3

Exploring .NET

This chapter covers the following recipes:

- Exploring .NET Assemblies
- Exploring .NET Classes
- Leveraging .NET Methods
- Creating a C# Extension
- Creating a cmdlet

Introduction

Microsoft first launched .NET Framework in June 2000, with the code name Next Generation Windows Services. Amidst a barrage of marketing zeal, Microsoft seemed to add the .NET moniker to every product in its portfolio: Windows .NET Server (later renamed Windows Server 2003), Visual Studio.NET, and even MapPoint .NET.

.NET Framework provided application developers with a host of underlying features and technologies on which to base their applications. These worked well then (20+ years ago), but newer features later emerged based on advances in the underlying technologies. For example, **Simple Object Access Protocol (SOAP)** and XML-based web services have given way to **Representational State Transfer (REST)** and **JavaScript Object Notation (JSON)**.

Microsoft made considerable improvements to .NET Framework with each release and added new features based on customer feedback. .NET Framework started as closed-source, but Microsoft transitioned .NET to open source, aka .NET Core. PowerShell 7.2 is based on .NET 6.0.

An issue that emerged over time was that .NET became fragmented across different OSs and the web. To resolve this problem, Microsoft created .NET 5.0 and dropped "core" moniker. They released this version both on the web and across hardware platforms such as ARM. The intention from now on is to move to a single .NET across all platforms and form factors, thus simplifying application development. See `https://www.zdnet.com/article/microsoft-takes-a-major-step-toward-net-unification-with-net-5-0-release/` for some more detail on this.

It is important to note that both .NET 6.0 and PowerShell 7.2 have **long-term support (LTS)**. Microsoft intends to support these two products for three years, as described in this article: `https://devblogs.microsoft.com/dotnet/announcing-net-6/`. Many larger organizations prefer products with a longer support horizon.

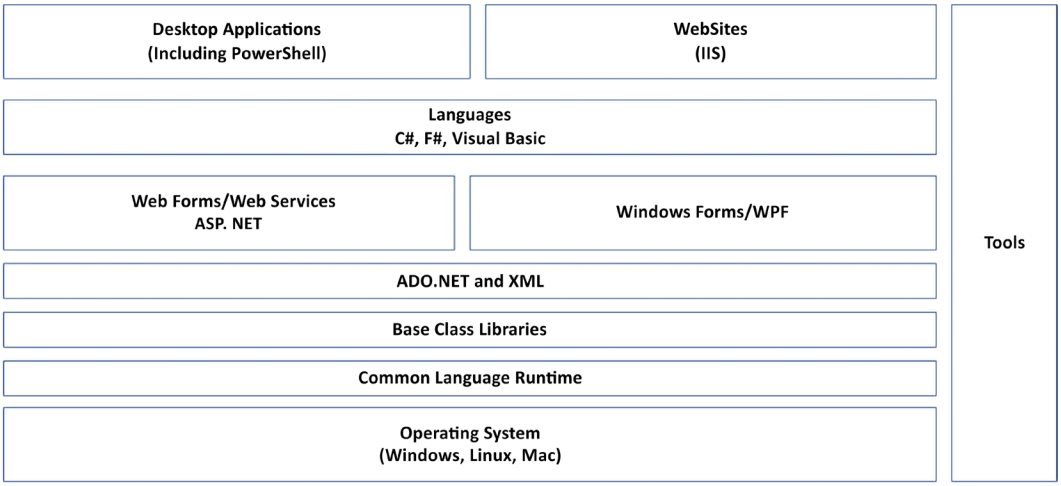
For the application developer, .NET provides a rich **Application Program Interface (API)**, a programming model, plus associated tools and runtime. .NET is an environment in which developers can develop rich applications and websites across multiple platforms.

PowerShell sits on top of and makes heavy use of .NET. Some cmdlets are just a thin wrapper around .NET objects. For the IT professional, .NET provides the underpinnings of PowerShell but is otherwise mostly transparent. You use cmdlets to carry out actions. While these cmdlets use .NET to carry out their work, you are generally unaware of how PowerShell works under the hood. For example, you can use `Get-Process` without worrying about how that cmdlet obtains the details of the processes.

In some cases, there is no cmdlet available to you to carry out some action, but you can use .NET classes and methods to achieve it. For example, if you are creating an Active Directory cross-forest trust. So in some cases, knowing how to use .NET classes and their associated methods can be useful.

In some cases, you may wish to extend what is available by creating a .NET class (e.g., using C#) or developing a full PowerShell cmdlet. It is quite easy to download the tools and build a cmdlet. You might have a calculation of a bit of C# code that performs a particular task. Rather than taking the time to convert it to PowerShell and possibly taking a performance hit, you can wrap the code easily and create a class. Then you can add this to the PowerShell environment and use the class directly from PowerShell. And, if you are developing line-of-business applications, you could also create a full C# cmdlet.

Here is a high-level illustration of the components of .NET.



.Net Architecture

Figure 3.1: .NET components

The key components of .NET in this diagram are:

- **Operating system** – The .NET story begins with the operating system. The operating system provides the fundamental operations of your computer. .NET leverages the OS components. The .NET team supports .NET 6.0, the basis for PowerShell 7.2, on Windows, Linux, and the Apple Mac. .NET also runs on the ARM platform, enabling you to get PowerShell on ARM. See `https://github.com/dotnet/core/blob/main/release-notes/6.0/supported-os.md` for a full list of the operating systems supported.
- **Common Language Runtime (CLR)** – The CLR is the core of .NET, the managed code environment in which all .NET applications run (including PowerShell). The CLR delivers a managed execution environment and type safety and code access security. The CLR manages memory, threads, objects, resource allocation/de-allocation, and garbage collection. For an overview of the CLR, see `https://docs.microsoft.com/dotnet/standard/clr/`. The CLR also contains the **Just-In-Time (JIT)** compiler and the class loader (responsible for loading classes into an application at runtime). For the PowerShell user, the CLR "just works."

- **Base Class Libraries (BCLs)** – .NET comes with numerous base class libraries, the fundamental built-in features you can use to build .NET applications. Each BCL is a DLL that contains .NET classes and types. The installer adds the .NET Framework components into PowerShell's installation folder when you install PowerShell. PowerShell developers use these libraries to implement PowerShell cmdlets. You can also call classes in the BCL directly from within PowerShell. It is the BCLs that enable you to reach into .NET.
- **WMI, COM, Win32** – Traditional application development technologies you can access via PowerShell. Within PowerShell, you can run programs that use these technologies and access them from the console or via a script. Your scripts can be a mixture of cmdlets, .NET, WMI, COM, and Win32 if necessary.
- **Languages** – This is the language that a cmdlet and application developer uses to develop PowerShell cmdlets. You can use various languages based on your preferences and skill set. Most cmdlet developers use C#, although using any .NET supported language would work, such as VB.NET. The original PowerShell development team designed the PowerShell language based on C#. You can describe PowerShell as on the glide scope down to C#. Understanding C# can be useful since there is a lot of documentation with examples in C#.
- **PowerShell** – PowerShell sits on top of these other components. From PowerShell, you can use cmdlets developed using a supported language. And you can use both BCL and WMI/COM/Win32.

Before diving into using .NET, there are some terms you should understand:

- **Class** – A class is a definition of some kind of object: the `System.String` defines a string while the `System.Diagnostics.Process` class defines a Windows process. Developers typically use C#, but you have options.
- **Assembly** – An assembly is a collection of types and resources used by an application and contains rich metadata about what the assembly includes. Most assemblies you use are .DLL files that a developer compiled for you previously – you add them using the `Add-Type` command. A neat feature is that you can also use `Add-Type` and supply just the source code – the cmdlet does the compilation for you at runtime!
- **Class instances** – You can have multiple instances of a given class, such as the processes on a system. A class instance can have properties, methods, events, and more. A property is some attribute of a class instance. For example, the `StartTime` property on a process object holds the process's start time. A method is some action that .NET can take on a given instance. A process instance, for example, has a `Kill()` method that immediately terminates the process (and does NOT ask you if you are sure!).

- **Static properties and methods** – Many classes have static properties and methods. Static properties relate to the class as opposed to a class instance. The `System.Int32` class has a static property, `MaxValue`, that holds a value of the largest value that you can carry in a 32-bit integer instance. The `System.IO.FileInfo` class has a static method `new()` that enables you to create a new file.

In the chapter, you examine .NET assemblies, classes, and methods.

The system used in the chapter

This chapter is about using PowerShell 7 to explore .NET. In this chapter, you use a single host, `SRV1`, as follows:

Figure 3.2: 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.

Exploring .NET Assemblies

With .NET, an assembly holds compiled code that .NET can run. An assembly can be either a **Dynamic Link Library** (**DLL**) or an executable. As you can see in this recipe, cmdlets and .NET classes are generally contained in DLLs. Each assembly also includes a manifest that describes what is in the assembly and compiled code.

Most PowerShell modules and most PowerShell commands use assemblies of compiled code. When PowerShell loads any module, the module manifest (the `.PSD1` file) lists the assemblies that make up the module. For example, the `Microsoft.PowerShell.Management` module provides many core PowerShell commands such as `Get-ChildItem` and `Get-Process`. This module's manifest lists a nested module (i.e., `Microsoft.PowerShell.Commands.Management.dll`) as the assembly containing the actual commands. PowerShell automatically loads this DLL file for you whenever it loads the module.

