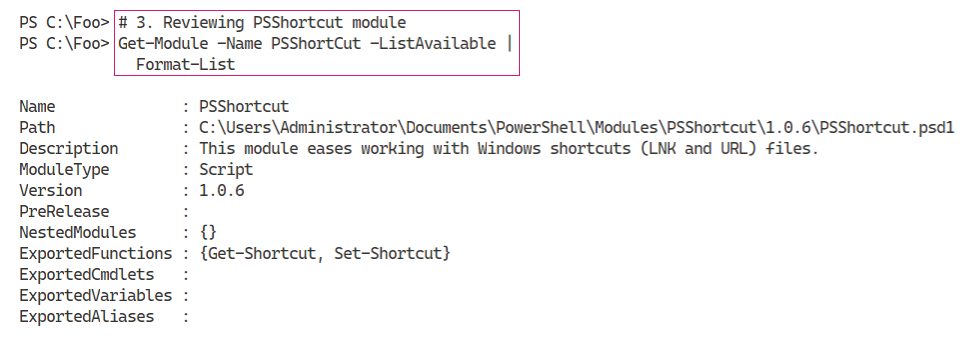
In step 1, you use the Find-Module command to discover modules in the Powershell Gallery whose name ends with “shortcut”. The output from this step looks like this:



Insert image B42024\_04\_39.png

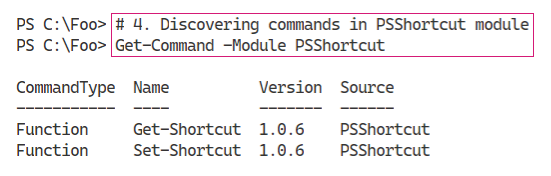
In step 2, you use Install-Module command to install the PSShortcut module. This step produces no output.

Once you have installed the PSShortcut module, in step 3, you use the Get-Module command to find more about the PSShortcut module. The output of this step looks like this:



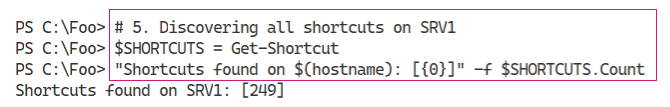
Insert image B42024\_04\_40.png

In step 4, you discover the commands provided by the PSShortcut module, which looks like this:



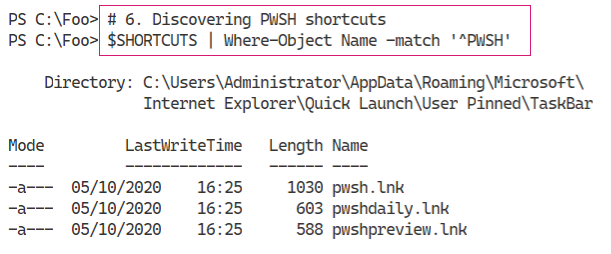
Insert image B42024\_04\_41.png

In step 5, you use the 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:



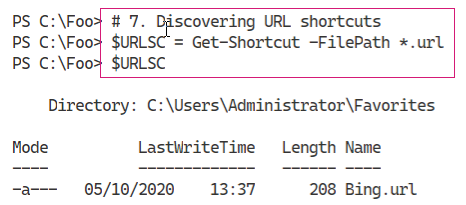
Insert image B42024\_04\_42.png

In step 6, you examine the set of link file shortcuts on SRV1 to find those that point to PWSH (that is a short cut to PowerShell 7), which looks like this:



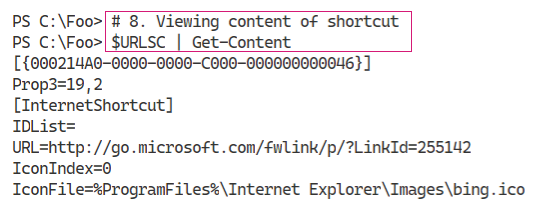
Insert image B42024\_04\_43.png

In step 7, you use Get-ShortCut to discover any URL shortcuts on SRV1, which looks like this:



