5

Exploring the .NET Framework

In this chapter, we cover the following recipes:

* Exploring .NET Assemblies
* Examining .NET Classes
* Leveraging .NET Methods
* Creating a C# Extension
* Creating a PowerShell Cmdlet

# Introduction

Microsoft first launched .NET in June 2000, amidst a frenzy of marketing zeal, with the code name Next Generation Windows Services. Microsoft seemed to add the .NET moniker to every product. There was Windows .NET Server (later re-renamed Windows Server 2003), Visual Studio.NET and even MapPoint .NET. As is often the case, .NET provided features which were superseded by newer features based on advances in technology. For example, SOAP (Simple Object Access Protocol) and XML-based web services have given way to REST (Representation State Transfer), and JSON (JavaScript Object Notation).

Microsoft made considerable improvements to .NET with each release and added new features based on customer feedback. .NET started as closed-source, but Microsoft has transitioned .NET to open source. PowerShell 7.1 is built on top of and takes advantage of the latest version of .NET Core, .NET 5.0.

For the developer, .NET is primarily an Application Program Interface (API) and a programming model with associated tools and run-time. .NET is an environment in which developers can develop rich applications and web sites. As has been mentioned, PowerShell sits on top and integrates fully with .NET.

For the IT Professional, .NET provides the underpinnings of PowerShell but is otherwise mostly transparent. You use cmdlets to carry out actions. These cmdlets use .NET to carry out their work. You can also access .NET directly from within PowerShell.

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



**Insert image B42024\_05\_01.png**

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 support .NET 5.0, the basis for PowerShell 7.1, on Windows, Linux, and the Apple Mac. For a full list of operating systems which support PowerShell 7, see https://github.com/dotnet/core/blob/master/release-notes/5.0/5.0-supported-os.md.
    - **Common Language Runtime (CLR)** - the CLR is the core of .NET, the managed code environment in which all .NET applications run (including PowerShell). The CLR delivers a managed execution environment as well as type safety and code access security. The CLR manages memory, threads, objects, resource allocation/de-allocation, and garbage collection. The CLR also contains the Just-In-Time (JIT) compiler and the Class Loader (responsible for loading classes into an application at run time). For the PowerShell user, the CLR ‘just works’.
    - **Base Class Libraries** **(BCLs)** - .NET comes with a large number of base class libraries, the fundamental built-in features that developers use to build .NET applications. Each BCL is a DLL which contains .NET classes and types. When you install PowerShell, the installer adds the.NET Framework components into PowerShell’s installation folder. PowerShell developers use these libraries to implement PowerShell cmdlets. You can also call classes in the BCL directly from within PowerShell.
    - **WMI, COM, Win32** - these are traditional application development technologies which you can access via PowerShell. Within PowerShell, you can run programs that use these technologies as well as access them from the console or via a script. Your scripts can be a mixture of .NET, WMI, COM, and Win32 if necessary.
    - **Languages** - this is the language that a developer uses to develop PowerShell cmdlets. You can use an extensive variety of languages, based on your preferences and skill set. Most cmdlet developers use C# although using any .NET supported language would work.
    - **PowerShell** - PowerShell sits on top of these other components. From PowerShell, you can use cmdlets developed using a supported language. And you can use both BCL and WMI/COM/Win32 as well.

Every operating system implements essential computing functions such as I/O, networking and much, much more. The .NET Framework sits on top of the OS and makes use of the OS. The BCLs are .NET’s APIs. And provide a vast range of services to applications. Application developers write applications utilizing the classes and types within these base class libraries.

The CLR is the environment in which all .NET applications, including PowerShell, run. The CLR provide the application developer with a range of facilities and approaches to application development. Still, for the IT Pro, the inner workings of the CLR are not all that important (usually!). For an overview of the CLR, see https://docs.microsoft.com/dotnet/standard/clr/

Win32 is the core Windows API set. Before .NET, developers wrote desktop applications using Win32. The Component Object Model (COM) and Distributed COM (DCOM) were object-oriented approaches to application development. WMI represented another set of APIs that developers use in developing applications. .NET provides a mechanism for an application to access WIN32, COM, and WMI in addition to the base BCL.

