2

Managing PowerShell 7 in the Enterprise

This chapter covers the following recipes:

- Utilizing Windows PowerShell Compatibility
- Installing RSAT
- Exploring Package Management
- Exploring PowerShellGet and the PS Gallery
- Creating and Using a Local PowerShell Repository
- Establishing a Script Signing Environment
- Working With Shortcuts and the PSShortCut Module
- Working With Archive Files
- Searching for Files Using the Everything Search Tool

Introduction

For an IT professional in an enterprise environment (and even in smaller ones), PowerShell 7 provides a platform to manage your environment. Many of the commands you need to carry out typical operations come with PowerShell 7, augmented by the commands provided with Windows Server feature tools. In addition, you can obtain third-party modules to extend your capabilities.

Automation, today, means using commands that come from many sources. Some are built-in, but there are gaps. Those gaps can be filled by modules you can obtain from the community and via the PowerShell Gallery. In many cases, you can make use of .NET and WMI classes.

When all else fails, there is often a command-line tool.

As you blend these tools into your workflow, you need to be aware of how a given set of commands work as well as the objects returned. You also need to deal with the mismatch. The `Get-ADComputer` command, for example, returns the name of the computer in AD in the `Name` property. Most commands that interact with a given computer use the parameter `ComputerName`.

In building Windows Server 2022, Microsoft did not update most of the role/feature management tools to support PowerShell 7. These tools work natively in Windows PowerShell but many do not work directly in PowerShell 7. The reason for this is that Microsoft based Windows PowerShell on Microsoft's .NET Framework, whereas PowerShell 7 is based on the open source .NET. The .NET team did not port all the APIs from the .NET Framework into .NET. Thus many of the role/feature commands that you can run in Windows PowerShell do not run natively since .NET did not implement a certain .NET Framework.

To get around this, PowerShell 7 comes with a Windows PowerShell compatibility solution. When you attempt to use some of the older commands (i.e., those that live in the System32 Windows directory), PowerShell 7 creates a PowerShell remoting session on the current host based on a Windows PowerShell 5.1 endpoint. Then, using implicit remoting, PowerShell creates and imports a script module of proxy functions that invoke the underlying commands in the remoting session. This enables you to use commands, such as `Add-WindowsFeature`, more or less seamlessly. You will examine the Windows PowerShell compatibility feature in *Utilizing Windows PowerShell Compatibility*.

Most of Windows Server 2022's features and roles provide management tools you can use to manage the role or feature. You can also install and use these feature tools on any host allowing you to manage features across your network, as you can see in *Installing RSAT*.

You can also find and utilize third-party modules that can improve your PowerShell experience. PowerShell implements a set of built-in package management commands, as you can see in *Exploring Package Management*.

Some organizations create their own private package repositories. These enable their organizations to share corporate modules inside the corporate network. You will create a new package repository in the *Creating and Using a Local PowerShell Repository* recipe.

PowerShell 7, like Windows PowerShell, supports the use of digitally signed scripts. You can configure PowerShell to run only properly signed scripts as a way of improving the organization's security. It is straightforward to establish a code-signing environment, using self-signed certifi-

cates, as you investigate in the *Establishing a code-signing environment* recipe.

To simplify the use of commands in an interactive environment, you may find it convenient to create shortcuts you can place on the desktop or in a folder of shortcuts. In *Working With Shortcuts and the PSShortCut Module*, you will manage shortcuts.

A common problem you can face when automating your environment is the management of archive files. As you saw in *Installing Cascadia Code font*, you can often get components delivered as a ZIP file. In *Working With Archive Files*, you examine commands for managing these files.

Windows, Windows Server, and Windows applications have grown significantly both in scope and the files that support them. Sometimes, finding key files you need can be challenging, especially on larger file servers.

The system used in the chapter

This chapter is all about using PowerShell 7 in an enterprise environment and configuring your environment to make the most out of PowerShell 7. In this chapter, you use a single host, SRV1, as follows:

Figure 2.1: Host in use for this chapter

In later chapters, you will use additional servers and will promote `SRV1` to be a domain-based server rather than being in a workgroup.

Utilizing Windows PowerShell Compatibility

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

An example of the compatibility mechanism is using `Get-WindowsFeature`, a cmdlet inside the `ServerManager` module. You use the command to get details of features that are installed, or not, inside Windows Server. You use other commands in the `ServerManager` module to install and remove features. Unfortunately, the Windows Server team has not yet updated this module to work within PowerShell 7. Via the compatibility solution, the commands in the `ServerManager` module enable you to add, remove, and view features. The Windows PowerShell compatibility mechanism allows you to use existing Windows PowerShell scripts in PowerShell 7, although with some very minor caveats.

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

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

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

And for more information on Windows PowerShell compatibility, see: `https://docs.microsoft.com/powershell/module/microsoft.powershell.core/about/about_windows_powershell_compatibility`.

Getting ready

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

How to do it...

1. Importing the `ServerManager` module

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

2. Viewing module details

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

3. Displaying a Windows feature

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

4. Running the same command in a remoting session

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

5. Getting the path to Windows PowerShell modules

```
$Paths = $env:PSModulePath -split ';'
$S32Path = $Paths |
  Where-Object {$_.ToString() -match 'system32'}
"System32 path: [$S32Path]"
```

6. Displaying path to the format XML for the Server Manager module

```
$FXML = "$S32path/ServerManager"
$FF = Get-ChildItem -Path $FXML\*.format.ps1xml
"Format XML files:"
"     $($FF.Name)"
```

7. Updating format XML in PowerShell 7

```
Foreach ($fF in $FFFf) {
    Update-FormatData -PrependPath $fF.FullName}
```

8. Using the command with improved output

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

How it works...

In *step 1*, you import the Server Manager module, with output like this:

```
PS C:\Foo> # 1. Importing the ServerManager Module
PS C:\Foo> Import-Module -Name ServerManager
WARNING: Module ServerManager is loaded in Windows PowerShell using
WinPSCompatSession remoting session; please note that all input and
output of commands from this module will be deserialized objects.
If you want to load this module into PowerShell please use
'Import-Module -SkipEditionCheck' syntax.
```

Figure 2.2: Importing the Server Manager module

In *step 2*, you use the `Get-Module` command to examine the Server Manager module's properties. The output of this step looks like this:

```
PS C:\Foo> # 2. Viewing module details
PS C:\Foo> Get-Module -Name ServerManager |
             Format-List


Name              : ServerManager
Path              : C:\Users\Administrator\AppData\Local\Temp\2\remoteIpMoProxy_ServerManager_2.0.0.0_
                    localhost_c7b5e52a-1429-4fcf-841f-d8800a63ff9f\remoteIpMoProxy_ServerManager_2.0.0.0_
                    localhost_c7b5e52a-1429-4fcf-841f-d8800a63ff9f.psm1
Description       : Implicit remoting for
ModuleType        : Script
Version           : 1.0
PreRelease        :
NestedModules     : {}
ExportedFunctions : {Disable-ServerManagerStandardUserRemoting, Enable-ServerManagerStandardUserRemoting,
                     Get-WindowsFeature, Install-WindowsFeature, Uninstall-WindowsFeature}
ExportedCmdlets   :
ExportedVariables :
ExportedAliases   : {Add-WindowsFeature, Remove-WindowsFeature}
```

Figure 2.3: Importing the Server Manager module

Now that you have loaded the Server Manager module, in *step 3*, you utilize one of the commands `Get-WindowsFeature` to view the details of a feature, with output like this:

```
PS C:\Foo> # 3. Displaying a Windows Feature
PS C:\Foo> Get-WindowsFeature -Name 'TFTP-Client'

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

Figure 2.4: Displaying the TFTP-Client feature

In *step 4*, by comparison, you run the same command inside the Windows PowerShell remoting session with slightly different results – like this:

```
PS C:\Foo> # 4. Running the same command in a remoting session
PS C:\Foo> $Session = Get-PSSession -Name WinPSCompatSession
PS C:\Foo> Invoke-Command -Session $Session -ScriptBlock {
             Get-WindowsFeature -Name 'TFTP-Client' |
               Format-Table
           }

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

Figure 2.5: Displaying the TFTP-Client feature inside the remoting session

With *step 5*, you determine and display the folder where Windows PowerShell modules can be found, with output like this:

```
PS C:\Foo> # 5. Getting the path to Windows PowerShell modules
PS C:\Foo> $Paths = $env:PSModulePath -split ';'
PS C:\Foo> $S32Path = $Paths |
             Where-Object {$_.ToString() -match 'system32'}
PS C:\Foo> "System32 path: [$S32Path]"
System32 path: [C:\WINDOWS\system32\WindowsPowerShell\v1.0\Modules]
```

Figure 2.6: Displaying the TFTP-Client feature inside the remoting session

In *step 6*, you find and display the Server Manager's display formatting XML file, with output like this:

```
PS C:\Foo> # 6. Displaying path to the format XML for Server Manager module
PS C:\Foo> $FXML = "$S32path/ServerManager"
PS C:\Foo> $FF = Get-ChildItem -Path $FXML\*.format.ps1xml
PS C:\Foo> "     $($FF.Name)"
PS C:\Foo> "Format XML files:"
Format XML files:
  Feature.format.ps1xml
```