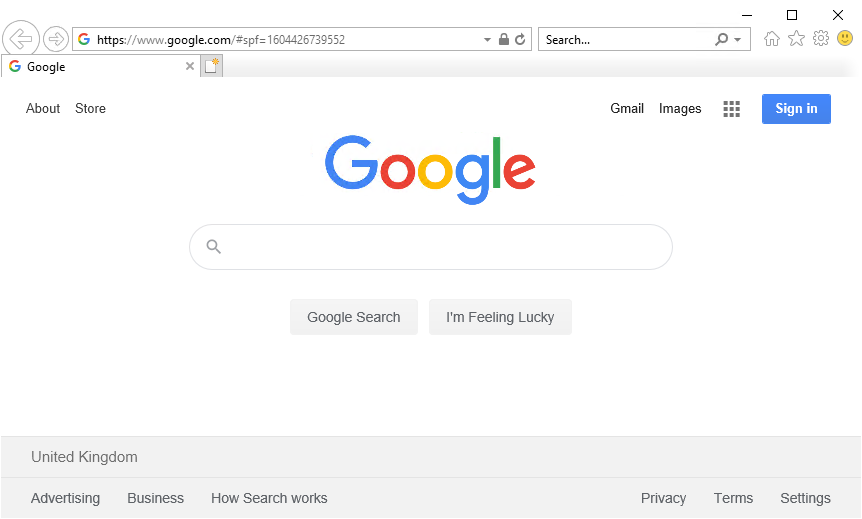
Insert image B42024\_04\_44.png

In step 8, you examine the contents of the .URL file, which looks like this:



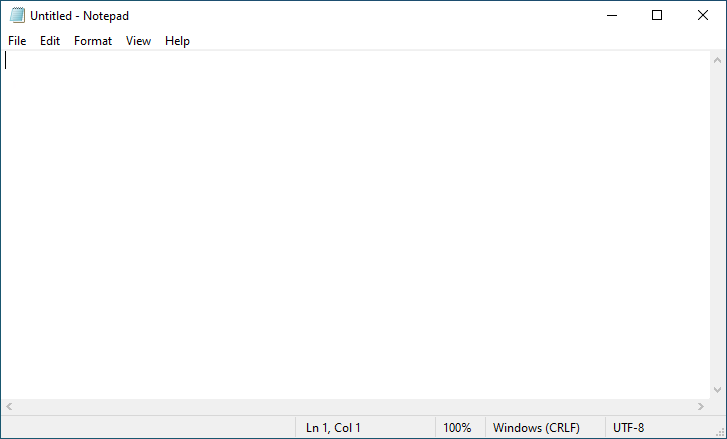
Insert image B42024\_04\_45.png

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 (i.e. Internet Explorer), and nativgates to the URL in the file, which looks like this:



Insert image B42024\_04\_46.png

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:



Insert image B42024\_04\_47.png

## There's more...

In step 7, you find any shortcuts to PowerShell 7. As you can see, there are three. These point to the release version of PowerShell 7, the preview version, and the daily build respectively. You created these shortcuts in Chapter 1.

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 2GB 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, have provided alternative compression schemes and associated utilities, including WinRAR and 7-ZIP. WinZip and WinRAR are both excellent programs but are commercial programs. 7-Zip is a freeware tool that is also popular. All three offer their own compression mechanisms (with associated file extension) 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 7Zip see https://www.7-zip.org. Each of the compression utilities offered by these groups also supports compression mechanism from other environments such as TAR.

In this recipe, you look at PowerShell 7’s built-in commands to manage archive files. The commands work only with .ZIP files. You can find a PowerShell module for 7Zip at https://github.com/thoemmi/7Zip4Powershell although the module is.

## 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

1. Discovering commands in archive module

Get-Command -Module Microsoft.PowerShell.Archive

1. Making a new folder

$NIHT = @{

  Name        = 'Archive'

  Path        = 'C:\Foo'

  ItemType    = 'Directory'

  ErrorAction = 'SilentlyContinue'

}

New-Item @NIHT | Out-Null

1. 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

}

1. Measuring 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

1. Compressing a set of files into an archive

$AFILE1 = 'C:\Foo\Archive1.zip'

Compress-Archive -Path $Files -DestinationPath "$AFile1"

1. Compressing a folder containing files

$AFILE2 = 'C:\Foo\Archive2.zip'