For most IT Pros, the language used to develop PowerShell and PowerShell extensions is not directly relevant. With that said, the PowerShell language is broadly compatible with C#. You can describe PowerShell as on the glide scope to C#.

At the top of this stack is PowerShell itself. PowerShell language constructs and syntax make use of the components of the .NET Framework and utilize classes and types in the Base Class libraries.

In this chapter, you examine the assemblies that makeup PowerShell 7.1. You then look at both the classes provided by .NET and how to leverage the methods provided by .NET. You also look at creating a simple C# extension and creating a PowerShell cmdlet

# Exploring .NET Assemblies

With .NET, an assembly holds compiled code which .NET can run and can be either a Dynamic Link Library (DLL) or an executable. Cmdlets and .NET classes are contained in DLLs as you can see in this recipe. Each assembly also contains a manifest which describes what is in the assembly, along with compiled code.

The output of the .NET language compilers is not native machine code (as would be the case for, say, C++. Instead, compilers create Intermediate Language (IL) output which is platform-independent. At run time, .NET loads the assemblies needed by an application and compiles this IL code into native machine code. This approach allows.NET to use the same DLL on both 32-bit and 64-bit versions of Windows. The Just-In-Time compilation process could induce a performance hit due to compiling lots of assembles each time you run an application or load a .NET DLL. To avoid this performance issue, .NET enables a developer to pre-compile assemblies in the global assembly cache.

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

A great feature of PowerShell is the ability for you to invoke a .NET Class method directly or to obtain a static .NET Class value. The syntax for calling a .NET Method or a .NET field demonstrated in numerous recipes in this book, is to enclose the class name in square brackets, and then follow it with two “:” characters followed by the name of the method or static filed.

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

## Getting Ready

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

## How to do it...

1. Counting loaded assemblies

$Assemblies = [System.AppDomain]::CurrentDomain.GetAssemblies()

"Assemblies loaded: {0:n0}" -f $Assemblies.Count

1. Viewing first 10

$Assemblies | Select-Object -First 10

1. Checking assemblies in Windows PowerShell

$SB = {

  [System.AppDomain]::CurrentDomain.GetAssemblies()

}

$PS51 = New-PSSession -UseWindowsPowerShell

$AssIn51 = Invoke-Command -Session $PS51 -ScriptBlock $SB

"Assemblies loaded in Windows PowerShell: {0:n0}" -f $Assin51.Count

1. Viewing Microsoft.PowerShell assemblies

$AssIn51 |

  Where-Object FullName -Match "Microsoft\.Powershell" |

    Sort-Object -Property Location

1. Exploring the Microsoft.PowerShell.Management module

$Mod = Get-Module -Name Microsoft.PowerShell.Management -ListAvailable

$Mod  | Format-List

1. Viewing module manifest

$Manifest = Get-Content -Path $Mod.Path

$Manifest | Select-Object -First 20

1. Discovering the module's assembly

Import-Module -Name Microsoft.PowerShell.Management

$Match = $Manifest | Select-String Modules

$Lube = $Match.Line

$DLL = ($Line -Split '"')[1]

Get-Item -Path $PSHOME\$DLL

1. Viewing associated loaded assembly

$Assemblies2 = [System.AppDomain]::CurrentDomain.GetAssemblies()

$Assemblies2 | Where-Object Location -match $DLL

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

$Commands = $Assemblies2

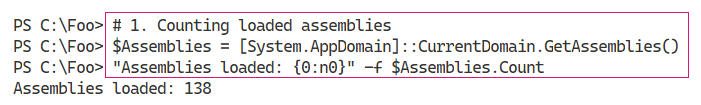
               Where-Object Location -match Commands.Management\.dll