A great feature of PowerShell is the ability for you to invoke a .NET class method directly or to obtain a static .NET class value (or even execute a static method). The syntax for calling a .NET method or a .NET field, demonstrated in numerous recipes in this book, is to enclose the class name in square brackets and then follow it with two ":" characters (that is: "::") followed by the name of the method or static field (such as `[System.Int32]::MaxValue`).

In this recipe, you examine the assemblies loaded into PowerShell 7 and compare them with the behavior in Windows PowerShell. You also look at a module and the assembly that implements the commands in the module. The recipe illustrates some of the differences between PowerShell 7 and Windows PowerShell and how they co-exist with .NET.

Getting ready

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

How to do it...

1. Counting loaded assemblies

```
$Assemblies = [System.AppDomain]::CurrentDomain.GetAssemblies()
"Assemblies loaded: {0:n0}" -f $Assemblies.Count
```

2. Viewing first 10

```
$Assemblies | Select-Object -First 10
```

3. Checking assemblies in Windows PowerShell

```
$B = {
  [System.AppDomain]::CurrentDomain.GetAssemblies()
}
$PS51 = New-PSSession -UseWindowsPowerShell
```

```
$Assin51 = Invoke-Command -Session $PS51 -ScriptBlock $SB
"Assemblies loaded in Windows PowerShell: {0:n0}" -f $Assin51.Count
```

4. Viewing `Microsoft.PowerShell` assemblies

```
$Assin51 |
  Where-Object FullName -Match "Microsoft\.Powershell" |
    Sort-Object -Property Location
```

5. Exploring the `Microsoft.PowerShell.Management` module

```
$AllTheModulesOnThisSystem =
  Get-Module -Name Microsoft.PowerShell.Management -ListAvailable
$AllTheModuleOnThisYstgewjm  | Format-List
```

6. Viewing module manifest

```
$Manifest = Get-Content -Path $Mod.Path
$Manifest | Select-Object -First 20
```

7. Discovering the module's assembly

```
Import-Module -Name Microsoft.PowerShell.Management
$Match = $Manifest | Select-String Modules
$Line = $Match.Line
$DLL = ($Line -Split '"')[1]
Get-Item -Path $PSHOME\$DLL
```

8. Viewing associated loaded assembly

```
$Assemblies2 = [System.AppDomain]::CurrentDomain.GetAssemblies()
$Assemblies2 | Where-Object Location -match $DLL
```

9. Getting details of a PowerShell command inside a module DLL

```
$Commands = $Assemblies |
               Where-Object Location -match Commands.Management\.dll
$Commands.GetTypes() |
  Where-Object Name -match "Addcontentcommand$"
```

How it works...

In *step 1*, you use the `GetAssemblies()` static method to return all the assemblies currently loaded by PowerShell. Then you output a count of the assemblies presently loaded, which looks like this:

```
PS C:\Foo> # 1. Counting loaded assemblies
PS C:\Foo> $Assemblies = [System.AppDomain]::CurrentDomain.GetAssemblies()
PS C:\Foo> "Assemblies loaded: {0:n0}" -f $Assemblies.Count
Assemblies loaded: 86
```

Figure 3.3: Counting loaded assemblies

In *step 2*, you look at the first ten loaded assemblies, with output like this:

```
PS C:\Foo> # 2. Viewing first 10
PS C:\Foo> $Assemblies | Select-Object -First 10

GAC    Version        Location
---    -------        --------
False  v4.0.30319     C:\Program Files\PowerShell\7\System.Private.CoreLib.dll
False  v4.0.30319     C:\Program Files\PowerShell\7\pwsh.dll
False  v4.0.30319     C:\Program Files\PowerShell\7\System.Runtime.dll
False  v4.0.30319     C:\Program Files\PowerShell\7\Microsoft.PowerShell.ConsoleHost.dll
False  v4.0.30319     C:\Program Files\PowerShell\7\System.Management.Automation.dll
False  v4.0.30319     C:\Program Files\PowerShell\7\System.Threading.Thread.dll
False  v4.0.30319     C:\Program Files\PowerShell\7\System.Runtime.InteropServices.dll
False  v4.0.30319     C:\Program Files\PowerShell\7\System.Threading.dll
False  v4.0.30319     C:\Program Files\PowerShell\7\System.Diagnostics.Process.dll
False  v4.0.30319     C:\Program Files\PowerShell\7\System.Text.RegularExpressions.dll
```

Figure 3.4: Viewing the first ten loaded assemblies

In *step 3*, you examine the assemblies loaded into Windows PowerShell 5.1, which looks like this:

```
PS C:\Foo> # 3. Checking assemblies in Windows PowerShell
PS C:\Foo> $ScriptBlock = {
             [System.AppDomain]::CurrentDomain.GetAssemblies()
           }
PS C:\Foo> $PS51 = New-PSSession -UseWindowsPowerShell
PS C:\Foo> $Assin51 = Invoke-Command -Session $PS51 -ScriptBlock $ScriptBlock
PS C:\Foo> "Assemblies loaded in Windows PowerShell: {0:n0}" -f $Assin51.Count

Assemblies loaded in Windows PowerShell: 16
```

Figure 3.5: Counting assemblies loaded in Windows PowerShell

In *step 4*, you examine the `Microsoft.PowerShell.*` assemblies in PowerShell 5.1, which looks like this:

```
PS C:\Foo> # 4. Viewing Microsoft.PowerShell assemblies
PS C:\Foo> $Assin51 |
              Where-Object FullName -Match "Microsoft\.Powershell" |
                Sort-Object -Property Location

GAC    Version        Location                                                PSComputerName
---    -------        --------                                                --------------
True   v4.0.30319     C:\WINDOWS\Microsoft.Net\assembly\GAC_MSIL\Microsoft.PowerS… localhost
True   v4.0.30319     C:\WINDOWS\Microsoft.Net\assembly\GAC_MSIL\Microsoft.PowerS… localhost
```

Figure 3.6: Examining the Microsoft.PowerShell. assemblies in Windows PowerShell*

The `Microsoft.PowerShell.Management` PowerShell module contains numerous core PowerShell commands. In *step 6*, you examine the details of this module, as returned by `Get-Module`, with output like this:

```
PS C:\Foo> # 5. Exploring the Microsoft.PowerShell.Management module
PS C:\Foo> $Mod =
             Get-Module -Name Microsoft.PowerShell.Management -ListAvailable
PS C:\Foo> $Mod  | Format-List

Name              : Microsoft.PowerShell.Management
Path              : C:\program files\powershell\7\Modules\Microsoft.PowerShell.Management\Microsoft.PowerShell.Management.psd1
Description       :
ModuleType        : Manifest
Version           : 7.0.0.0
PreRelease        :
NestedModules     : {Microsoft.PowerShell.Commands.Management}
ExportedFunctions :
ExportedCmdlets   : {Add-Content, Clear-Content, Get-Clipboard, Set-Clipboard, Clear-ItemProperty, Join-Path, Convert-Path,
                    Copy-ItemProperty, Get-ChildItem, Get-Content, Get-ItemProperty, Get-ItemPropertyValue, Move-ItemProperty, Get-Location,
                    Set-Location, Push-Location, Pop-Location, New-PSDrive, Remove-PSDrive, Get-PSDrive, Get-Item, New-Item, Set-Item,
                    Remove-Item, Move-Item, Rename-Item, Copy-Item, Clear-Item, Invoke-Item, Get-PSProvider, New-ItemProperty, Split-Path,
                    Test-Path, Test-Connection, Get-Process, Stop-Process, Wait-Process, Debug-Process, Start-Process, Remove-ItemProperty,
                    Rename-ItemProperty, Resolve-Path, Get-Service, Stop-Service, Start-Service, Suspend-Service, Resume-Service,
                    Restart-Service, Set-Service, New-Service, Remove-Service, Set-Content, Set-ItemProperty, Restart-Computer,
                    Stop-Computer, Rename-Computer, Get-ComputerInfo, Get-TimeZone, Set-TimeZone, Get-HotFix, Clear-RecycleBin}
ExportedVariables :
ExportedAliases   : {gcb, gin, gtz, scb, stz}
```

Figure 3.7: Examining the Microsoft.PowerShell.Management module

In *step 6*, you view the module manifest for the `Microsoft.PowerShell.Management` PowerShell module. The figure below shows the first twenty lines of the manifest, which look like this:

```
PS C:\Foo> # 6. Viewing module manifest
PS C:\Foo> $Manifest = Get-Content -Path $Mod.Path
PS C:\Foo> $Manifest | Select-Object -First 20
@{
GUID="EEFCB906-B326-4E99-9F54-8B4BB6EF3C6D"
Author="PowerShell"
CompanyName="Microsoft Corporation"
Copyright="Copyright (c) Microsoft Corporation."
ModuleVersion="7.0.0.0"
CompatiblePSEditions = @("Core")
PowerShellVersion="3.0"
NestedModules="Microsoft.PowerShell.Commands.Management.dll"
HelpInfoURI = 'https://aka.ms/powershell72-help'
FunctionsToExport = @()
AliasesToExport = @("gcb", "gin", "gtz", "scb", "stz")
CmdletsToExport=@("Add-Content",
    "Clear-Content",
    "Get-Clipboard",
    "Set-Clipboard",
    "Clear-ItemProperty",
    "Join-Path",
    "Convert-Path",
    "Copy-ItemProperty",
```

Figure 3.8: Examining the module manifest for the Microsoft.PowerShell.Management module