Compress-Archive -Path "C:\Foo\Archive" -DestinationPath $AFile2

1. Viewing the archive files

Get-ChildItem -Path $AFILE1, $AFILE2

1. Viewing archive content with Windows Explorer

explorer.exe $AFILE1

1. Viewing the second archive with Windows Explorer

explorer.exe $AFILE2

1. Making a new output folder

$Opath = 'C:\Foo\Decompressed'

$NIHT2 = @{

  Path        = $Opath

  ItemType    = 'Directory'

  ErrorAction = 'SilentlyContinue'

}

New-Item @NIHT2 | Out-Null

1. Decompressing the Archive1.zip archive

Expand-Archive -Path $AFILE1 -DestinationPath $Opath

1. Measuring decompressed files

$Files = Get-ChildItem -Path $OUTF

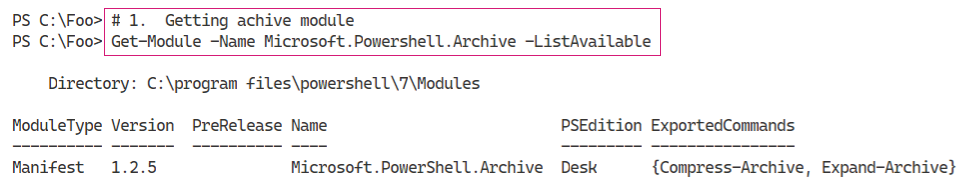
$Count = $Files.Count

$LenKB = (($Files | Measure-Object -Property length -Sum).Sum)/1mb

"[{0}] decompressed files, occupying {1:n2}mb" -f $Count, $LenKB

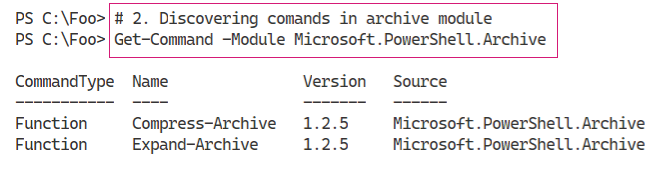
## How it works...

In step 1, you use Get-Module to examine the Microsoft.PowerShell.Archive module, which looks like this:



Insert image B42024\_04\_48.png

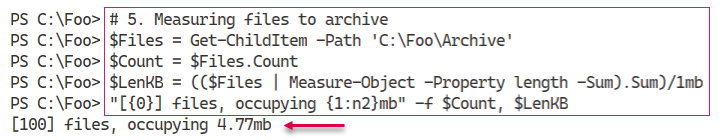
In step 2, you use Get-Command to discover the commands in the Microsoft.PowerShell.Archive module, which looks like this:



Insert image B42024\_04\_49.png

In step 4, you create a new folder which, in step 4, you populate with one hundred text files. These two steps produce no output.

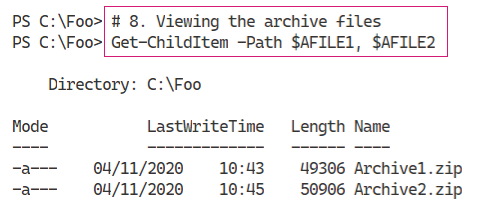
In step 5, you use Get-ChildItem to get all the files in the archive folder and measure the size of all the files. The output looks like this:



Insert image B42024\_04\_50.png

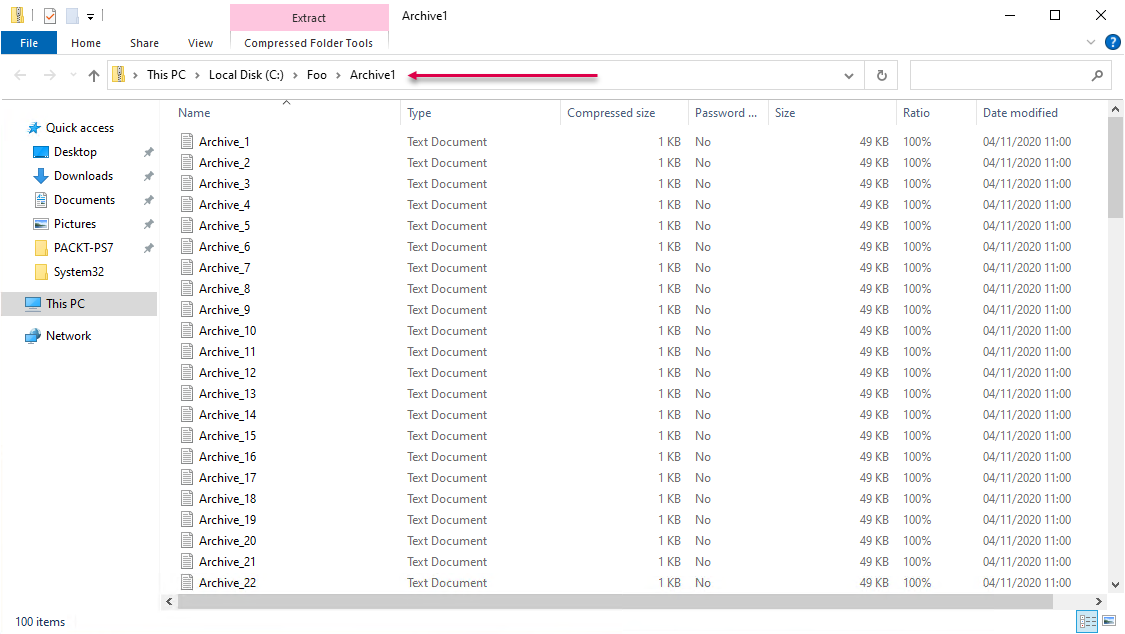
In step 6, you compress the set of files you created in step 5. This step compresses a set of files into an archive file which produces no output. In step 7, you compress a folder and it’s contents. This step creates a root folder in the archive file which holds the archived (and compressed) file. These two steps produce no output.

In step 8, you use Get-ChildItem to view the two archive files, which looks like this:



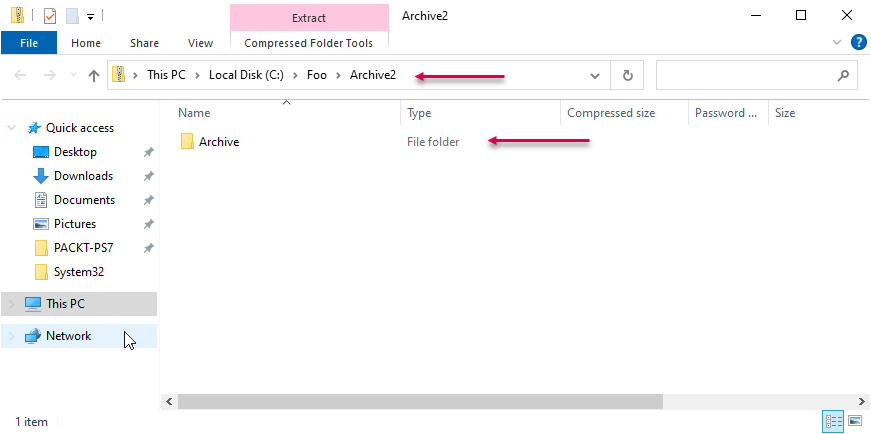
Insert image B42024\_04\_51.png

In step 9, you use Windows Explorer to view the files in the first archive files, which shows the individual files you compressed into the archive. The output from this step looks like this:



Insert image B42024\_04\_52.png

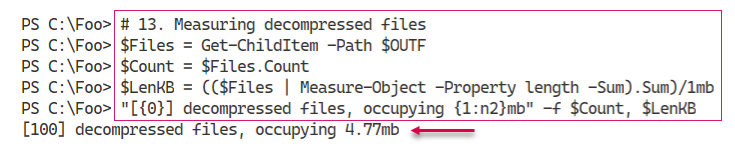
In step 10, you use Windows Explorer to view the second archive file, which looks like this:



Insert image B42024\_04\_53.png

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:



Insert image B42024\_04\_54.png

## There's more...

In step 6 and step 7, you compress one hundred 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. The second creates an archive with a root folder containing the one hundred 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.

In this chapter, you have examined several of the techniques, and PowerShell support, which you may employ in production.