$Commands.GetTypes() |

  Where-Object Name -match "Addcontentcommand$"

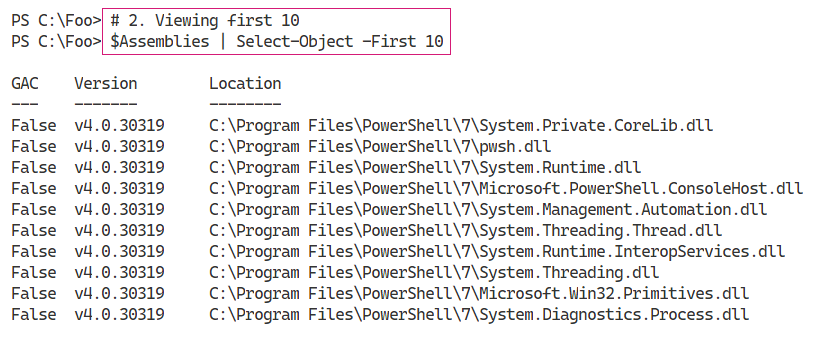
## How it works...

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



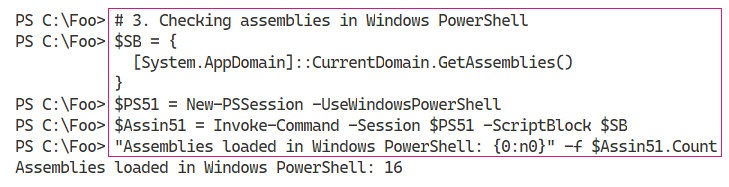
Insert image B42024\_05\_02.png

In step 2, you look at the first 10 assemblies returned, which looks like this;