Figure 2.7: Displaying the TFTP-Client feature inside the remoting session

In *step 7*, you import the display XML files – which produces no output. Finally, in *step 8*, you re-run the `Get-WindowsFeature` command. Since you have now imported the necessary display XML, you see output like this:

```
PS C:\Foo> # 8. Using the command with improved output
PS C:\Foo> Get-WindowsFeature -Name TFTP-Client

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

Figure 2.8: Displaying the TFTP-Client feature inside PowerShell 7 session

There's more...

In *step 3*, you used the `Get-WindowsFeature` to view one feature. When you run this command, PowerShell runs the command in the compatibility session and returns the object(s) for display in PowerShell 7. However, as you may notice, the output is not the same as you would see normally in Windows PowerShell. This is because the Windows PowerShell compatibility does not, by default, add Format-XML to the current PowerShell session.

Once you have loaded the Server Manager's display XML, as you can see in *step 8*, running the `Get-WindowsFeature` command now produces the same nicely formatted output you can see in Windows PowerShell.

Installing RSAT

The **Remote Server Admin Tools** (**RSAT**) is a set of management tools you use to manage individual Windows Features. The RSAT are fundamental to administering the roles and features you can install on Windows Server. The Windows Server DNS Server feature, for example, comes with both an MMC and a module that you use to manage the DNS Server on a given host.

A nice thing about the commands in the `ServerManager` module – everything is a feature, meaning you do not know whether the tools manage a Windows feature or a Windows role.

Each feature in Windows Server can optionally have management tools, and most do. These tools include PowerShell cmdlets, functions, aliases, GUI **Microsoft Management Console** (**MMC**) files, and Win32 console applications. The DNS Server's RSAT tools include a PowerShell module, an MMC , as well as the command-line tool `dnscmd.exe`. While you probably do not need the console applications since you can use the cmdlets, that is not always the case. And you may have some older batch 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.

As mentioned, PowerShell 7 is not installed in Windows by default, at least not at the time of writing. The PowerShell team made PowerShell 7.1 available from the Microsoft Store, which is useful to install PowerShell 7.1 or later on Windows 10/11 systems. Windows Server does not support the Microsoft store.

You have other methods of installing PowerShell 7 on your systems. The first option is to use `Install-PowerShell.ps1`, which you can download from GitHub, as shown in this recipe. You can also use this recipe on Windows 10 hosts. This approach has the advantage of being the most up-to-date source of the latest versions of PowerShell.

Getting ready

This recipe uses `SRV1`, a Windows Server workgroup host. There are no Windows features or server applications loaded on this server.

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)]"
"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 actually available on `SRV1`

```
Import-Module -Name ServerManager -WarningAction SilentlyContinue
$Features = Get-WindowsFeature
$FeaturesInstalled = $Features |
                       Where-Object Installed
$RsatFeatures = $Features |
                  Where-Object Name -Match 'RSAT'
$RsatFeaturesInstalled = $Rsatfeatures |
```

```
                    Where-Object Installed
```

7. Displaying counts of features installed

```
"On Host [$(hostname)]"
"Total features available      [{0}]" -f $Features.Count
"Total features installed      [{0}]" -f $FeaturesInstalled.Count
"Total RSAT features available [{0}]" -f $RsatFeatures.Count
"Total RSAT features installed [{0}]" -f $RsatFeaturesInstalled.
Count
```

8. Adding all RSAT tools to SRV1

```
Get-WindowsFeature -Name *RSAT* |
  Install-WindowsFeature
```

9. Getting details of RSAT tools now installed on SRV1

```
$FeaturesSRV1        = Get-WindowsFeature
$InstalledOnSRV1     = $FeaturesSRV1 | Where-Object Installed
$RsatInstalledOnSRV1 = $InstalledOnSRV1 | Where-Object Installed |
                          Where-Object Name -Match 'RSAT'
```

10. Displaying counts of commands after installing the RSAT tool

```
"After Installation of RSAT tools on SRV1"
$INS = 'Features installed on SRV1'
"$($InstalledOnSRV1.Count) $INS"
"$($RsatInstalledOnSRV1.Count) $INS"
```

11. Displaying RSAT tools on SRV1

```
$Modules = "$env:windir\system32\windowspowerShell\v1.0\modules"
$ServerManagerModules = "$Modules\ServerManager"
Update-FormatData -PrependPath "$ServerManagerModules\*.format.
ps1xml"
Get-WindowsFeature |
  Where-Object Name -Match 'RSAT'
```

12. Rebooting SRV1 and then logging on as the local administrator

```
Restart-Computer -Force
```

How it works...

In *step 1*, you use `Get-Command` to obtain all the existing commands available on `SRV1`. Then you create a count of the commands available as well as a count of the PowerShell cmdlets available to you, with output 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]
Commands available before RSAT installed [1809]
Cmdlets available before RSAT installed  [597]
```

Figure 2.9: Counting the available commands and cmdlets on SRV1

In *step 2*, you discover the different kinds of commands returned by `Get-Command`. The output 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,…
 1154 Function  {A:, Add-BCDataCacheExtension, Add-DnsClientDohServerAddress, Add…
  597 Cmdlet    {Add-AppxPackage, Add-AppxProvisionedPackage, Add-AppxVolume, Add…
```

Figure 2.10: Viewing command types returned by Get-Command

In PowerShell, the `Get-Command` cmdlet returns occurrences of different object types to describe the available commands. As you saw in the previous step, there are three command types returned by default by `Get-Command`. There is a fourth command type, **Application**, which `Get-Command` does not return by default.

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.CmdletInf
```

Figure 2.11: Determining full object type names

In *step 4*, you get the modules available on SRV1, which returns no console output. In *step 5*, you display a count of the modules discovered, with output like this:

```
PS C:\Foo> # 5. Displaying a count of modules available
PS C:\Foo> #    before adding the RSAT tools
PS C:\Foo> $CountOfModulesBeforeRSAT = $ModulesBefore.Count
PS C:\Foo> "$CountOfModulesBeforeRSAT modules available"
75 modules available
```

Figure 2.12: Displaying the number of modules available

In *step 6*, you obtain a count of the features available on SRV1, which produces no output. In the following step, *step 7*, you display counts of the total features available and the number of RSAT tool sets available. The output from this step 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 $FeaturesInstalled.count
PS C:\Foo> "Total RSAT features available [{0}]"  -f $RSATFeatures.count
PS C:\Foo> "Total RSAT features installed [{0}]"  -f $RSATFeaturesInstalled.count
On Host [SRV1]
Total features available      [266]
Total features installed      [12]
Total RSAT features available [50]
Total RSAT features installed [0]
```

Figure 2.13: Displaying the number of features and RSAT features available

In *step 8*, you install all the available features onto SRV1, with 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 2.14: Adding all RSAT tools to SRV1

In *step 9*, you get details of all the features now available on SRV1, producing no output. In *step 10*, you obtain and display a count of the features, including RSAT features, on SRV1, with output like this:

```
PS C:\Foo> # 10. Displaying counts of commands after installing the RSAT tools
PS C:\Foo> "After Installation of RSAT tools on SRV1"
PS C:\Foo> $INS = 'Features installed on SRV1'
PS C:\Foo> "$($InstalledOnSRV1.Count) $INS"
PS C:\Foo> "$($RsatInstalledOnSRV1.Count) $INS"
After Installation of RSAT tools on SRV1
76 features installed on SRV1
50 RSAT features installed on SRV1
```

Figure 2.15: Counting features and RSAT features

In *step 11*, you use `Get-WindowsFeature` to display all the RSAT tools and their installation status, with output like this:

```
PS C:\Foo> # 11. Displaying RSAT tools on SRV1
PS C:\Foo> $MODS = "$env:windir\system32\windowspowerShell\v1.0\modules"
PS C:\Foo> $SMMOD = "$MODS\ServerManager"
PS C:\Foo> Update-FormatData -PrependPath "$SMMOD\*.format.ps1xml"
PS C:\Foo> Get-WindowsFeature |
             Where-Object Name -Match 'RSAT'