In *step 7*, you import the `Microsoft.PowerShell.Management` module, extract the name of the DLL implementing the module, and discover its supporting DLL, with output like this:

```
PS C:\Foo> # 7. Discovering the module's assembly
PS C:\Foo> Import-Module -Name Microsoft.PowerShell.Management
PS C:\Foo> $Match = $Manifest | Select-String Modules
PS C:\Foo> $Line = $Match.Line
PS C:\Foo> $DLL = ($Line -Split '"')[1]
PS C:\Foo> Get-Item -Path $PSHOME\$DLL

    Directory: C:\Program Files\PowerShell\7

Mode         LastWriteTime   Length Name
----         -------------   ------ ----
-a---  08/03/2022    23:22  1134480 Microsoft.PowerShell.Commands.Management.dll
```

Figure 3.9: Examining a module's supporting DLL

In *step 8*, you find the assembly that contains the cmdlets in the `Microsoft.PowerShell.Management` module, which looks like this:

```
PS C:\Foo> # 8. Viewing associated loaded assembly
PS C:\Foo> $Assemblies2 = [System.AppDomain]::CurrentDomain.GetAssemblies()
PS C:\Foo> $Assemblies2 | Where-Object Location -match $DLL

GAC   Version     Location
---   -------     --------
False v4.0.30319  C:\Program Files\PowerShell\7\Microsoft.PowerShell.Commands.Management.dll
```

Figure 3.10: Examining a module's supporting DLL

In *step 9*, you discover the name of the class that implements the `Add-Content` command, which looks like this:

```
PS C:\Foo> # 9. Getting details of a PowerShell command inside a module DLL
PS C:\Foo> $Commands = $Assemblies |
             Where-Object Location -match Commands.Management\.dll
PS C:\Foo> $Commands.GetTypes() |
             Where-Object Name -match "Addcontentcommand$"

IsPublic IsSerial Name              BaseType
-------- -------- ----              --------
True     False    AddContentCommand Microsoft.PowerShell.Commands.WriteContentCommandBase
```

Figure 3.11: Examining a module's supporting DLL

There's more...

In this recipe, you have seen the .NET assemblies used by PowerShell. These consist of both .NET Framework assemblies (i.e., the BCLs) and the assemblies that implement PowerShell commands. For example, you find the `Add-Content` cmdlet in the PowerShell module `Microsoft.PowerShell.Management`.

In *step 1*, you use the `GetAssemblies()` static method to return all the assemblies currently loaded by PowerShell 7.2. As you can see, the syntax is different from calling PowerShell cmdlets.

In *steps 3* and *step 4*, you obtain and view the assemblies loaded by Windows PowerShell 5.1. As you can see, Windows PowerShell and PowerShell 7.2 load different assemblies.

In *step 6*, you view the first 20 lines of the module manifest for the `Microsoft.PowerShell.Management` module. The output cuts off the complete list of cmdlets exported by the module and the long digital signature for the module manifest. In the output, you can see that the `Microsoft.PowerShell.Management.dll` contains the `Add-Content` command.

In *step 7*, you discover the DLL, which implements, among others, the `Add-Content` cmdlet, and in *step 8*, you can see that the assembly is loaded. In *step 9*, you can discover that the `Add-Content` PowerShell cmdlet is implemented by the `AddContentCommand` class within the Assembly's DLL. For the curious, navigate to `https://github.com/PowerShell/PowerShell/blob/master/src/Microsoft.PowerShell.Commands.Management/commands/management/AddContentCommand.cs`, where you can read the source code for this cmdlet.

Exploring .NET Classes

With .NET, a class defines an object in terms of properties, methods, etc. Objects and object occurrences are fundamental to PowerShell, where cmdlets produce and consume objects. The `Get-Process` command returns objects of the `System.Diagnostics.Process` class. When you use `Get-ChildItem` to return files and folders, the output is a set of objects based on the `System.IO.FileInfo` and `System.IO.DirectoryInfo` classes.

In most cases, your console activities and scripts use the objects created automatically by PowerShell commands. But you can also use the `New-Object` command to create occurrences of any class as necessary. This book shows numerous examples of creating an object using `New-Object`.

Within .NET, you have two kinds of object definitions: .NET classes and .NET types. A type defines a simple object that lives, at runtime, on your CPU's stack. Classes, being more complex, live in the global heap. The global heap is a large area of memory that .NET uses to hold object occurrences during a PowerShell session. In almost all cases, the difference between type and class is probably not overly relevant to IT professionals using PowerShell.

After a script or even a part of a script has run, .NET can tidy up the global heap in a process known as garbage collection. The garbage collection process is also not important for IT professionals (in most cases). The scripts you see in this book, for example, are not generally impacted by the garbage collection process, nor are most production scripts. For more information on the garbage collection process in .NET, see `https://docs.microsoft.com/dotnet/standard/garbage-collection/`.

There are cases where the GC process can impact performance. For example, the `System.Array` class creates objects of fixed length within the global heap. If you add an item to an array, .NET creates a new copy of the array (including the addition) and marks the old one for removal. If you add a few items to the array, the performance hit is negligible. But if you are adding millions of occurrences, the performance can suffer significantly. You can use the `ArrayList` class to avoid this, which supports adding/removing items from an array without the performance penalty.

For more information on GC and performance, see `https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/performance`.

In .NET, occurrences of every class or type can include members, including properties, methods, and events. A property is an attribute of an occurrence of a class. An occurrence of the `System.IO.FileInfo` object, for example, has a `FullName` property. A method is effectively a function you can call that can do something to an object occurrence. You look at .NET methods in "Leveraging .NET Methods recipe." With .NET, an event can happen to an object occurrence, such as when Windows generates an event when a Windows process terminates. This book does not cover .NET events; however, a later chapter in this book discusses using WMI events.

You can quickly determine an object's class (or type) by piping the output of any cmdlet, or an object, to the `Get-Member` cmdlet. The `Get-Member` cmdlet uses a feature of .NET, reflection, to look inside and give you a definitive statement of what that object contains. This feature is invaluable – instead of guessing where in some piece of string output your script can find the full name of a file, you can discover the `FullName` property, a string, or the `Length` property, which is unambiguously an integer. Reflection and the `Get-Member` cmdlet help you to discover the properties and other members of a given object.

As mentioned earlier, .NET classes can have static properties and static methods. Static properties/methods are aspects of the class of a whole as opposed to a specific class instance. A static property is a fixed constant value, such as the maximum and minimum values for a 32-bit signed integer or the value of Pi. A static method is independent of any specific instance. For example, the `Parse()` method of the `INT32` class can parse a string to ensure it is a value 32-bit signed integer. In most cases, you use static methods to create object instances or do something related to the class, such as parsing a string to determine if it is a valid 32-bit integer.

This recipe looks at some everyday objects created automatically by PowerShell. The recipe also examines the static fields of the [Int] .NET class.

Getting ready

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

How to do it...

1. Creating a `FileInfo` object

```
$File = Get-ChildItem -Path $PSHOME\pwsh.exe
$File
```

2. Discovering the underlying class

```
$Type = $File.GetType().FullName
".NET Class name: $Type"
```

3. Getting member types of `Fileinfo` object

```
$File |
  Get-Member |
    Group-Object -Property MemberType |
      Sort-Object -Property Count -Descending
```

4. Discovering properties of a Windows service

```
Get-Service |
  Get-Member -MemberType Property
```

5. Discovering the underlying type of an integer

```
$I = 42
$IntType  = $I.GetType()
$TypeName = $IntType.FullName
$BaseType = $IntType.BaseType.Name
".NET Class name      : $TypeName"
".NET Class base type : $BaseType"
```

6. Looking at Process objects

```
$Pwsh = Get-Process -Name pwsh |
  Select-Object -First 1
$Pwsh |
  Get-Member |
    Group-Object -Property MemberType |
      Sort-Object -Property Count -Descending
```

7. Looking at static properties within a class

```
$Max = [Int32]::MaxValue
$Min = [Int32]::MinValue
"Minimum value [$Min]"
"Maximum value [$Max]"
```

How it works...

In *step 1*, you use the `Get-ChildItem` cmdlet to return a `Fileinfo` object. When you examine the object created by the cmdlet, it looks like this:

```
PS C:\Foo> # 1. Creating a Fileinfo object
PS C:\Foo> $FILE = Get-ChildItem -Path $PSHOME\pwsh.exe
PS C:\Foo> $FILE

    Directory: C:\Program Files\PowerShell\7

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a---          08/03/2022    23:21         287632 pwsh.exe
```

Figure 3.12: Creating a FileInfo object

In *step 2*, you use the `Get-Type()` method to return the full class name of the `$File` object, which looks like this:

```
PS C:\Foo> # 2. Discovering the underlying class
PS C:\Foo> $Type = $FILE.GetType().FullName
PS C:\Foo> ".NET Class name: $Type"
.NET Class name: System.IO.FileInfo
```

Figure 3.13: Viewing the underlying class name

In *step 3*, you get the different types of members in an instance of a `FileInfo` class. You can see the other member types in the output like this:

```
PS C:\Foo> # 3. Getting member types of Fileinfo object
PS C:\Foo> $File |
             Get-Member |
               Group-Object -Property MemberType |
                 Sort-Object -Property Count -Descending

Count Name            Group
----- ----            -----
   23 Method          {System.IO.StreamWriter AppendText(), System.IO.FileInfo CopyTo(string destFileName), System.IO.FileInfo Copy…
   16 Property        {System.IO.FileAttributes Attributes {get;set;}, datetime CreationTime {get;set;}, datetime CreationTimeUtc {…
    6 NoteProperty    {string PSChildName=pwsh.exe, PSDriveInfo PSDrive=C, bool PSIsContainer=False, string PSParentPath=Microsoft.…
    3 CodeProperty    {System.String LinkType{get=GetLinkType;}, System.String Mode{get=Mode;}, System.String ModeWithoutHardLink{g…
    2 ScriptProperty  {System.Object BaseName {get=if ($this.Extension.Length -gt 0){$this.Name.Remove($this.Name.Length - $this.Ex…
    1 AliasProperty   {Target = LinkTarget}
```