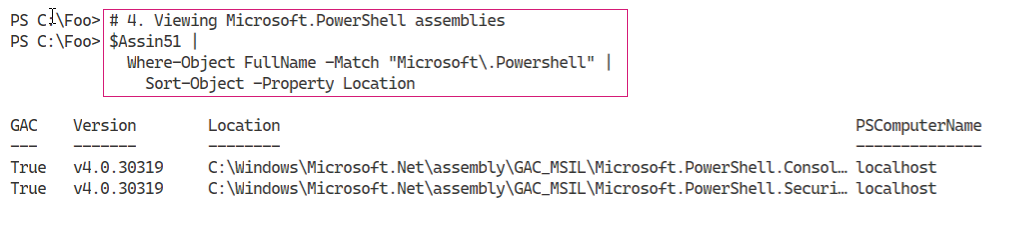
Insert image B42024\_05\_03.png

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



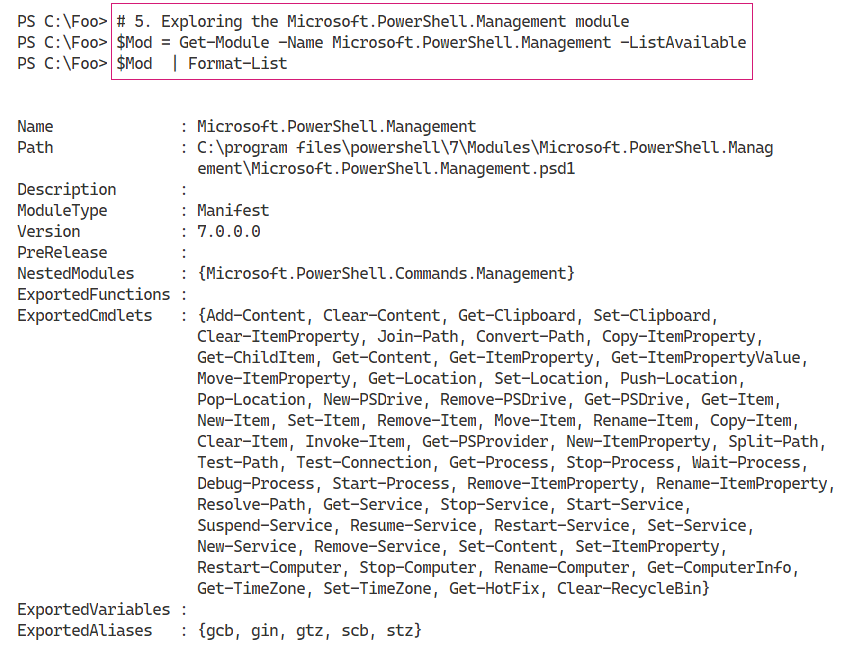
Insert image B42024\_05\_04.png

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



Insert image B42024\_05\_05.png

In step 5, you examine the details of the Microsoft.PowerShell.Management module, which contains numerous core commands in PowerShell. The output of this step looks like this:



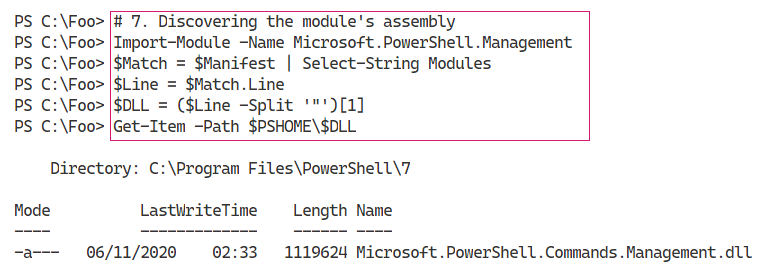
Insert image B42024\_05\_06.png

In step 6, you view the manifest for the Microsoft.PowerShell.Management module. The figure below shows the first 20 lines of the manifest, which looks like this:



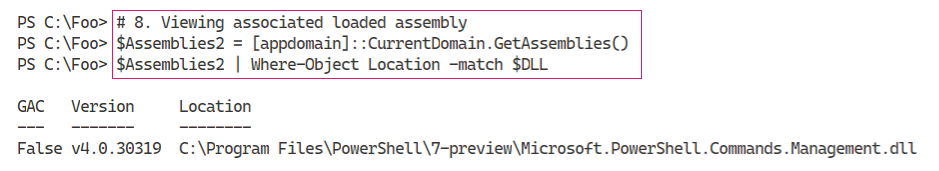
Insert image B42024\_05\_07.png

In step 7, you extract the name of the DLL implementing the Microsoft.PowerShell.Management module and examine the location on disk, which looks like this:



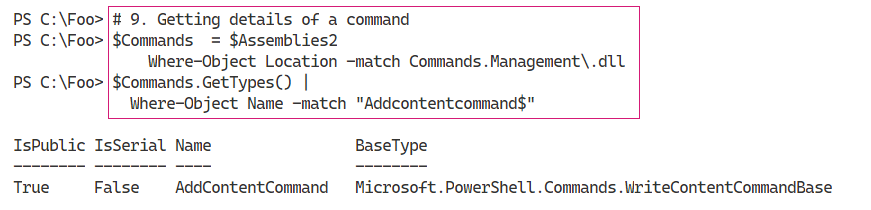
Insert image B42024\_05\_08.png

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



Insert image B42024\_05\_09.png

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



Insert image B42024\_05\_10.png

## There's more...

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

In step 1, you use the GetAssemblies() method to return all the assemblies currently loaded by PowerShell 7.1. As you can see, the syntax is different from calling PowerShell cmdlets.

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

In step 6, you view the first 20 lines of 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 this figure, you can see that the command Add-Content is implemented within Microsoft.PowerShell.Management.dll.

In step 7, you discover the DLL which implements inter alias 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 command is implemented by the AddContentCommand class within the Assembly’s DLL. For the curious, navigate to https://github.com/PowerShell/PowerShell/blob/master/src/Microsoft.PowerShell.Commands.Management/commands/management/AddContentCommand.cs, and you can read the source code for this cmdlet.