Display Name                                            Name                       Install State
------------                                            ----                       -------------
[X] Remote Server Administration Tools                  RSAT                           Installed
    [X] Feature Administration Tools                    RSAT-Feature-Tools             Installed
        [X] SMTP Server Tools                           RSAT-SMTP                      Installed
        [X] BitLocker Drive Encryption Administration … RSAT-Feature-Tools-Bit…        Installed
            [X] BitLocker Drive Encryption Tools        RSAT-Feature-Tools-Bit…        Installed
            [X] BitLocker Recovery Password Viewer      RSAT-Feature-Tools-Bit…        Installed
        [X] BITS Server Extensions Tools                RSAT-Bits-Server               Installed
        [X] DataCenterBridging LLDP Tools               RSAT-DataCenterBridgin…        Installed
        [X] Failover Clustering Tools                   RSAT-Clustering                Installed
            [X] Failover Cluster Management Tools       RSAT-Clustering-Mgmt           Installed
            [X] Failover Cluster Module for Windows Po… RSAT-Clustering-PowerS…        Installed
            [X] Failover Cluster Automation Server      RSAT-Clustering-Automa…        Installed
            [X] Failover Cluster Command Interface      RSAT-Clustering-CmdInt…        Installed
        [X] Network Load Balancing Tools                RSAT-NLB                       Installed
        [X] Shielded VM Tools                           RSAT-Shielded-VM-Tools         Installed
        [X] SNMP Tools                                  RSAT-SNMP                      Installed
        [X] Storage Migration Service Tools             RSAT-SMS                       Installed
        [X] Storage Replica Module for Windows PowerSh… RSAT-Storage-Replica           Installed
        [X] System Insights Module for Windows PowerSh… RSAT-System-Insights           Installed
        [X] WINS Server Tools                           RSAT-WINS                      Installed
    [X] Role Administration Tools                       RSAT-Role-Tools                Installed
        [X] AD DS and AD LDS Tools                      RSAT-AD-Tools                  Installed
            [X] Active Directory module for Windows Po… RSAT-AD-PowerShell             Installed
            [X] AD DS Tools                             RSAT-ADDS                      Installed
                [X] Active Directory Administrative Ce… RSAT-AD-AdminCenter            Installed
                [X] AD DS Snap-Ins and Command-Line To… RSAT-ADDS-Tools                Installed
            [X] AD LDS Snap-Ins and Command-Line Tools  RSAT-ADLDS                     Installed
        [X] Hyper-V Management Tools                    RSAT-Hyper-V-Tools             Installed
        [X] Remote Desktop Services Tools               RSAT-RDS-Tools                 Installed
            [X] Remote Desktop Gateway Tools            RSAT-RDS-Gateway               Installed
            [X] Remote Desktop Licensing Diagnoser Too… RSAT-RDS-Licensing-Dia…        Installed
        [X] Windows Server Update Services Tools        UpdateServices-RSAT            Installed
        [X] Active Directory Certificate Services Tools RSAT-ADCS                      Installed
            [X] Certification Authority Management Too… RSAT-ADCS-Mgmt                 Installed
            [X] Online Responder Tools                  RSAT-Online-Responder          Installed
        [X] Active Directory Rights Management Service… RSAT-ADRMS                     Installed
        [X] DHCP Server Tools                           RSAT-DHCP                      Installed
        [X] DNS Server Tools                            RSAT-DNS-Server                Installed
        [X] Fax Server Tools                            RSAT-Fax                       Installed
        [X] File Services Tools                         RSAT-File-Services             Installed
            [X] DFS Management Tools                    RSAT-DFS-Mgmt-Con              Installed
            [X] File Server Resource Manager Tools      RSAT-FSRM-Mgmt                 Installed
            [X] Services for Network File System Manag… RSAT-NFS-Admin                 Installed
        [X] Network Controller Management Tools         RSAT-NetworkController         Installed
        [X] Network Policy and Access Services Tools    RSAT-NPAS                      Installed
        [X] Print and Document Services Tools           RSAT-Print-Services            Installed
        [X] Remote Access Management Tools              RSAT-RemoteAccess              Installed
            [X] Remote Access GUI and Command-Line Too… RSAT-RemoteAccess-Mgmt         Installed
            [X] Remote Access module for Windows Power… RSAT-RemoteAccess-Powe…        Installed
        [X] Volume Activation Tools                     RSAT-VA-Tools                  Installed
```

Figure 2.16: Displaying RSAT features on SRV1

In the final step in this recipe, *step 12*, you reboot the host creating no console output.

There's more...

The output from *step 1* shows there are 1,809 total commands and 597 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 specific Windows Server version.

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 search engine to discover more details about each of these command types. There are two further types of command (**ExternalScript** and **Application**) but these are not returned by default.

In *step 8*, you install all RSAT tools and as you can see, Windows requires a reboot to complete the installation of the tools. Later, in *step 12*, you reboot `SRV1` to complete the installation of these tools.

In *step 10*, you create the `$INS` variable, which you later use in the display of the results. This technique enables you to see the code without worrying about line breaks in the book!

In *step 11*, you display all the RSAT tools using `Get-WindowsFeature`. Strictly speaking, this step displays the Windows features that have "RSAT" somewhere in the feature name. As you can see, there is no standard for actual feature names. The output shows the individual tools as either Feature Administration tools or Role Administration tools, although that difference is not relevant in your using the Server Manager cmdlets.

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, provides an API to package management providers such as PowerShellGet, discussed in the *Exploring PowerShellGet and 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, including the PowerShell Gallery.

The modules in the various internet repositories, including the PowerShell Gallery, vary in quality. Some are excellent and are heavily used by the community, while others are less useful or of lower quality. Some are written by the PowerShell product team. 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` after you have installed PowerShell 7.

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 5 packages discovered

```
$PACKAGES |
    Select-Object -First 5 |
      Format-Table -AutoSize -Wrap
```

6. Installing the `ChocolateyGet` provider

```
Install-PackageProvider -Name ChocolateyGet -Force |
  Out-Null
```

7. Verifying `ChocolateyGet` is in the list of installed providers

```
Import-PackageProvider -Name ChocolateyGet
Get-PackageProvider -ListAvailable |
  Select-Object -Property Name,Version
```

8. Discovering packages using the `ChocolateyGet` provider

```
$CPackages = Find-Package -ProviderName ChocolateyGet -Name *
"$($CPackages.Count) packages available via ChocolateyGet"
```

How it works...

In *step 1*, you use `Get-Command` to discover the cmdlets in the `PackageManagement` module, with output 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 2.17: The PowerShell 7 console

In *step 2*, you use the `Get-PackageProvider` cmdlet to view the package providers that you have loaded and imported, with output 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 2.18: Reviewing installed package providers

In *step 3*, you use the `Find-PackageProviders` cmdlet to find additional package providers. 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.
ChocolateyGet            Package Management (OneGet) provider that facilitates installing
                         Chocolatey packages from any NuGet repository.
ContainerImage           This is a PackageManagement provider module which helps in discovering,
                         downloading and installing Windows Container OS images.
                         For more details and examples refer to our project site
                         at https://github.com/PowerShell/ContainerProvider.
NanoServerPackage        A PackageManagement provider to  Discover, Save and Install Nano Server
                         Packages on-demand
Chocolatier              Package Management (OneGet) provider that facilitates installing
                         Chocolatey packages from any NuGet repository.
WinGet                   Package Management (OneGet) provider that facilitates installing
                         WinGet packages from any NuGet repository.
DotNetGlobalToolProvider OneGet package provider for dotnet global tools.
```

Figure 2.19: Discovering additional package providers

You can use the `Find-Package` cmdlet to discover packages that you can download, install, and use. In *step 4*, you discover packages available with output that resembles 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 7,880 packages
```

Figure 2.20: Counting available packages

In *step 5*, you view the details of five (the first five) packages you discovered in the prior step. The output looks like this:

```
PS C:\Foo> # 5. Showing the 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.
DellBIOSProvider    2.6.0   PSGallery The 'Dell Command | PowerShell Provider' provides native configuration capability of Dell
                                      Optiplex, Latitude, Precision, XPS Notebook and Venue 11 systems within PowerShell.
PSWindowsUpdate     2.2.0.3 PSGallery This module contain cmdlets to manage Windows Update Client.
NetworkingDsc       8.2.0   PSGallery DSC resources for configuring settings related to networking.
PackageManagement   1.4.7   PSGallery PackageManagement (a.k.a. OneGet) is a new way to discover and install software packages from
                                      around the web.
                                       It is a manager or multiplexor of existing package managers (also called package providers)
                                      that unifies Windows package management with a single Windows PowerShell interface. With
                                      PackageManagement, you can do the following.
                                        - Manage a list of software repositories in which packages can be searched, acquired and
                                      installed
                                        - Discover software packages
                                        - Seamlessly install, uninstall, and inventory packages from one or more software repositories
```

Figure 2.21: Viewing package summaries

In *step 6*, you install a new package provider – the `ChocolateyGet` provider. This step produces no output. In *step 7*, you view the currently imported package providers, with output like this:

```
PS C:\Foo> # 7. Verifying ChocolateyGet is in the list of installed providers
PS C:\Foo> Import-PackageProvider -Name ChocolateyGet
PS C:\Foo> Get-PackageProvider -ListAvailable |
             Select-Object -Property Name,Version

Name          Version
----          -------
ChocolateyGet 4.0.0.0
NuGet         3.0.0.1
PowerShellGet 2.2.5.0
PowerShellGet 1.0.0.1
```

Figure 2.22: Viewing package summaries

In the final step in this recipe, *step 8*, you discover all the currently available packages you can download using the `ChocolatelyGet` package provider – the output looks like this:

```
PS C:\Foo> # 8. Discovering Packages using the ChocolateyGet provider
PS C:\Foo> $CPackages = Find-Package -ProviderName ChocolateyGet -Name *
PS C:\Foo> "$($CPackages.Count) packages available via ChocolateyGet"
6842 packages available via ChocolateyGet
```

Figure 2.23: Counting packages available via the ChocolateyGet provider

There's more...

In *step 4,* you display a count of the packages available in the PS Gallery. By the time you read this, the number of packages is certain to rise. Package authors in the community are constantly adding new and improved packages to the PS Gallery. At the same time, since some users may have developed some automation on top of older providers, these too remain in the PS Gallery.

In *step 7*, you install the `ChocolateyGet` packaged provider. With this provider, you can discover and install packages from the Chocolatey repository in addition to the PowerShell Gallery. In *Chapter 1*, you used the Chocolatey command-line tool while in *step 8*, you find packages using the `Find-Package` cmdlet. The ChocolateyGet package provider is a wrapper around the command-line tool – and probably a lot easier to use for a simple case. The PowerShell Gallery and Chocolatey have many packages in common so you can probably find whatever packages you might need in either or both.

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 Windows Server features, may not have every command you need. At a practical level, the various Windows feature teams, as well as the PowerShell team, do not have infinite resources to implement every good idea. And that is where the PowerShell community and the **PowerShell Gallery** (**PS Gallery**) come in.

The PS Gallery provides a huge range of PowerShell artifacts for you to leverage. To some degree, if you can conceive of a use for PowerShell, there are likely to be things in the Gallery to delight you. If you are a Grateful Dead fan, for example, there are some scripts to manage your live concert collection (`https://www.powershellgallery.com/packages/gdscripts/1.0.4`). If you need to generate nice-looking HTML reports, then you can use the PSWriteHTML module (`https://www.powershellgallery.com/packages/PSWriteHTML/0.0.173`).

PowerShellGet is a module that enables you to discover, install, update, and publish key PowerShell artifacts found in the PowerShell Gallery. The artifacts you can leverage include modules, scripts, and more. You can think of this module as an abstraction on top of the Package Management tools. You can read more about the module at: `https://docs.microsoft.com/powershell/module/powershellget`.

Getting ready

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

How to do it...

1. Reviewing the commands available in the PowerShellGet module