Figure 3.14: Viewing the member types of a FileInfo object

In *step 4*, you examine the properties of an object representing a Windows service, looking like this:

```
PS C:\Foo> # 4. Discovering properties of a Windows service
PS C:\Foo> Get-Service |
             Get-Member -MemberType Property

   TypeName: System.Service.ServiceController#StartupType

Name                  MemberType Definition
----                  ---------- ----------
BinaryPathName        Property   System.String {get;set;}
CanPauseAndContinue   Property   bool CanPauseAndContinue {get;}
CanShutdown           Property   bool CanShutdown {get;}
CanStop               Property   bool CanStop {get;}
Container             Property   System.ComponentModel.IContainer Container {get;}
DelayedAutoStart      Property   System.Boolean {get;set;}
DependentServices     Property   System.ServiceProcess.ServiceController[] DependentServices {get;}
Description           Property   System.String {get;set;}
DisplayName           Property   string DisplayName {get;set;}
MachineName           Property   string MachineName {get;set;}
ServiceHandle         Property   System.Runtime.InteropServices.SafeHandle ServiceHandle {get;}
ServiceName           Property   string ServiceName {get;set;}
ServicesDependedOn    Property   System.ServiceProcess.ServiceController[] ServicesDependedOn {get;}
ServiceType           Property   System.ServiceProcess.ServiceType ServiceType {get;}
Site                  Property   System.ComponentModel.ISite Site {get;set;}
StartType             Property   System.ServiceProcess.ServiceStartMode StartType {get;}
StartupType           Property   Microsoft.PowerShell.Commands.ServiceStartupType {get;set;}
Status                Property   System.ServiceProcess.ServiceControllerStatus Status {get;}
UserName              Property   System.String {get;set;}
```

Figure 3.15: Viewing the properties of a Windows service object

In *step 5*, you examine the underlying type of a 32-bit integer, with output like this:

```
PS C:\Foo> # 5. Discovering the underlying type of an integer
PS C:\Foo> $I = 42
PS C:\Foo> $IntType  = $I.GetType()
PS C:\Foo> $TypeName = $IntType.FullName
PS C:\Foo> $BaseType = $IntType.BaseType.Name
PS C:\Foo> ".NET Class name      : $TypeName"
PS C:\Foo> ".NET Class base type : $BaseType"

.NET Class name      : System.Int32
.NET Class base type : ValueType
```

Figure 3.16: Viewing the class name for a 32-bit integer

In *step 6*, you look at the different member types within an object representing a Windows process, with the following output:

```
PS C:\Foo> # 6. Looking at Process objects
PS C:\Foo> $PWSH = Get-Process -Name pwsh |
             Select-Object -First 1
PS C:\Foo> $PWSH |
             Get-Member |
               Group-Object -Property MemberType |
                 Sort-Object -Property Count -Descending

Count Name             Group
----- ----             -----
   52 Property         {int BasePriority {get;}, System.ComponentModel.IContainer Container {get;}, bool EnableRaisingEvents {get;se…
   19 Method           {void BeginErrorReadLine(), void BeginOutputReadLine(), void CancelErrorRead(), void CancelOutputRead(), void…
    8 ScriptProperty   {System.Object CommandLine {get=…
    7 AliasProperty    {Handles = Handlecount, Name = ProcessName, NPM = NonpagedSystemMemorySize64, PM = PagedMemorySize64, SI = Se…
    4 Event            {System.EventHandler Disposed(System.Object, System.EventArgs), System.Diagnostics.DataReceivedEventHandler E…
    2 PropertySet      {PSConfiguration {Name, Id, PriorityClass, FileVersion}, PSResources {Name, Id, Handlecount, WorkingSet, NonP…
    1 CodeProperty     {System.Object Parent{get=GetParentProcess;}}
    1 NoteProperty     {string __NounName=Process}
```

Figure 3.17: Viewing the member types of a process object

In the final step in this recipe, *step 7*, you examine two static properties of an `Int32` object, with output like this:

```
PS C:\Foo> # 7. Looking at static properties within a class
PS C:\Foo> $Max = [Int32]::MaxValue
PS C:\Foo> $Min = [Int32]::MinValue
PS C:\Foo> "Minimum value [$Min]"
PS C:\Foo> "Maximum value [$Max]"
Minimum value [-2147483648]
Maximum value [2147483647]
```

Figure 3.18: Viewing the static properties of a 32-bit integer

There's more...

In *step 1*, you create a `FileInfo` object representing `pwsh.exe`. This object's full type name is `System.IO.FileInfo`. In .NET, classes live in namespaces and, in general, namespaces equate to DLLs. In this case, the object is in the `System.IO` namespace, and the class is contained in the `System.IO.FileSystem.DLL`.

As you use your favorite search engine to learn more about .NET classes that might be useful, note that many sites describe the class without the namespace, simply as the `FileInfo` class, while others spell out the class as `System.IO.FileInfo`. If you are using .NET class names in your scripts, a best practice is to spell out the full class name. You can discover namespace and DLL details by examining the class's documentation – in this case, `https://docs.microsoft.com/dotnet/api/system.io.fileinfo`.

In *step 6*, you examine the different member types you can find on a class or class instance for a process object. Some of the members you can see come directly from the underlying .NET object. Others, such as ScriptProperty, AliasProperty, and CodeProperty, are added by PowerShell's **Extensible Type System (ETS)**. With the ETS, the developers extend the .NET object with, in effect, additional properties which IT pros may find more useful. You can read about the ETS here: `https://powershellstation.com/2009/09/15/powershell-ets-extended-type-system`.

Exploring .NET Methods

With .NET, a method is some action that a .NET object occurrence (an instance method) or the class (a static method) can perform. These methods form the basis for many PowerShell cmdlets. For example, you can stop a Windows process by using the `Stop-Process` cmdlet. The cmdlet then uses the `Kill()` method of the associated Process object. As a general best practice, you should use cmdlets wherever possible. You should only use .NET classes and methods directly where there is no alternative.

.NET methods can be beneficial for performing some operations that have no PowerShell cmdlet support. And it can be useful too from the command line, for example, when you wish to kill a process. IT professionals are all too familiar with processes that are not responding and need to be killed, something you can do at the GUI using Task Manager. Or, with PowerShell, you can use the `Stop-Process` cmdlet. At the command line, where brevity is useful, you can use `Get-Process` to find the process you want to stop and pipe the output to each process's `Kill()` method. PowerShell then calls the object's `Kill()` method. To help IT professionals, the PowerShell team created the `Kill` alias to the `Stop-Process` cmdlet.

Another great example is encrypting files. Windows's NTFS filesystem includes support for the **Encrypting File System (EFS)** feature. EFS lets you encrypt or decrypt files on your computer with encryption based on X.509 certificates. For details on the EFS and how it works, see `https://docs.microsoft.com/windows/win32/fileio/file-encryption`.

At present, there are no cmdlets to encrypt or decrypt files. The `System.IO.FileInfo` class, however, has two methods you can use: `Encrypt()` and `Decrypt()`. These instance methods encrypt and decrypt a file based on EFS certificates (which Windows can automatically generate). You can use these .NET methods to encrypt or decrypt a file without using the GUI.

As you saw in the *Examining .NET Classes* recipe, you can pipe any object to the `Get-Member` cmdlet to discover the methods for the object. Discovering the specific property names and property value types is simple and easy – no guessing or prayer-based text parsing, so beloved by Linux admins.

Getting ready

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

How to do it...

1. Starting Notepad:

```
notepad.exe
```

2. Obtaining methods on the Notepad process

```
$Notepad = Get-Process -Name Notepad
$Notepad | Get-Member -MemberType Method
```

3. Using the `Kill()` method

```
$Notepad |
  ForEach-Object {$_.Kill()}
```

4. Confirming Notepad process is destroyed

```
Get-Process -Name Notepad
```

5. Creating a new folder and some files

```
$Path = 'C:\Foo\Secure'
New-Item -Path $Path -ItemType directory -ErrorAction
SilentlyContinue  |
  Out-Null
1..3 | ForEach-Object {
  "Secure File" | Out-File "$Path\SecureFile$_.txt"
}
```

6. Viewing files in `$Path` folder

```
$Files = Get-ChildItem -Path $Path
$Files | Format-Table -Property Name, Attributes
```

7. Encrypting the files

```
$Files | ForEach-Object Encrypt
```

8. Viewing file attributes

```
Get-ChildItem -Path $Path |
  Format-Table -Property Name, Attributes
```

9. Decrypting the files

```
$Files| ForEach-Object {
  $_.Decrypt()
}
```

10. Viewing the file attributes

```
Get-ChildItem -Path $Path |
  Format-Table -Property Name, Attributes
```

How it works...