# Examining .NET Classes

With .NET, a class defines an object. Objects and object occurrences are fundamental to PowerShell, where cmdlets produce and consume objects. For example, the Get-Process command returns objects of the class System.Diagnostics.Process. If you use Get-ChildItem to return files and folders, the output is a set of objects based on the class System.IO.FileInfo and System.IO.DirectoryInfo.

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

Within .NET, you have two kinds of object definitions: class and type. A type defines a simple object that lives, at runtime on your CPUs’ stack. Classes, being more complex, live in the global heap. The global heap is a large area of memory which .NET uses to hold typically 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 on performance. For example, the System.Array class creates objects of fixed length. If you add an item to an array, .NET creates a new copy of the array including the addition), and removes the old one. If you are adding a few items to the array, the performance hit is negligible. But if you are adding millions of occurrences, the performance can suffer significantly. To avoid this, you can just use the ArrayList class, which supports adding/removing items from an array without the performance penalty. For more information on GC and performance, see https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/performance.

.NET 5.0 features many improvements to the garbage collection process. You can read more about the GC improvements in .NET 5.0 here https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-5/.

In NET, occurrences of every class or type can include members, including Properties, Methods, and Events. A property is an attribute of an occurrence of a class. An occurrence of the System.IO.FileInfo object, for example, has a FullName property A method is effectively a function you can call which can do something to an object occurrence. You look at .NET Methods in more details in “Leveraging .NET Methods”. With .NET, an event is something that can happen to an object occurrence, such when an event generated when a Windows process has completed. .NET events are not covered in this book, although using WMI events is described in Chapter 16 in the “Using WMI Events” recipe.

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

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

In this recipe, you look at some everyday objects created automatically by PowerShell. You also examine the static fields of the [Int] .Net class.

## Getting Ready

You run this recipe on SRV1, a workgroup host running Windows Server Datacenter edition. This host has PowerShell, and VS Code installed.

## How to do it...

1. Creating a Fileinfo object

$FILE = Get-ChildItem -Path $PSHOME\pwsh.exe

$FILE

1. Discovering the underlying class

$TYPE = $FILE.GetType().FullName

".Net Class name: $TYPE"

1. Getting member types of Fileinfo object

$File |

  Get-Member |

    Group-Object -Property MemberType |

      Sort-Object -Property Count -Descending

1. Discovering properties of a Windows service

Get-Service |

  Get-Member -MemberType Property

1. 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"

1. 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

1. Looking at static properties of a class

$Max = [Int32]::MaxValue

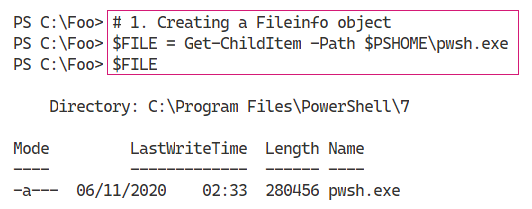
$Min = [Int32]::MinValue

"Minimum value [$Min]"

"Maximum value [$Max]"

## How it works...

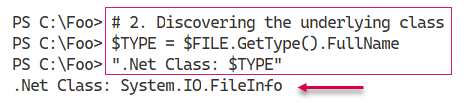
In step 1, you use the Get-ChildItem cmdlet to return an object representing the PowerShell 7 executable file, with output like this:



Insert image B42024\_05\_11.png

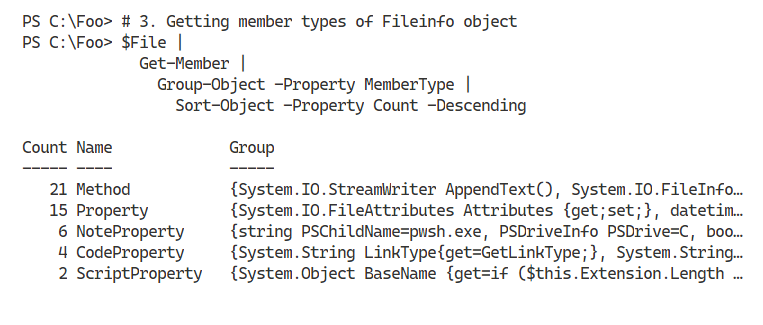
You can determine the class name of an object by using the GetType() method. This method is present on every object and returns information about the object’s type.