```
Get-Command -Module PowerShellGet
```

2. Discovering Find-* cmdlets in PowerShellGet module

```
Get-Command -Module PowerShellGet -Verb Find
```

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

```
$Commands     = Find-Command
$Modules      = Find-Module
$DSCResources = Find-DscResource
$Scripts      = Find-Script
```

4. Reporting on results

```
"On Host [$(hostname)]"
"Commands found:          [{0:N0}]"  -f $Commands.Count
"Modules found:           [{0:N0}]"  -f $Modules.Count
"DSC Resources found:     [{0:N0}]"  -f $DSCResources.Count
"Scripts found:           [{0:N0}]"  -f $Scripts.Count
```

5. Discovering NTFS-related modules

```
$Modules |
  Where-Object Name -match NTFS
```

6. Installing the `NTFSSecurity` module

```
Install-Module -Name NTFSSecurity -Force
```

7. Reviewing module contents

```
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 `PSLogging` module

```
Save-Module -Name PSLogging -Path $DLFLDR
```

11. Viewing the contents of the download folder

```
Get-ChildItem -Path $DownloadFolder -Recurse -Depth 2 |
  Format-Table -Property FullName
```

12. Importing the `PSLogging` module

```
$ModuleFolder = Join-Path -Path $DownloadFolder -ChildPath
'PSLogging'
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 PSLogging
```

How it works...

In *step 1*, you use the `Get-Command` cmdlet to investigate the commands in the PowerShellGet module, with output that should look 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 2.24: The PowerShell 7 console

In *step 2*, you discover the cmdlets in the `PowerShellGet` module that have the `Find` verb, with output 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 2.25: Discovering the Find- cmdlets in the PowerShellGet module*

In *step 3*, which generates no output, you find all the commands, modules, DSC resources, and scripts in the PowerShell Gallery. In *step 4*, you report on the number of objects found in the PS Gallery, with output like this:

```
PS C:\Foo> # 4. Reporting on results
PS C:\Foo> "On Host [$(hostname)]"
PS C:\Foo> "Commands found:          [{0:N0}]"  -f $Commands.Count
PS C:\Foo> "Modules found:           [{0:N0}]"  -f $Modules.Count
PS C:\Foo> "DSC Resources found:     [{0:N0}]"  -f $DSCResources.Count
PS C:\Foo> "Scripts found:           [{0:N0}]"  -f $Sripts.Count

On Host [SRV1]
Commands found:          [146,638]
Modules found:           [7,886]
DSC Resources found:     [2,146]
Scripts found:           [1,545]
```

Figure 2.26: Discovering the Find- cmdlets in the PowerShellGet module*

In *step 5*, you examine the modules returned (in *step 3)* to see if any of them have "NTFS" in the module name. The output looks like this:

```
PS C:\Foo> # 5. Discovering NTFS-related modules
PS C:\Foo> $Modules |
             Where-Object Name -match NTFS

Version    Name                     Repository  Description
-------    ----                     ----------  -----------
4.2.6      NTFSSecurity             PSGallery   Windows PowerShell M…
1.4.1      cNtfsAccessControl       PSGallery   The cNtfsAccessContr…
1.0        NTFSPermissionMigration  PSGallery   This module is used …
```

Figure 2.27: Discovering NTFS-related modules in the PowerShell Gallery

Having discovered an NTFS-related module, in *step 6*, you install the module. This step creates no output. In *step 7*, you use `Get-Command` to discover the commands in the `NTFSSecurity` module, with output like this:

```
PS C:\Foo> # 7. Reviewing module contents
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 2.28: Discovering commands in the NTFSSecurity module

In *step 8*, you use a command from the `NTFSSecurity` module, `Get-NTFSAccess`, to view the access control list on the `C:\Foo` folder, which looks like this:

```
S C:\Foo> # 8. Testing the Get-NTFSAccess cmdlet
PS C:\Foo> Get-NTFSAccess -Path C:\Foo

    Path: C:\Foo (Inheritance enabled)

Account                 Access Rights     Applies to                Type   IsInherited   InheritedFrom
-------                 -------------     ----------                ----   -----------   -------------
NT AUTHORITY\SYSTEM     FullControl       ThisFolderSubfoldersAndF… Allow  True          C:
BUILTIN\Administrators  FullControl       ThisFolderSubfoldersAndF… Allow  True          C:
BUILTIN\Users           ReadAndExecute, … ThisFolderSubfoldersAndF… Allow  True          C:
BUILTIN\Users           CreateDirectories ThisFolderAndSubfolders   Allow  True          C:
BUILTIN\Users           CreateFiles       ThisFolderAndSubfolders   Allow  True          C:
CREATOR OWNER           GenericAll        SubfoldersAndFilesOnly    Allow  True          C:
```

Figure 2.29: Using Get-NTFSAccess

As an alternative to installing fully a PS Gallery module, you might find it useful to download a module into a new folder. From there, you can investigate (and possibly improve it) before installing it in your environment. To do that, in *step 9*, you create a new download folder, creating no output. In *step 10*, you download the `PSLogging` module, which also produces no output.

In *step 11*, you discover the files contained in the `PSLogging` module with output like this:

```
PS C:\Foo> # 11. Viewing the contents of the download folder
PS C:\Foo> Get-ChildItem -Path $DownloadFolder -Recurse |
             Format-Table -Property FullName

FullName
--------
C:\Foo\DownloadedModules\PSLogging
C:\Foo\DownloadedModules\PSLogging\2.5.2
C:\Foo\DownloadedModules\PSLogging\2.5.2\PSLogging.psd1
C:\Foo\DownloadedModules\PSLogging\2.5.2\PSLogging.psm1
```

Figure 2.30: Discovering files contained in the PSLogging module

Finally, in *step 12*, you check the commands contained in this module with output like this:

```
PS C:\Foo> # 12. Checking commands in the module
PS C:\Foo> Get-Command -Module PSLogging

CommandType Name              Version  Source
----------- ----              -------  ------
Function    Send-Log          0.0      PSLogging
Function    Start-Log         0.0      PSLogging
Function    Stop-Log          0.0      PSLogging
Function    Write-LogError    0.0      PSLogging
Function    Write-LogInfo     0.0      PSLogging
Function    Write-LogWarning  0.0      PSLogging
```

Figure 2.31: Discovering files contained in the PSLogging module

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 you can search for in the Gallery:

- Command – these are individual commands contained within the gallery. You can use `Find-Command` to help you discover the name of a module that might contain a command.
- Module – these are PowerShell modules. Some of which may not work in PowerShell 7 and some that may not work at all.
- DSC Resource – these are Windows PowerShell DSC resources. PowerShell 7 does not provide the rich DSC functions and features available with Windows PowerShell at this time.
- Script – these are complete PowerShell scripts. You can use them as-is or adapt them to your needs.
- Role Capability – this resource was meant for packages that enhance Windows roles but is little used in practice.

In *steps 3* and *step 4*, you discover the number of commands, modules, DSC Resources, and PowerShell 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. Note that step 3 may take several minutes. If you plan to search for multiple modules, then downloading the details of *all* the modules makes subsequent searching quicker.

In *step 5*, you search the PS Gallery for modules whose name includes the string "NTFS". The modules returned may or may not be helpful for you. As you can see, the search returns the `NTFSSecurity` module that, in *steps 6* through *step 8*, you install and use.

Before you start using a PS Gallery module, it is a great idea to first download it and test it. In *step 9*, you create a new folder, and then in *step 10*, you download the module into this new folder. In *step 11*, you observe the contents of the module.

Since you have downloaded the module to a folder, not on the `PSModulePath` environment variable, PowerShell cannot find it automatically. Therefore, in *step 12*, you import the module manually. You then look at the commands available within the module.

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. Some of those objects are not of the highest quality or may be of no use to you. But 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 the best practices suggested in any production script you develop.

Creating and Using a Local Package 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 update the module using `Update-Module`.

As an alternative to using a public repository, you can create your own private 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 methods you can use to set up your internal package 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 command `Register-PSRepository` to enable you to use the PowerShellGet commands to view this share as a PowerShell repository. You must register this repository for every host from which you access your 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

Run this recipe on `SRV1` after you have loaded PowerShell 7.

How to do it...

1. Creating a new repository folder

```
$PATH = 'C:\RKRepo'
New-Item -Path $PATH -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 a Hello World module folder

```
$HWDIR = 'C:\HW'
New-Item -Path $HWDIR -ItemType Directory | Out-Null
```

6. Creating a very simple module

```
$HS = @"
Function Get-HelloWorld {'Hello World'}
Set-Alias –[Name GHW -=Value 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 new folder on SRV1 to hold your new PowerShell repository, producing no output. In *step 2*, you use the New-SmbShare cmdlet to create a new share, with output like this:

```
PS C:\Foo> # 2. Sharing the folder
PS C:\Foo> $SMBHT = @{
             Name        = 'RKRepo'
             Path        = $PATH
             Description = 'Reskit Repository'
             FullAccess  = 'Everyone'
           }
PS C:\Foo> New-SmbShare @SMBHT

Name   ScopeName Path      Description
----   --------- ----      -----------
RKRepo *         C:\RKRepo Reskit Repository
```

Figure 2.32: The PowerShell 7 console

In *step 3*, you register this new SMB share as a new PowerShell repository, which creates no console output. In *step 4*, you view the configured PowerShell repositories, with output like this:

```
PS C:\Foo> # 4. Viewing configured repositories
PS C:\Foo> Get-PSRepository

Name       InstallationPolicy  SourceLocation
----       ------------------  --------------
RKRepo     Trusted             \\SRV1\RKRepo
PSGallery  Untrusted           https://www.powershellgallery.com/api/v2
```

Figure 2.33: Viewing configured repositories

To demonstrate using this new repository, in *step 5*, you first create a folder, and then in *step 6*, you create a simple module in this folder. These two steps produce no console output.

In *step 7*, you import and use the module, with output like this:

```
PS C:\Foo> # 7. Testing the module locally
PS C:\Foo> Import-Module -Name $HWDIR\HW.PSM1 -Verbose
VERBOSE: Importing function 'Get-HelloWorld'.
VERBOSE: Importing alias 'GHW'.
PS C:\Foo> GHW
Hello World
```

Figure 2.34: Testing HW module

In *step 8*, you create a module manifest, and in *step 9*, you publish the module to your repository. These two steps produce no output. In *step 10*, you use `Find-Module` to find the modules in your repository, with output like this:

```
PS C:\Foo> # 10. Viewing the results of publishing
PS C:\Foo> Find-Module -Repository RKRepo

Version   Name  Repository   Description
-------   ----  ----------   -----------
1.0.1     HW    RKRepo       Hello World module
```

Figure 2.35: Viewing the modules in the new repository

In *step 11*, you view the repository's home folder, with output like this:

```
PS C:\Foo> # 11. Checking the repository's home folder
PS C:\Foo> Get-ChildItem -Path $LPATH

    Directory: C:\RKRepo

Mode          LastWriteTime   Length  Name
----          -------------   ------  ----
-a---  25/04/2022    17:38     3462  HW.1.0.1.nupkg
```

Figure 2.36: Testing HW module

There's more...

In *step 1*, you create a folder on the `C:\` drive to act as a repository that you then share. In production, you should probably put this share in a highly-available system with redundant disk drives.

In *step 3*, you set up an SMB share to act as a PowerShell repository on `SRV1`. You may need to adjust the value of the `$Path` variable in this step if you are testing this recipe on a host with a different name. Also, if you plan to use this repository on other systems, you need to explicitly register this repository on each host since registering a repository works on a system-by-system basis.

In *step 11*, you can see the NuGet package, which is your module. When you published the module, the cmdlet created and stored the NuGet package in the shared folder.

Establishing a Script Signing Environment

You can often find that it is essential to know if an application, or a PowerShell script, has been modified since it was released. You can use Windows Authenticode Digital Signatures for this purpose.

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 sign a PowerShell script, you can set PowerShell's execution policy to 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 (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 (or the Windows PowerShell ISE), select all the text in the script, then run that selected text. If an Execution policy of `RemoteSigned` is blocking a particular script you downloaded from the internet or from the PS Gallery, 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 that 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 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 after you have installed PowerShell 7.

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. Setting the execution policy to all signed for this process

```
Set-ExecutionPolicy -ExecutionPolicy AllSigned -Scope Process
```

10. Running the signed script

```
C:\Foo\Signed.ps1
```

11. Copying certificate to the 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 the 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
```

16. Resetting the Execution policy for this process

```
Set-ExecutionPolicy -ExecutionPolicy Unrestricted -Scope Process
```

How it works...

In *step 1*, you create a new self-signed certificate for code signing. This step creates no output. In *step 2*, you display the newly created certificate, with output 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
----------                                -------              --------------------
55694D1AA117028A739E9F5CC7AEBBB1209D45E5  CN=Reskit Code Sign… Code Signing
```

Figure 2.37: Displaying the certificate

In *step 3*, you create a very simple script and view the script file's details, with the following output:

```
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---          26/04/2022    17:22         78 Signed.ps1
```

Figure 2.38: Displaying the certificate

With *step 4*, you sign the script using your newly created code-signing certificate. The output is as follows:

```
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
-----------------                         ------       -------------                                  ----
55694D1AA117028A739E9F5CC7AEBBB1209D45E5  UnknownError A certificate chain processed, but              signed.ps1
                                                       terminated in a root certificate which
                                                       is not trusted by the trust provider.
```

Figure 2.39: Signing your script

In *step 5*, you view the file details for your newly signed script, with output 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---          26/04/2022    17:25           2136 Signed.ps1
```

Figure 2.40: Signing your script

Next, in *step 6*, you view the contents of the script file, with output like this:

```
PS C:\Foo> # 6. Viewing the signed script
PS C:\Foo> Get-Content -Path C:\Foo\Signed.ps1

  # Sample Script
  'Hello World from PowerShell 7!'
  "Running on [SRV1]"

# SIG # Begin signature block
# MIIFeQYJKoZIhvcNAQcCoIIFajCCBWYCAQExCzAJBgUrDgMCGgUAMGkGCisGAQQB
# gjcCAQSgWzBZMDQGCisGAQQBgjcCAR4wJgIDAQAABBAfzDtgWUsITrck0sYpfvNR
# AgEAAgEAAgEAAgEAAgEAMCEwCQYFKw4DAhoFAAQUiXIZ7que8HGeN6y3v9batOmI
# VzygggMQMIIDDDCCAfSgAwIBAgIQESy8ARMRF6RNfapBk//p+zANBgkqhkiG9w0B
# AQsFADAeMRwwGgYDVQQDDBNSZXNraXQgQ29kZSBTaWduaW5nMB4XDTIyMDQyNjE2
# MTExM1oXDTIzMDQyNjE2MzExM1owHjEcMBoGA1UEAwwTUmVza2l0IENvZGUgU2ln
# bmluZzCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAMQkI0Zo8ctEiPwd
# R5LdCw7QQ0WVK6gsqAvgXOpMVYe9V70Y+h3jyomxA3svRAKJVriyfsp+uEaCN5WS
# aT863u553qMBYkgvokJzmQIxWJVO1B1dBbSWutn9wihSjzaqUm3KWh/k8XS8ow9W
# KvWChT7Vn5fmsfX/6tXtmJxaG47DirWZUZ/9V4RJADQL24RT5SsE16WboCKanRAL
# can6OJK8OCbIQvM0uXne+7bv1n+3x+ZfBptJFB8Sf0CfGiUYe/q3P1rWZS+do1hd
# bPfRxca44SU4zxLSnllvIXeQgCTPe93NHPOFkgDblizhtbjFSF03SOdxj7UmOE1/
# RQcLy00CAwEAAaNGMEQwDgYDVR0PAQH/BAQDAgeAMBMGA1UdJQQMMAoGCCsGAQUF
# BwMDMB0GA1UdDgQWBBSbIu3CywcdPmTZRkdvE647Q6XrXjANBgkqhkiG9w0BAQsF
# AAOCAQEAphlEvSTYsxdBYWir+/3ShQ05454IfIIgAtNar9RTGmdobpeMdGTWIglS
# mXRUvnTGIWNPqEdd3m+Pm9hUwuW4Av/FOlMwIE0WsqvS5MoepKY6oinpaa4rUDZi
# XbLBTdutNTa4u7YExjVxYgOUXmdbGG0l/OjiWmqQ7BC40zmJr1LrJ+NfQkMDf9LR
# VjkrLX1MD/V6ZqZtJK57XjmX1VmiZeS0pNNpvvSIGE7O/N37Bf9iBmg2HkLz/cmO
# Aqo6IrIHmK3UuG1VEjCPh7479KqwSEcqkGaVWTBuuA4xhfl5mjJXaK17MqCYD6bZ
# BpTrBLs1G3ULSeCqzVhSXPGGnSQ5AzGCAdMwggHPAgEBMDIwHjEcMBoGA1UEAwwT
# UmVza2l0IENvZGUgU2lnbmluZwIQESy8ARMRF6RNfapBk//p+zAJBgUrDgMCGgUA
# oHgwGAYKKwYBBAGCNwIBDDEKMAigAoAAoQKAADAZBgkqhkiG9w0BCQMxDAYKKwYB
# BAGCNwIBBDAcBgorBgEEAYI3AgELMQ4wDAYKKwYBBAGCNwIBFTAjBgkqhkiG9w0B
# CQQxFgQUtXvYPUYhpvHmSjkz4NMrIU+3M6QwDQYJKoZIhvcNAQEBBQAEggEAAo7K
# r2w2oNathBRiZ4mohR44DqMFgq5dfkbW9+wWg1Q5C2opNSWFPbKJWJJWRHv7r21P
# 0t0myr/TGo1MQO93afZPJvpeDP1pA/Gc+VNQJwk3YG6yl+HnhthaxuyNzHCZqDdX
# ydzTO5VazDdHIOUT63ZN8RDi7JZQ0lXHiPxSbpkoIvAkQJ5NPK6X/Rbi8YOu69/U
# Q71jptE0I9HYL8htjns0Mp41DHJp0pKPsL5JmNyilj6xj/vad0RrrPvrHhKbXWOP
# TAwvZro2J6BK0zSucHNnOX2AYpGdnvkVOYnCEa60nQg1EXBvkYH8M4PoY457UlCy
# qN+2J0RqrkyF1AF85Q==
# SIG # End signature block
```

Figure 2.41: Viewing the signed script

In *step 7*, you use the `Get-AuthenticodeSignature` to get the details of the file signature and view the results, as you can see here:

```
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]
                            112CBC01131117A44D7DAA4193FFE9FB

                          [Not Before]
                            26/04/2022 17:11:13

                          [Not After]
                            26/04/2023 17:31:13

                          [Thumbprint]
                            55694D1AA117028A739E9F5CC7AEBBB1209D45E5

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 2.42: Viewing details of the script's digital signature

In *step 8*, you run the script with output 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 2.43: Running the script

In *step 9*, you change the current process's Execution policy to `AllSigned`, which generates no console output. In *step 10* you run the script again with the output like this:

```
PS C:\Foo> # 10. Running the signed script
PS C:\Foo> C:\Foo\Signed.ps1
C:\Foo\Signed.ps1
Line |
   2 |  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 2.44: Running the script

In *step 11*, you copy the self-signed certificate to the current user's trusted root CA certificate store. There is no output from the console, but when PowerShell attempts to add the certificate, Windows opens a security warning message box that looks like this:

Figure 2.45: Authorizing the addition of the certificate

In *step 12*, you recheck the script's digital signature, 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]
                            112CBC01131117A44D7DAA4193FFE9FB

                          [Not Before]
                            26/04/2022 17:11:13

                          [Not After]
                            26/04/2023 17:31:13

                          [Thumbprint]
                            55694D1AA117028A739E9F5CC7AEBBB1209D45E5

TimeStamperCertificate :
Status                  : Valid
StatusMessage           : Signature verified.
Path                    : C:\Foo\Signed.ps1
SignatureType           : Authenticode
IsOSBinary              : False
```

Figure 2.46: Checking the script's digital signature

In *step 13*, you rerun the script file. Doing so generates a warning message, as you can see here:

```
PS C:\Foo> # 13. Running the signed script
PS C:\Foo> C:\Foo>:\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 2.47: Rerunning the script

To resolve the warning message, in *step 14*, you copy the certificate to the current user's trusted publisher store. Then in *step 15*, you run the signed script successfully, as you can see in this output:

```
PS C:\Foo> # 15. Running the signed script
PS C:\Foo> C:\Foo\Signed.ps1
Hello World from PowerShell 7!
Running on [SRV1]
```

Figure 2.48: Rerunning the script

In the final step in this recipe, *step 16*, you reset the process level execution policy to `Unrestricted`.

There's more...

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

In *step 4*, you sign the script using the newly created code-signing certificate. The output from this step shows that you signed the script with a certificate that was not issued by a trusted certificate authority. In *step 5*, you can see that the script file exists, but is now 2,136 bytes! To see why this relatively large file size, in *step 6*, you can see the script, with a large text block at the end holding the digital signature.

With *step 7*, you verify the signature on the script, which results in the same status message. The file is signed, but Windows does not (yet) trust the certificate authority that issued the code-signing certificate.

In *step 8*, you run the script. Despite the signature issue, the script runs fine, since you set the PowerShell execution policy to `Unrestricted` when you installed PowerShell. In *step 9*, you set the execution policy to `Allsigned`, and in *step 10*, you run the script. As you can see, PowerShell stops you from running the script due to the certificate chain issue.

In *step 11,* you copy the self-signed certificate into the current user's trusted root certificate store. You can only perform this copy action by using .NET objects as there are no cmdlets. When you invoke the `Add()` method, Windows displays a security warning. Note that Windows tends to display this dialog box *under* your PowerShell console or VS code. When you run this step, if the code seems to hang, check to see if the dialog box is hidden before agreeing to the addition to complete the step.

By copying the self-signed certificate to the trusted root store, Windows believes the code signing certificate is signed by a trusted CA. In *step 12*, you verify that the signature is trusted. Despite this, in *step 13*, you were sent a warning when you attempt to run the signed script. To avoid this warning, you copy the certificate to the current user's trusted publisher certificate store. With this done, in *step 15*, you rerun the script to observe that it now runs successfully.

In a production environment, you would more likely use either a public CA or an internal CA. If you use a public CA, then Windows *should* have that CA's certificate in the local host's trusted root store. You may need to add that certificate to the Trusted Publisher's store on each local host that you want to be able to run signed scripts. If you use an internal CA, you must ensure that each host is automatically enrolled for these certificates. For details on configuring certificate auto-enrollment, see: `https://docs.microsoft.com/windows-server/networking/core-network-guide/cncg/server-certs/configure-server-certificate-autoenrollment`.

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 that 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 could 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 make use of 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` after you have installed PowerShell 7.

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 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 ile 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 cmdlet to discover any modules with the string "Shortcut" in the module name. The output looks like this:

```
PS C:\Foo> # 1. Finding the PSShortCut module
PS C:\Foo> Find-Module -Name '*Shortcut'

Version   Name                                Repository   Description
-------   ----`                               ----------   -----------
2.2.0     DSCR_Shortcut                       PSGallery    PowerShell DSC Resource to create shortcut file.
0.2.1     PSAdvancedShortcut                  PSGallery    Advanced shortcut appliance to create and modify shortcut file properties that are not easily…
2.2.0     Remove-iCloudPhotosShortcut         PSGallery    A PowerShell module for removing iCloud Photos shortcuts from This PC and Quick access on Win…
1.0.6     PSShortcut                          PSGallery    This module eases working with Windows shortcuts (LNK and URL) files.
0.1.0     oneShortcut                         PSGallery    PowerShell module to query, create, and remove OneDrive shortcuts to SharePoint document libr…
```

Figure 2.49: Rerunning the script

In *step 2*, you install the `PSShortCut` module, producing no output. After you have the module installed, in *step 3*, you examine the module details, with output 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 2.50: Viewing module details

In *step 4*, you use `Get-Command` to confirm the names of the commands in the module, with output 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 2.51: Using Get-Command to find the commands in the PSShortCut module

As shown in *step 5*, you can search the entire host to discover all the shortcuts on the server, with results 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: [261]
```

Figure 2.52: Finding shortcuts on SRV1

In *step 6*, you discover the shortcut to PowerShell 7 (which you created in *Chapter 1*). The output is 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---          09/04/2022    16:11       1030 pwsh.lnk
```

Figure 2.53: Discovering shortcuts to PowerShell 7

In *step 7*, you use the `Get-Shortcut` command to find all the URL shortcuts, with output like this:

```
PS C:\Foo> # 7. Discovering URL shortcut
PS C:\Foo> $URLSC = Get-Shortcut -FilePath *.url
PS C:\Foo> $URLSC

    Directory: C:\Users\Administrator\Favorites

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---          31/03/2022    17:30        208 Bing.url
```

Figure 2.54: Discovering URL shortcuts

In *step 8*, you view the contents of the shortcut, with output 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 2.55: Viewing the contents of a URL shortcut

In *step 9*, you create a new URL shortcut, which generates no console output. In *step 10*, you use the URL shortcut. This step produces no console output, but you do see the Edge browser popping up, like this:

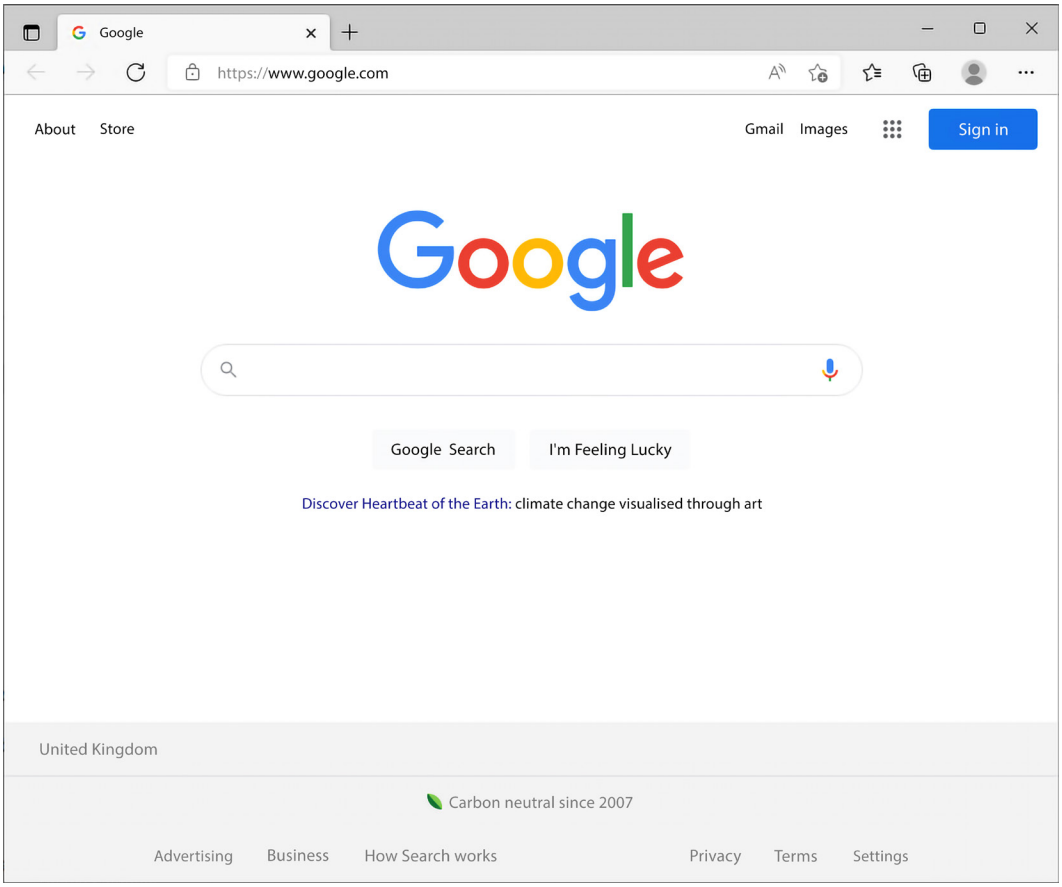

Figure 2.57: Using a shortcut to Google

In *step 11*, you create a new file-based URL (pointing to `notepad.exe`, which generates no output. Then in *step 12*, you use the shortcut. This step produces no output, but you should see Notepad pop up, like this:

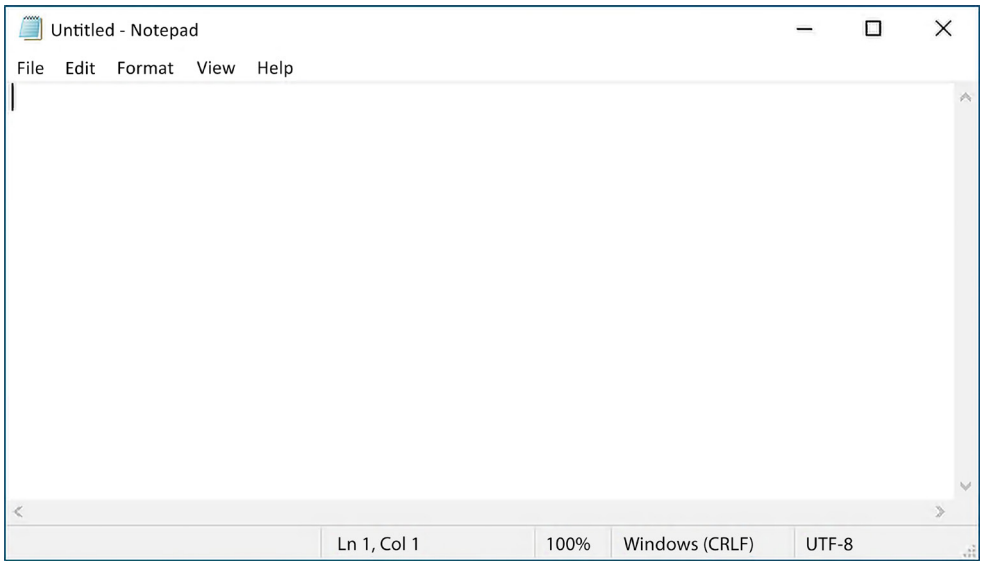

Figure 2.57: Using the shortcut to Notepad

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. For an unofficial guide to the format of these files, see `http://www.lyberty.com/encyc/articles/tech/dot_url_format_-_an_unofficial_guide.html`.

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 quickly became a near-standard for data transfer. A later Windows version, WinZip, became popular. 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 find more information about the ZIP file format at `https://en.wikipedia.org/wiki/Zip_(file_format)`.

Numerous developers, 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 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 will 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 old and has not been updated in many years.

Getting ready

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

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 files to archive

```
$Files = Get-ChildItem -Path 'C:\Foo\Archive'
$Count = $Files.Count
$LenKB = (($Files | Measure-Object -Property length -Sum).Sum)/1mb
"[{0}] files, occupying {1:n2}mb" -f $Count, $LenKB
```

6. Compressing a set of files into an archive

```
$AFILE1 = 'C:\Foo\Archive1.zip'
Compress-Archive -Path $Files -DestinationPath "$AFile1"
```

7. Compressing a folder containing files

```
$AFILE2 = 'C:\Foo\Archive2.zip'
Compress-Archive -Path "C:\Foo\Archive" -DestinationPath $AFile2
```

8. Viewing the archive files

```
Get-ChildItem -Path $AFILE1, $AFILE2
```

9. Viewing archive content with Windows Explorer

```
explorer.exe $AFILE1
```

10. Viewing the second archive with Windows Explorer

```
explorer.exe $AFILE2
```

11. Making a new output folder

```
$Opath = 'C:\Foo\Decompressed'
$NIHT2 = @{
  Path        = $Opath
```

```
  ItemType    = 'Directory'
  ErrorAction = 'SilentlyContinue'
}
New-Item @NIHT2 | Out-Null
```

12. Decompress the `Archive1.zip` archive

```
Expand-Archive -Path $AFILE1 -DestinationPath $Opath
```

13. Measuring the size of the decompressed files

```
$Files = Get-ChildItem -Path $Opath
$Count = $Files.Count
$LenKB = (($Files |
           Measure-Object -Property length -Sum).Sum)/1mb
"[{0}] decompressed files, occupying {1:n2}mb" -f $Count, $LenKB
```

How it works...

In *step 1*, you get details of the `Microsoft.Powershell.Archive` module, with output 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 2.58: Using the shortcut to Notepad

In *step 2*, you use `Get-Command` to discover additional details about the commands contained in the Archive module, with output 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 2.59: Viewing the commands in the Microsoft.PowerShell.Archive module

To prepare to use the archiving commands, in *step 3*, you create a hash table to hold the parameters to `New-Item` to create a new folder. In *step 4*, you populate the folder with a set of files. These two steps produce no output.

In *step 5*, you get a count of the files in the folder and the space they occupy, with output like this:

```
PS C:\Foo> # 5. Measuring 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 2.60: Measuring the files to be compressed

In *step 6*, you compress a set of files into an archive file, and in *step 7*, you compress a folder (containing files) into another archive file. These two steps produce no output. In *step 8*, you view the two archive files, with output 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---          28/04/2022     10:52          50106 Archive1.zip
-a---          28/04/2022     10:53          51706 Archive2.zip
```

Figure 2.61: Viewing the archive files

In *step 9*, you view the first archive file in Windows File Explorer, with output like this:

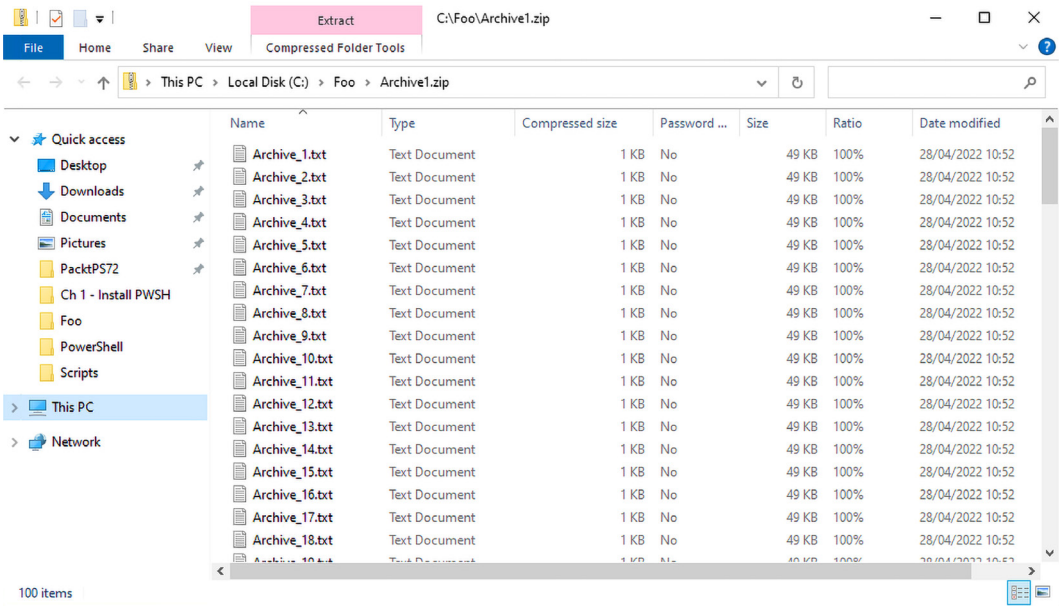

Figure 2.62: Viewing Archive1.zip

In *step 10*, you view the second archive file, which looks like this:

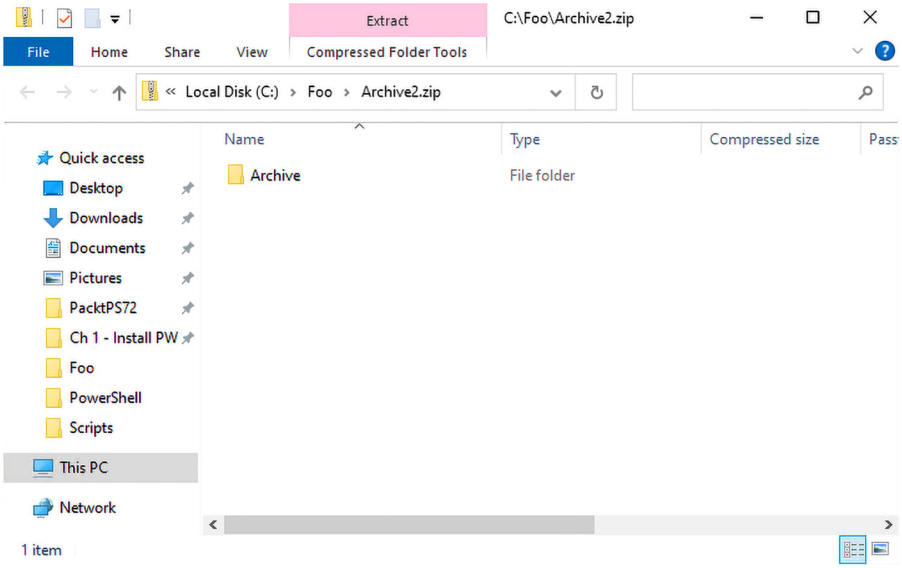

Figure 2.63: Viewing Archive1.zip

To demonstrate expanding an archive, in *step 11*, you create a new folder (`C:\Foo\Decompressed`), which generates no console output, In *step 12*, you use `Expand-Archive` to decompress the files within the archive to the specified destination folder, which also generates no output.

In the final step, *step 13*, you count the decompressed files with output 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 2.64: Viewing Archive1.zip

There's more...