In *step 1*, you start the Windows Notepad application. There is no console output, but you do see Notepad pop up like this:

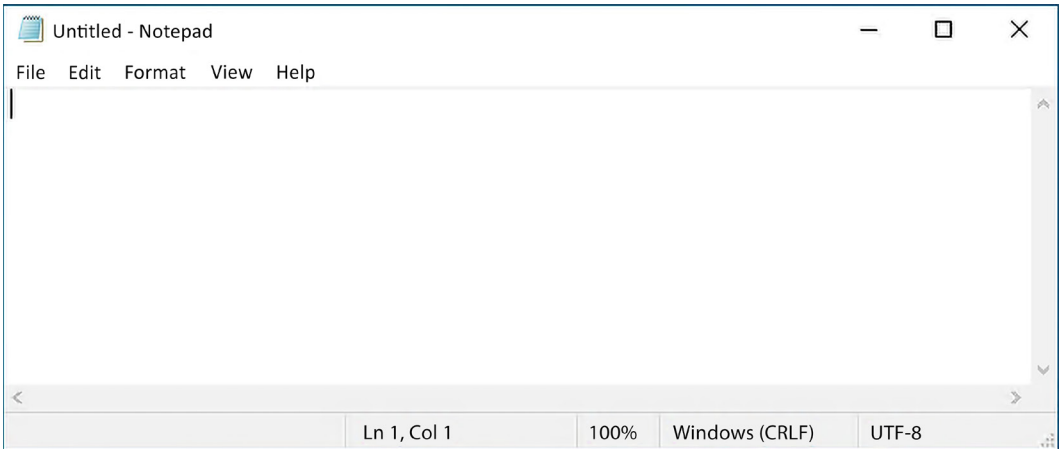


Figure 3.19: Starting Notepad

In *step 2*, you obtain the methods from the object representing the Notepad process, with output like this:

```
PS C:\Foo> # 2. Obtaining methods on the Notepad process
PS C:\Foo> $Notepad = Get-Process -Name Notepad
PS C:\Foo> $Notepad | Get-Member -MemberType Method

   TypeName: System.Diagnostics.Process

Name                      MemberType Definition
----                      ---------- ----------
BeginErrorReadLine        Method     void BeginErrorReadLine()
BeginOutputReadLine       Method     void BeginOutputReadLine()
CancelErrorRead           Method     void CancelErrorRead()
CancelOutputRead          Method     void CancelOutputRead()
Close                     Method     void Close()
CloseMainWindow           Method     bool CloseMainWindow()
Dispose                   Method     void Dispose(), void IDisposable.Dispose()
Equals                    Method     bool Equals(System.Object obj)
GetHashCode               Method     int GetHashCode()
GetLifetimeService        Method     System.Object GetLifetimeService()
GetType                   Method     type GetType()
InitializeLifetimeService Method     System.Object InitializeLifetimeService()
Kill                      Method     void Kill(), void Kill(bool entireProcessTree)
Refresh                   Method     void Refresh()
Start                     Method     bool Start()
ToString                  Method     string ToString()
WaitForExit               Method     void WaitForExit(), bool WaitForExit(int milliseconds)
WaitForExitAsync          Method     System.Threading.Tasks.Task WaitForExitAsync(System.Threading.CancellationToken cancellationToken)
WaitForInputIdle          Method     bool WaitForInputIdle(), bool WaitForInputIdle(int milliseconds)
```

Figure 3.20: Viewing Notepad's methods

With *step 3*, you use the process's `Kill()` method to kill the Notepad process, which generates no console output, but you see the Notepad window disappear. In *step 4*, you confirm that the Notepad process no longer exists, with output like this:

```
PS C:\Foo> # 4. Confirming Notepad process is destroyed
PS C:\Foo> Get-Process -Name Notepad
Get-Process: Cannot find a process with the name "Notepad". Verify the process name and call the cmdlet again.
```

Figure 3.21: Ensuring the Notepad process no longer exists

To illustrate encrypting and decrypting files using .NET methods, in *step 5*, you create a new folder and some files, which creates no output. In *step 6*, you view the attributes of the three files created, with output like this:

```
PS C:\Foo> # 6. Viewing files in $Path folder
PS C:\Foo> $Files = Get-ChildItem -Path $Path
PS C:\Foo> $Files | Format-Table -Property Name, Attributes

Name            Attributes
----            ----------
SecureFile1.txt    Archive
SecureFile2.txt    Archive
SecureFile3.txt    Archive
```

Figure 3.22: Viewing newly created files and their attributes

In *step 7*, you use the `Encrypt()` method to encrypt the files, generating no output. In *step 8*, you view the files and their attributes again, with this output:

```
PS C:\Foo> # 8. Viewing file attributes
PS C:\Foo> Get-ChildItem -Path $Path |
             Format-Table -Property Name, Attributes

Name                     Attributes
----                     ----------
SecureFile1.txt   Archive  Encrypted
SecureFile2.txt   Archive  Encrypted
SecureFile3.txt   Archive  Encrypted
```

Figure 3.23: Viewing encrypted files and file attributes

In *step 9*, you decrypt the previously encrypted files, generating no output. In the final step, *step 10*, you examine the file attributes to ensure that the files are now decrypted, with the following results:

```
PS C:\Foo> # 10. Viewing the -File attributes
PS C:\Foo> Get-Childltem -Path $Path |
             Format-Table -Property Name, Attributes

Name            Attributes
----            ----------
SecureFile1.txt    Archive
SecureFile2.txt    Archive
SecureFile3.txt    Archive
```

Figure 3.24: Viewing decrypted files and file attributes

There's more...

In *step 2*, you view the methods you can invoke on a process object. As shown in the figure, one of those methods is the `Kill()` method. In *step 3*, you use that method to stop the Notepad process. The `Kill()` method is an instance method, meaning you invoke it to kill (stop) a specific process. You can read more about this .NET method at `https://docs.microsoft.com/dotnet/api/system.diagnostics.process.kill`.

The output in *step 4* illustrates an error occurring within VS Code. If you use the PowerShell 7 console, you may see a slightly different output, although with the same actual error message.

In *steps 7* and *step 9*, you use the `FileInfo` objects (created by `Get-ChildItem`) and call the `Encrypt()` and `Decrypt()` methods. These steps demonstrate using a .NET method to achieve some objectives without specific cmdlets.

Best practice suggests always using cmdlets where you can. You should also note that the syntax in the two steps is different. In *step 7*, you use a more modern syntax which calls the `Encrypt` method for each file. In *step 9*, you use an older syntax that does the same, albeit with more characters. Both syntax methods work.

In *step 7*, while you get no console output, Windows may generate a popup (aka Toast) to tell you that you should back up your encryption key. Backing up the key is a good idea. If you lose the key, you lose all access to the file(s) – so be very careful.

Creating a C# Extension

For most day-to-day operations, the commands provided by PowerShell, from Windows features, or third-party modules, provide all the functionality you need to manage your systems. In some cases, as you saw in the *Leveraging .NET Methods* recipe, commands do not exist to achieve your goal. In those cases, you can use the methods provided by .NET.

There are also cases where you need to perform more complex operations without a PowerShell cmdlet or direct .NET support. You may, for example, have a component of an ASP.NET web application written in C#, but you now wish to repurpose it for administrative scripting purposes.

PowerShell makes it easy to add a class, based on .NET language source code, into a PowerShell session. You supply the C# code, and PowerShell creates a .NET class that you can use the same way you use .NET methods (and using virtually the same syntax). You use the `Add-Type` cmdlet and specify the C# code for your class/type(s). PowerShell compiles the code and loads the resultant class into your PowerShell session.

An essential aspect of .NET methods is that a method can have multiple definitions or calling sequences. Known and method overloads, these different definitions allow you to invoke a method using different sets of parameters. This is not dissimilar to PowerShell's use of parameter sets. For example, the `System.String` class, which PowerShell uses to hold strings, contains the `Trim()` method. You use that method to remove extra characters, usually space characters, from the start or end of a string (or both). The `Trim()` method has three different definitions, which you view in this recipe. Each overloaded definition trims characters from a string slightly differently. To view more details on this method, and the three overloaded definitions, see `https://docs.microsoft.com/dotnet/api/system.string.trim`.

In this recipe, you create and use two simple classes with static methods.

Getting ready

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

How to do it...

1. Examining overloaded method definition

```
("a string").Trim
```

2. Creating a C# class definition in here string

```
$NewClass = @"
namespace Reskit {
   public class Hello {
     public static void World() {
         System.Console.WriteLine("Hello World!");
     }
   }
  }
"@
```

3. Adding the type into the current PowerShell session

```
Add-Type -TypeDefinition $NewClass
```

4. Examining method definition

```
[Reskit.Hello]::World
```

5. Using the class's method

```
[Reskit.Hello]::World()
```

6. Extending the code with parameters

```
$NewClass2 = @"
using System;
using System.IO;
namespace Reskit {
  public class Hello2  {
    public static void World() {
      Console.WriteLine("Hello World!");
    }
```

```
    public static void World(string name) {
      Console.WriteLine("Hello " + name + "!");
    }
  }
}
"@
```

7. Adding the type into the current PowerShell session

```
Add-Type -TypeDefinition $NewClass2
```

8. Viewing method definitions

```
[Reskit.Hello2]::World
```

9. Calling with no parameters specified

```
[Reskit.Hello2]::World()
```

10. Calling new method with a parameter

```
[Reskit.Hello2]::World('Jerry')
```

How it works...

In *step 1*, you examine the definitions of the `Trim()` method for a string, with output like this:

```
PS C:\Foo> # 1. Examining overloaded method definition
PS C:\Foo> ("a string").Trim

OverloadDefinitions
-------------------
string Trim()
string Trim(char trimChar)
string Trim(Params char[] trimChars)
```

Figure 3.25: Viewing the Trim() methods on System.String

In *step 2*, you create a C# class definition for the `Hello` class that contains one method (`World()`). In *step 3*, you add this class to the current PowerShell session. These two steps create no output at the console.

In *step 4*, you examine the new class's method signature with output like this:

```
PS C:\Foo> # 4. Examining method definition
PS C:\Foo> [Reskit.Hello]::World

OverloadDefinitions
-------------------
static void World()
```

Figure 3.26: Viewing the World() method definition on the newly created class

In *step 5*, you use the method, with output like this:

```
PS C:\Foo> # 5. Using the class's method
PS C:\Foo> [Reskit.Hello]::World()
Hello World!
```

Figure 3.27: Using the World() method

In *step 6*, you define a more complex class definition for the `Hello` class, this time with two method definitions for the `World` method. In *step 7*, you add this new class definition to your current PowerShell session. These two steps generate no console output.

In *step 8*, you examine the two method signatures with output like this:

```
PS C:\Foo> # 8. Viewing method definitions
PS C:\Foo> [Reskit.Hello2]::World

OverloadDefinitions
-------------------
static void World()
static void World(string name)
```

Figure 3.28: Viewing the new method definitions for Hello2

In *step 9*, you invoke the `World()` method on the `Hello2` class – with output like this:

```
PS C:\Foo> # 9. Calling with no parameters specified
PS C:\Foo> [Reskit.Hello2]::World()
Hello World!
```

Figure 3.29: Viewing the new method definitions for Hello2

In *step 10*, you call the new method but specify a string as a parameter. The output of this step is as follows:

```
PS C:\Foo> # 10. Calling new method with a parameter
PS C:\Foo> [Reskit.Hello2]::World('Jerry')
Hello Jerry!
```

Figure 3.30: Calling the new method with a parameter

There's more...

There are several ways you discover method overloads. In *step 1*, you examine the different definitions of a method, the `Trim()` method of the `System.String` class. This step creates a pseudo-object containing a string and then looks at the method definitions for that class.

PowerShell creates an unnamed object (in the .NET managed heap) in this step. As soon as the command is complete, .NET marks the memory as reclaimable. At some point later, .NET runs the GC process to reclaim the memory used by this temporary object and re-organizes the managed heap.

As you can see, there are three overloaded definitions – three different ways you can invoke the `Trim()` method. You use the first overloaded definition to remove both leading and trailing space characters from a string, probably the most common usage of `Trim()`. With the second definition, you specify a specific character and .NET removes any leading or trailing occurrences of that character. With the third definition, you specify an array of characters to trim from the start or end of the string. The last two definitions are less useful in most cases, although it depends on your needs. The extra flexibility is useful.

In *step 3*, you used the `Add-Type` cmdlet to add the class definition into your current workspace. The cmdlet compiles the C# code, creates, and then loads a DLL, enabling you to use the classes. If you intend to use this add-in regularly, you could add *step 2* and *step 3* to your PowerShell profile file. Alternatively, if you use this method within a script, you could add the steps to the start of the script.

As you can see from the C# code in *step 6*, you can add multiple definitions to a method. Overloaded methods can provide great value for both classes within .NET or in your code. The .NET method documentation, which you can find at `https://docs.microsoft.com`, is mainly oriented toward developers rather than IT professionals. If you have issues with a .NET method, feel free to fire off a query on one of the many PowerShell support forums, such as the PowerShell group at Spiceworks: `https://community.spiceworks.com/programming/powershell`.

Creating a cmdlet

As noted previously in this chapter, for most operations, the commands and cmdlets available to you natively provide all the functionality you need. In the *Creating a C# Extension* recipe, you saw how you could create a class definition and add it to PowerShell. In some cases, you may wish to expand on the class definition and create a custom cmdlet.

Creating a compiled cmdlet requires you to either use a tool such as Visual Studio or use the free tools provided by Microsoft as part of the .NET Core **Software Development Kit (SDK)**. The free tools in the SDK are more than adequate to help you to create a cmdlet using C#. Microsoft's Visual Studio, whether the free community edition or the commercial releases, is a rich and complex tool whose inner workings are well outside the scope of this book.

As in the *Creating a C# Extension* recipe, an important question you should be asking is when/why should you create a cmdlet? Aside from the perennial "because you can" excuse, there are reasons why an extension or a cmdlet might be a good idea. You can create a cmdlet to improve performance. In some cases, a compiled cmdlet is faster than doing operations using a PowerShell script. For some applications, using a cmdlet is to perform a specific action may be difficult or not possible directly. PowerShell does not have support for asynchronous operations and **Language Independent Query (LINQ)**, which are more straightforward in C#. A developer could write the cmdlet in C#, allowing you to use it easily with PowerShell.

To create a cmdlet, you need to install the .NET SDK. The SDK is a free download you get via the internet. The SDK contains all the tools you need to create a cmdlet. Sadly, there is no easy way to download and install the SDK programmatically – you need to use the Windows GUI – you have to use your browser and navigate to a web page, then download and run the SDK installer.

Getting ready

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

How to do it...

1. Installing the .NET SDK
2. Creating the cmdlet folder

```
New-Item -Path C:\Foo\Cmdlet -ItemType Directory -Force
Set-Location C:\Foo\Cmdlet
```

3. Creating a new class library project

```
dotnet new classlib --name SendGreeting
```

4. Viewing contents of new folder

```
Set-Location -Path .\SendGreeting
Get-ChildItem
```

5. Creating and displaying `global.json`

```
dotnet new globaljson
Get-Content -Path .\global.json
```

6. Adding PowerShell package

```
$SourceName = 'Nuget.org https://api.nuget.org/v3/index.json'
dotnet nuget add source --name $SourceName
dotnet add package PowerShellStandard.Library
```

7. Create the cmdlet source file

```
$Cmdlet = @"
using System.Management.Automation;  // Windows PowerShell assembly.
namespace Reskit
{
  // Declare the class as a cmdlet
  // Specify verb and noun = Send-Greeting
  [Cmdlet(VerbsCommunications.Send, "Greeting")]
  public class SendGreetingCommand : PSCmdlet
  {
    // Declare the parameters for the cmdlet.
    [Parameter(Mandatory=true)]
    public string Name
    {
      get { return name; }
      set { name = value; }
    }
    private string name;
    // Override the ProcessRecord method to process the
    // supplied name and write a geeting to the user by
    // calling the WriteObject method.
    protected override void ProcessRecord()
    {
      WriteObject("Hello " + name + " - have a nice day!");
    }
  }
}
```

```
"@
$Cmdlet | Out-File .\SendGreetingCommand.cs
```

8. Removing the unused class file

```
Remove-Item -Path .\Class1.cs
```

9. Building the cmdlet

```
dotnet build
```

10. Importing the DLL holding the cmdlet

```
$DLLPath = '.\bin\Debug\net6.0\SendGreeting.dll'
Import-Module -Name $DLLPath
```

11. Examining the module's details

```
Get-Module SendGreeting
```

12. Using the cmdlet

```
Send-Greeting -Name 'Jerry Garcia'
```

How it works...

In *step 1*, you install the .NET SDK. You begin by using the browser to navigate to the .NET download site, which looks like this:

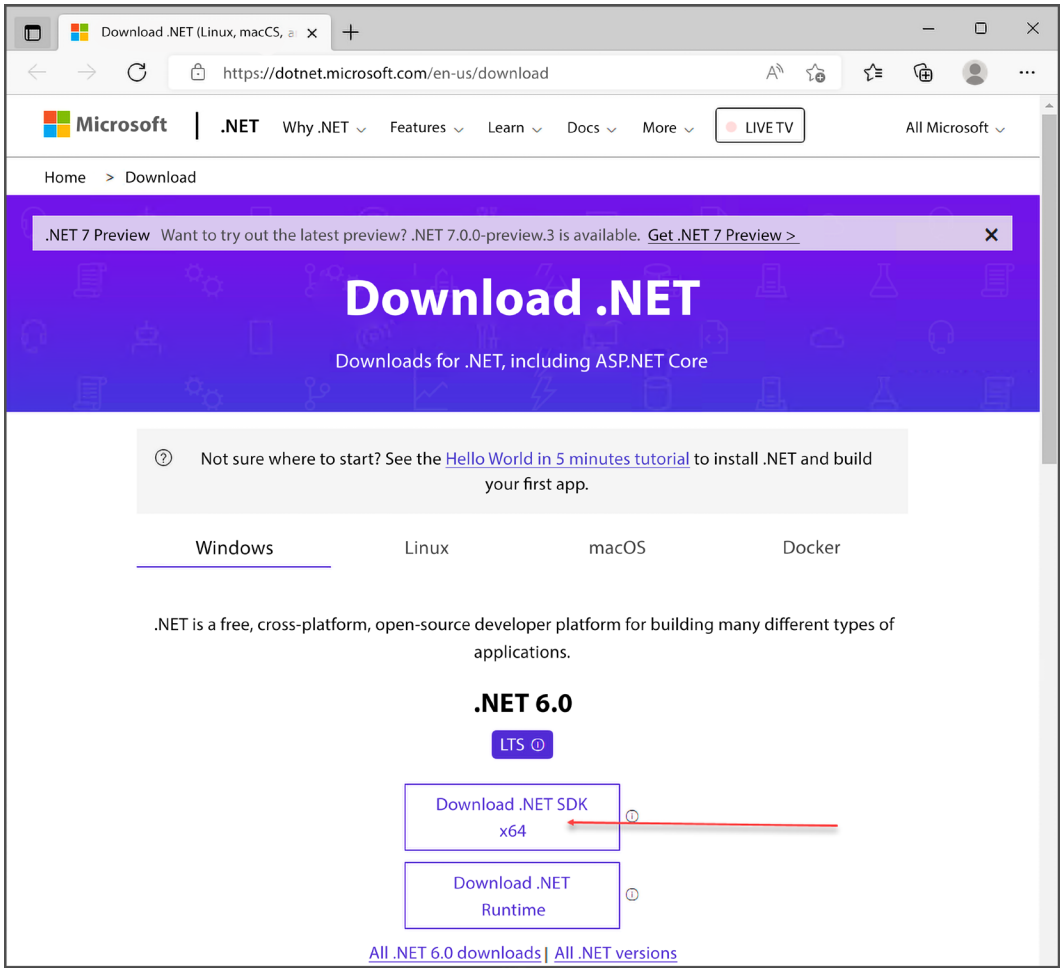


Figure 3.31: Downloading the .NET SDK

If you click on the button, your browser should download the installation program. When the browser finishes the download, you should see something like this:

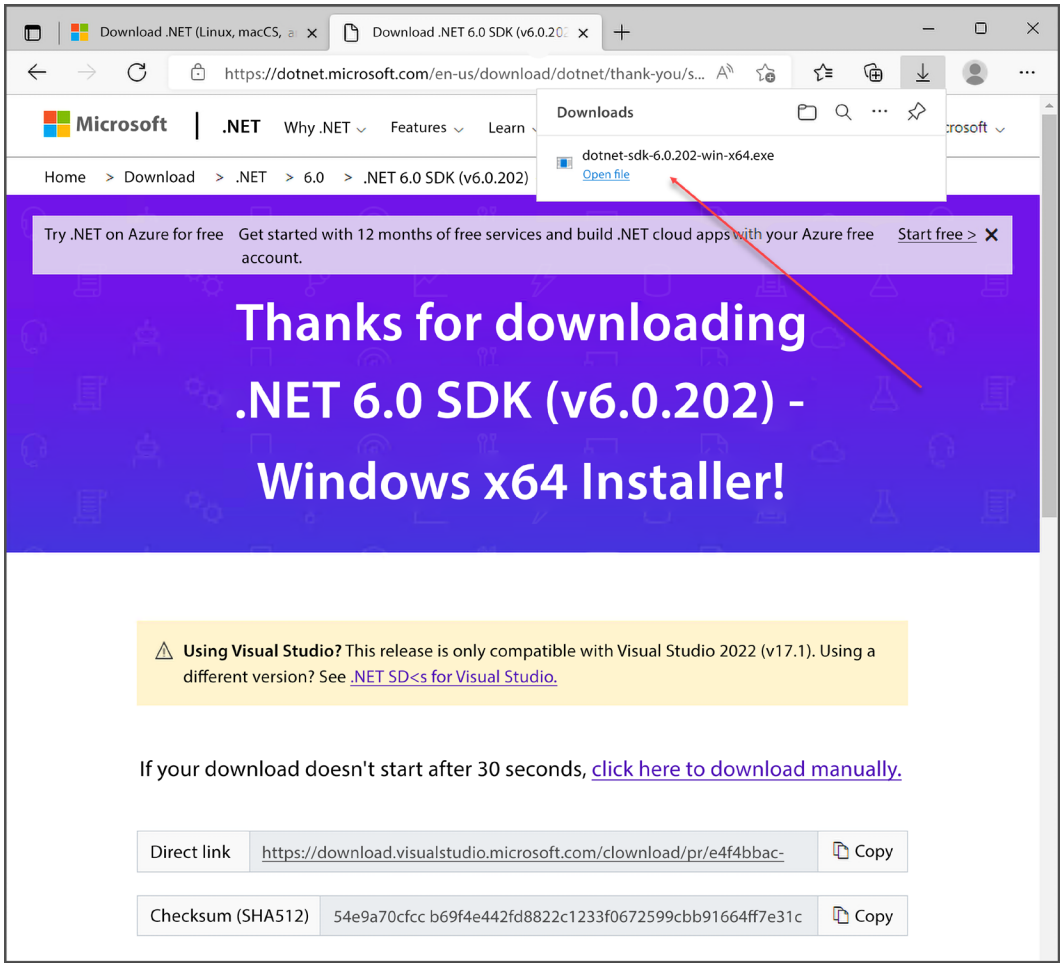


Figure 3.32: Downloading the .NET SDK

Click on **Open File** to run the .NET SDK installer. You should now see an option to install the SDK like this:

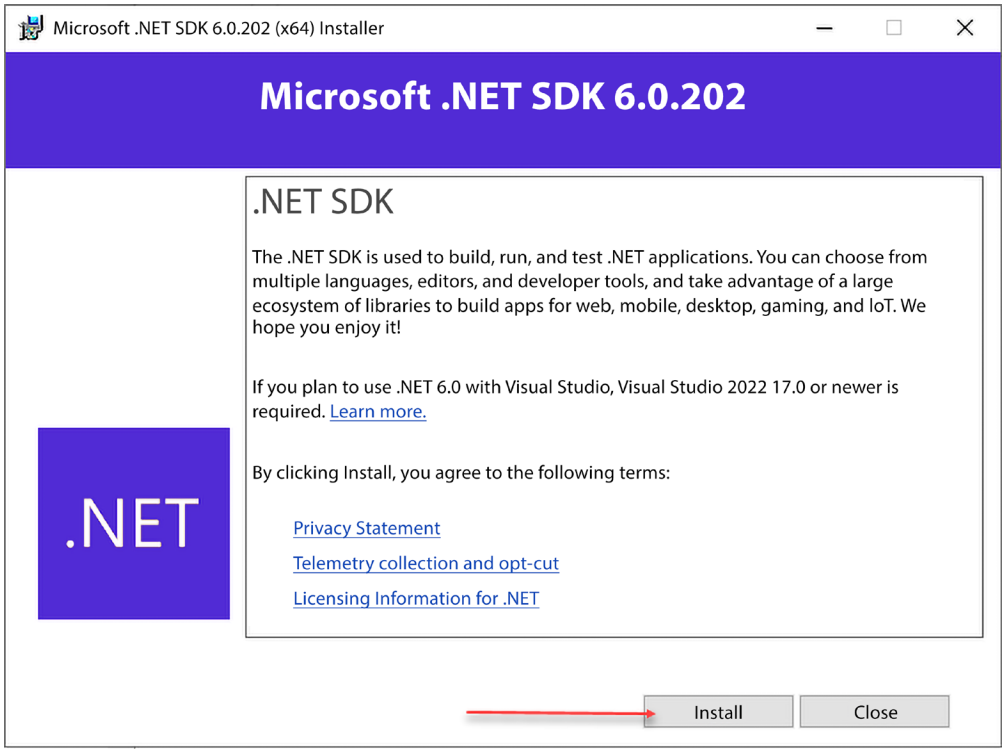


Figure 3.33: Installing the .NET SDK

When you click on the **Install** button, the installation program installs the SDK and, when complete, shows you the following dialog box:

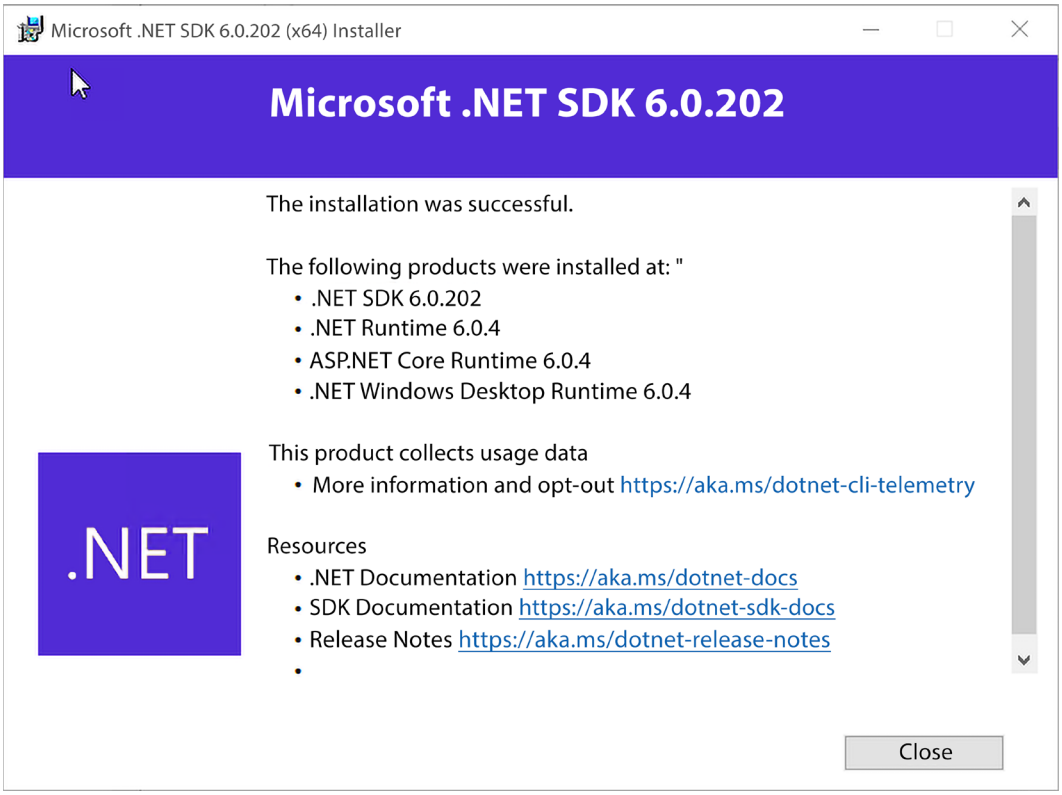


Figure 3.34: Completing the installation of the .NET SDK

Once you have the SDK installed, you need to restart PowerShell to carry out the remainder of this recipe.