In *step 1* and *step 2*, you can see the Archive module has two commands internally. As you can see, the module details returned by `Get-Module` show the module to have two commands. In the next step, `Get-Command` confirms the module has the two commands but that these are PowerShell functions and not PowerShell cmdlets. If you navigate to the folder containing this module, you can see both the module manifest (`Microsoft.PowerShell.Archive.psd1`) and the module file that defines the exported functions (`Microsoft.PowerShell.Archive.psm1`).

In *step 9*, you created the archive file `Archive1.zip`, which contains 100 files. In *step 12*, you used `Expand-Archive` to expand the files from the archive into the specified folder. As you can see in *step 13*, the files previously compressed into an archive are now decompressed and in the destination path.

Searching for Files Using the Everything Search Tool

Void Tools is a software company that produced the Everything search tool. Most IT pros run it from the GUI but there is a community module to help you use this tool using PowerShell. The Everything tool and the PowerShell community have developed a nice little product that you can easily leverage in PowerShell. The Everything applications return fully qualified filenames for those files on your system whose names meet a specified pattern. You can use either a wild card (e.g., `*.PS1`) or a regular expression (e.g., `'\.PS1$'`) as a pattern. There is also an add-on module that provides an easy-to-use PowerShell COMMAND.

The key benefit is that it is very fast. On a large Windows host with say one million total files, it might take 10-12 seconds to return a list of all the files on that system. To find all the `.PS1` files on the system drive would take under 100 milliseconds. By comparison, using `Get-ChildItem` would take several minutes.

If you are managing a large file server, tools like Everything could be very useful. And for a bit of light relief, you can bring up the GUI, specify ALL files, and sort files by date modified. After a second or two, you can watch as Windows and Windows applications create and update files on your system. Do you know what all those files are?

This recipe first downloads and installs the Everything tool and the `PSEverything` module from the PowerShell Gallery. Then you use the tool to find files on the system.

For more details on the Everything tool, see `https://www.voidtools.com/`. For the latest downloads, you can view the download page at `https://www.voidtools.com/downloads/`. And for more information about installing Everything, see a great support article at `https://www.voidtools.com/forum/viewtopic.php?f=5&t=5673&p=15546`. The tool provides a large range of installation options that you can use, depending on your needs.

Getting ready

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

How to do it...

1. Getting download locations

```
$ELoc  = 'https://www.voidtools.com/downloads'
$Release = Invoke-WebRequest -Uri $ELoc # Get all
$FLoc  = 'https://www.voidtools.com'
$EPath = $FLOC + ($Release.Links.href |
           Where-Object { $_ -match 'x64' } |
             Select-Object -First 1)
$EFile = 'C:\Foo\EverythingSetup.exe'
```

2. Downloading the Everything installer

```
Invoke-WebRequest -Uri $EPath -OutFile $EFile -verbose
```

3. Install Everything

```
$Iopt = "-install-desktop-shortcut -install-service"
$Iloc = 'C:\Program Files\Everything'
.\EverythingSetup.exe /S -install-options  $Iipt /D=$Iopt
```

4. Open the GUI for the first time

```
& "C:\Program Files\Everything\Everything.exe"
```

5. Finding the PSEverything module

```
Find-Module -Name PSEverything
```

6. Installing the `PSEverything` module

```
Install-Module -Name PSEverything -Force
```

7. Discovering commands in the module

```
Get-Command -Module PSEverything
```

8. Getting a count of files in folders below `C:\Foo`

```
Search-Everything |
  Get-Item |
    Group-Object DirectoryName |
      Where-Object name -ne '' |
        Format-Table -property Name, Count
```

9. Finding PowerShell scripts using wild cards

```
Search-Everything *.ps1 |
  Measure-Object
```

10. Finding all PowerShell scripts using regular expression

```
Search-Everything -RegularExpression '\.ps1$' -Global |
  Measure-Object
```

How it works...

In *step 1*, you set the locations for the download of the Everything tool. Then in *step 2*, you download the installation package. With *step 3*, you perform a silent install of the tool onto SRV1. With *step 4*, you open the Everything GUI. This produces no console output but you do see the following popup:

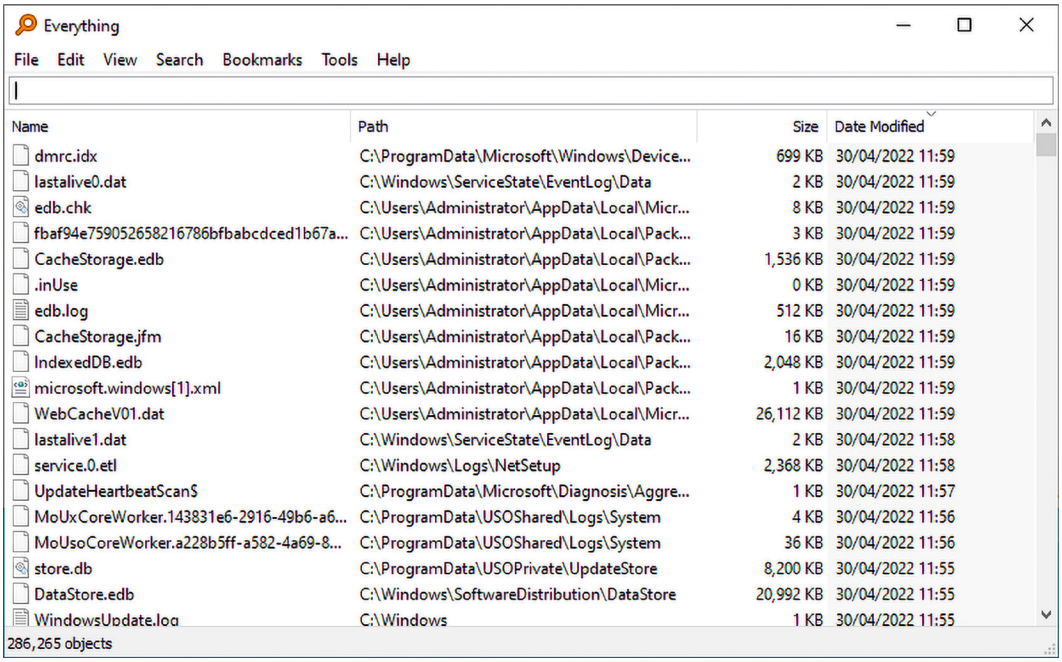

Figure 2.65: Everything GUI

With *step 5*, you find the PSEverything module with output like this:

```
PS C:\Foo> # 5. Find the PSEverything module
PS C:\Foo> Find-Module -Name PSEverything

Version   Name          Repository   Description
-------   ----          ----------   -----------
3.3.0     PSEverything  PSGallery    Powershell access to Everything - Blazingly fast file system se…
```

Figure 2.66: Finding the PSEverything module

In *step 6*, you install the module, which generates no console output. In *step 7* you view the commands in the module with output like this:

```
PS C:\Foo> # 7. Find commands in the module
PS C:\Foo> Get-Command -Module PSEverything

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Cmdlet          Search-Everything                                  3.3.0      PSEverything
Cmdlet          Select-EverythingString                            3.3.0      PSEverything
```

Figure 2.67: Getting the commands in the PSEverything module

```
PS C:\Foo> # 8. Getting a count of files in folders below C:\Foo
PS C:\Foo> Search-Everything |
             Get-Item  |
               Group-Object DirectoryName |
                 Where-Object name -ne '' |
                   Format-Table -Property Name, Count

Name                                      Count
----                                      -----
C:\Foo                                       17
C:\Foo\Archive                              100
C:\Foo\CascadiaCode\otf\static               48
C:\Foo\CascadiaCode\ttf                       8
C:\Foo\CascadiaCode\ttf\static               48
C:\Foo\CascadiaCode\woff2                     8
C:\Foo\CascadiaCode\woff2\static             48
C:\Foo\Decompressed                         100
C:\Foo\DownloadedModules\PSLogging\2.5.2      3
```

Figure 2.68: Counting the files in each folder below C:\Foo on SRV1

In *step 8*, you search for all the `*.PS1` files at and below `C:\Foo`, then return a count, with output like this:

```
PS C:\Foo> # 9. Finding PowerShell scripts using wild cards
PS C:\Foo> Search-Everything *.ps1 |
             Measure-Object

Count             : 4
Average           :
Sum               :
Maximum           :
Minimum           :
StandardDeviation :
Property          :
```

*Figure 2.69: Counting the *.PS1 files in and below C:\Foo*

In the final step in this recipe, *step 10*, you count the total number of files on `SRV1` whose filename matches the regular expression "`\.ps1$`" – which is the equivalent of the wild card used in the prior step. The output from this step looks like this:

```
PS C:\Foo> # 10. Finding all PowerShell scripts using regular expression
PS C:\Foo> Search-Everything -RegularExpression '\.ps1$' -Global |
             Measure-Object

Count             : 1629
Average           :
Sum               :
Maximum           :
Minimum           :
StandardDeviation :
Property          :
```

*Figure 2.70: Counting the *.PS1 files on SRV1*

There's more...

In *step 1*, you set variables that correspond to the location from which you can download the Everything installation package. This location was correct at the time of writing – but may have changed.

The screenshot for *step 4* shows the Everything GUI with the files sorted with the most recently changed files at the top of the window. Note the number of files that Windows changes every minute! Do you know what these files are or why Windows is changing them? Answers on a postcard, please.

In *step 8*, you view the folders below `C:\Foo` and the files contained in each folder. You created these folders and files by performing the recipes in this book. If you skipped any of the prior recipes, loaded different modules, etc., you may see different output.

The `Search-Everything` cmdlet returns the filenames that match a specified pattern. The cmdlet does not search inside any of the files. As you see in *step 7*, there is another command in the module, `Select-EverythingString`, which does search inside the files found.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

`https://packt.link/SecNet`