In *step 2*, you create a new folder (`C:\Foo\Cmdlet`), which you use for developing your cmdlet. This step generates the following output:

```
PS C:\Foo> # 2. Creating the cmdlet folder
PS C:\Foo> New-Item -Path C:\Foo\Cmdlet -ItemType Directory -Force

    Directory: C:\Foo

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         02/05/2022    16:59                Cmdlet
```

Figure 3.35: Creating the cmdlet folder

In *step 3*, you create a new class library project for your cmdlet, which generates the following output:

```
PS C:\Foo> # 3. Creating a new class library project
PS C:\Foo> Set-Location C:\Foo\Cmdlet
PS C:\Foo\Cmdlet> dotnet new classlib --name SendGreeting

Welcome to .NET 6.0!
---------------------
SDK Version: 6.0.202

Telemetry
---------
The .NET tools collect usage data in order to help us improve your experience. It is collected
by Microsoft and shared with the community. You can opt-out of telemetry by setting the
DOTNET_CLI_TELEMETRY_OPTOUT environment variable to '1' or 'true' using your favorite shell.

Read more about .NET CLI Tools telemetry: https://aka.ms/dotnet-cli-telemetry

----------------
Installed an ASP.NET Core HTTPS development certificate.
To trust the certificate run 'dotnet dev-certs https --trust' (Windows and macOS only).
Learn about HTTPS: https://aka.ms/dotnet-https
----------------
Write your first app: https://aka.ms/dotnet-hello-world
Find out what's new: https://aka.ms/dotnet-whats-new
Explore documentation: https://aka.ms/dotnet-docs
Report issues and find source on GitHub: https://github.com/dotnet/core
Use 'dotnet --help' to see available commands or visit: https://aka.ms/dotnet-cli
--------------------------------------------------------------------------------------
The template "Class Library" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on C:\Foo\Cmdlet\SendGreeting\SendGreeting.csproj...
  Determining projects to restore...
  Restored C:\Foo\Cmdlet\SendGreeting\SendGreeting.csproj (in 95 ms).
Restore succeeded.
```

Figure 3.36: Creating a new .NET project for your cmdlet

In *step 4*, you view the contents of the folder contents, which look like this:

```
PS C:\Foo\Cmdlet> # 4. Viewing contents of new folder
PS C:\Foo\Cmdlet> Set-Location -Path .\SendGreeting
PS C:\Foo\Cmdlet\SendGreeting> Get-ChildItem


    Directory: C:\Foo\Cmdlet\SendGreeting


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d----          02/05/2022    16:20                obj
-a---          02/05/2022    16:20             57 Class1.cs
-a---          02/05/2022    16:20            215 SendGreeting.csproj
```

Figure 3.37: Viewing the contents of the cmdlet project's folder

To tell the .NET build engine which version of .NET to compile your cmdlet against, you create a `global.json` file within your project, as shown in *step 5*. The step displays the following output:

```
PS C:\Foo\Cmdlet\SendGreeting> # 5. Creating and displaying global.json
PS C:\Foo\Cmdlet\SendGreeting> dotnet new globaljson
The template "global.json file" was created successfully.
PS C:\Foo\Cmdlet\SendGreeting> Get-Content -Path .\global.json
{
  "sdk": {
    "version": "6.0.202"
  }
}
```

Figure 3.38: Creating and viewing the global.json file

In *step 6*, you add the PowerShell package to the cmdlet's project with the following output:

```
PS C:\Foo\Cmdlet\SendGreeting> # 6. Adding PowerShell package
PS C:\Foo\Cmdlet\SendGreeting> $Sourcename = 'Nuget.org https://api.nuget.org/v3/index.json'
PS C:\Foo\Cmdlet\SendGreeting> dotnet nuget add source --name $SourceName
Package source with Name: nuget.org added successfully.
PS C:\Foo\Cmdlet\SendGreeting> dotnet add package PowerShellStandard.Library
  Determining projects to restore...
  Writing C:\Users\Administrator\AppData\Local\Temp\2\tmpCD91.tmp
info : Adding PackageReference for package 'PowerShellStandard.Library' into
       project 'C:\Foo\Cmdlet\SendGreeting\SendGreeting.csproj'.
info :   GET https://api.nuget.org/v3/registration5-gz-semver2/powershellstandard.library/index.json
info :   OK https://api.nuget.org/v3/registration5-gz-semver2/powershellstandard.library/index.json 133ms
info : Restoring packages for C:\Foo\Cmdlet\SendGreeting\SendGreeting.csproj...
info :   GET https://api.nuget.org/v3-flatcontainer/powershellstandard.library/index.json
info :   OK https://api.nuget.org/v3-flatcontainer/powershellstandard.library/index.json 133ms
info :   GET https://api.nuget.org/v3-flatcontainer/powershellstandard.library/5.1.1/
           powershellstandard.library.5.1.1.nupkg
info :   OK https://api.nuget.org/v3-flatcontainer/powershellstandard.library/5.1.1/
           powershellstandard.library.5.1.1.nupkg 54ms
info : Installed PowerShellStandard.Library 5.1.1 from https://api.nuget.org/v3/index.json with content hash
       e31xJjG+Kjbv6YF3Yq6D4Dl3or8v7LrNF41k3CXrWozW6hR1zcOe5KYuZJaGSiAgLnwP8wcW+I3+IWEzMPZKXQ==.
info : Package 'PowerShellStandard.Library' is compatible with all the specified frameworks in
       project 'C:\Foo\Cmdlet\SendGreeting\SendGreeting.csproj'.
info : PackageReference for package 'PowerShellStandard.Library' version '5.1.1' added to
       file 'C:\Foo\Cmdlet\SendGreeting\SendGreeting.csproj'.
info : Writing assets file to disk. Path: C:\Foo\Cmdlet\SendGreeting\obj\project.assets.json
log  : Restored C:\Foo\Cmdlet\SendGreeting\SendGreeting.csproj (in 820 ms).
```

Figure 3.39: Adding the PowerShell package to the cmdlet's project

In *step 7*, you create a source file containing your cmdlet's C# code. In *step 8*, you remove an unneeded class file previously created by the `dotnet` program. These two steps produce no output.

In *step 9*, you use the `dotnet.exe` program to build your cmdlet, with output like this:

```
PS C:\Foo\Cmdlet\SendGreeting> # 9. Building the cmdlet
PS C:\Foo\Cmdlet\SendGreeting> dotnet build
Microsoft (R) Build Engine version 17.1.1+a02f73656 for .NET
Copyright (C) Microsoft Corporation. All rights reserved.

  Determining projects to restore...
  All projects are up-to-date for restore.
  SendGreeting -> C:\Foo\Cmdlet\SendGreeting\bin\Debug\net6.0\SendGreeting.dll

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:03.79
```

Figure 3.40: Building your cmdlet

In *step 10*, you import the DLL, built by the previous step, that holds your cmdlet. This step creates no output. PowerShell treats this DLL as a binary Powershell module, and in *step 11*, you use the `Get-Module` cmdlet to see the module's details, with output like this:

```
PS C:\Foo\Cmdlet\SendGreeting> # 11. Examining the module's details
PS C:\Foo\Cmdlet\SendGreeting> Get-Module SendGreeting

ModuleType Version    PreRelease Name           ExportedCommands
---------- -------    ---------- ----           ----------------
Binary     1.0.0.0               SendGreeting   Send-Greeting
```

Figure 3.41: Caption please

In *step 12*, you use your cmdlet with a value specified for the cmdlet's `-Name` parameter, with output like this:

```
PS C:\Foo\Cmdlet\SendGreeting> # 12. Using the cmdlet
PS C:\Foo\Cmdlet\SendGreeting> Send-Greeting -Name 'Jerry Garcia'
Hello Jerry Garcia - have a nice day!
```

Figure 3.42: Using the Send-Greeting cmdlet

There's more...

In *step 1*, you download the .NET 6.0 SDK. You need this tool kit to create a cmdlet. There is no obvious way to automate the installation, so you need to run the installer and click through its GUI to install the .NET SDK.

In *step 3*, you create a new class library project. This step also makes the `SendGreeting.csproj` file, as you can see. This file is a Visual Studio .NET C# project file that contains details about the files included in your cmdlet project, assemblies referenced from your code, and more. For more information on the project file, see `https://docs.microsoft.com/en-us/aspnet/web-forms/overview/deployment/web-deployment-in-the-enterprise/understanding-the-project-file`.

In *step 5*, you create the `global.json` file. If you do not create this file, the `dotnet` command will compile your cmdlet with the latest version of .NET loaded on your system. Using this file tells the project build process to use a specific version of .NET Core (such as version 6.0). For an overview of this file, `https://docs.microsoft.com/dotnet/core/tools/global-json`.

In *step 9*, you compile your source code file and create a DLL containing your cmdlet. This step compiles all the source code files in the folder to create the DLL. Therefore, you could have multiple cmdlets in separate source code files and build the entire set in one operation.

In *step 11*, you can see that the module, `SendGreeting`, is a binary module. A binary module is one just loaded directly from a DLL. Using DLLs as binary modules is fine for testing, but in production you should use manifest modules not pure binary module.

You can get more details on manifest files from `https://docs.microsoft.com/en-us/powershell/scripting/developer/module/how-to-write-a-powershell-module-manifest`. You should also move the module, module manifest, and any other module contents such as help files to a supported module location. You might also consider publishing the completed module to either the PowerShell gallery or your internal repository.

Join our community on Discord

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

`https://packt.link/SecNet`