In step 2, you discover and display a full class name, which looks like this:



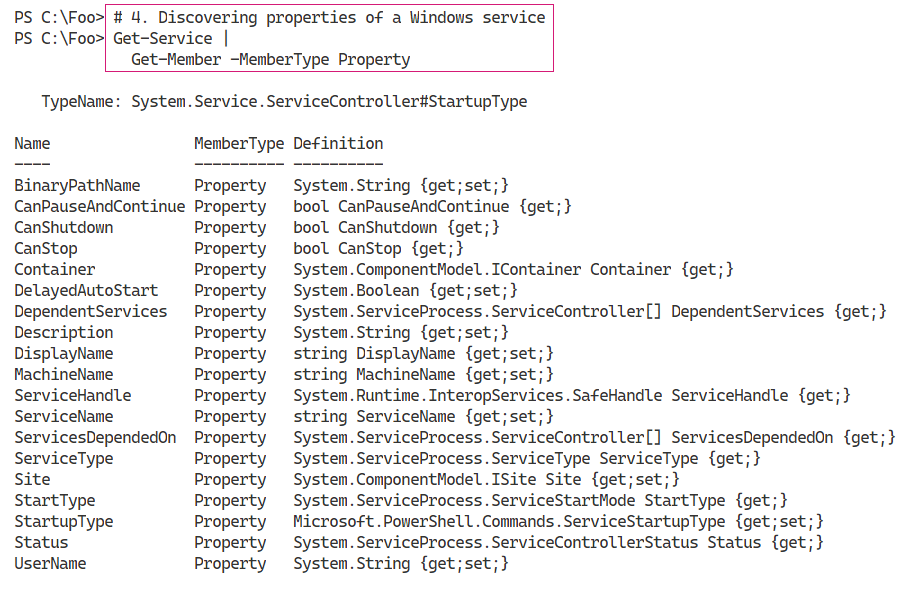
Insert image B42024\_05\_12.png

In step 3, you use Get-Member to display the different member types contained within a FileInfo object, with output like this:



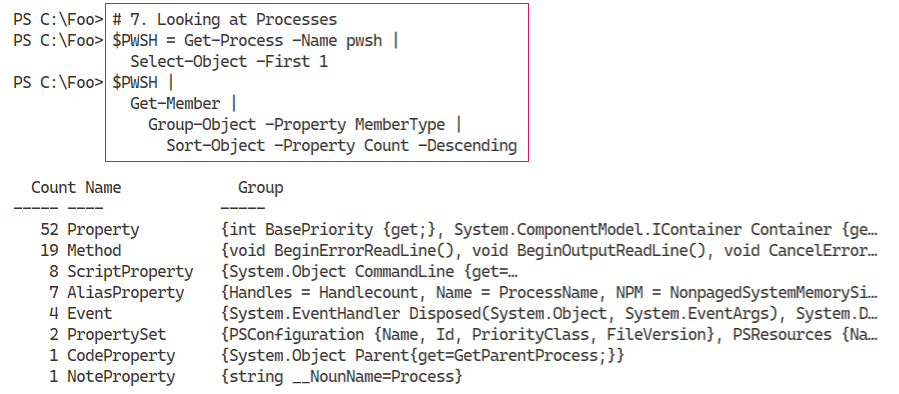
Insert image B42024\_05\_13.png

In step 4, you examine objects returned from then Get-Service cmdlet. The output of this stop looks like this:



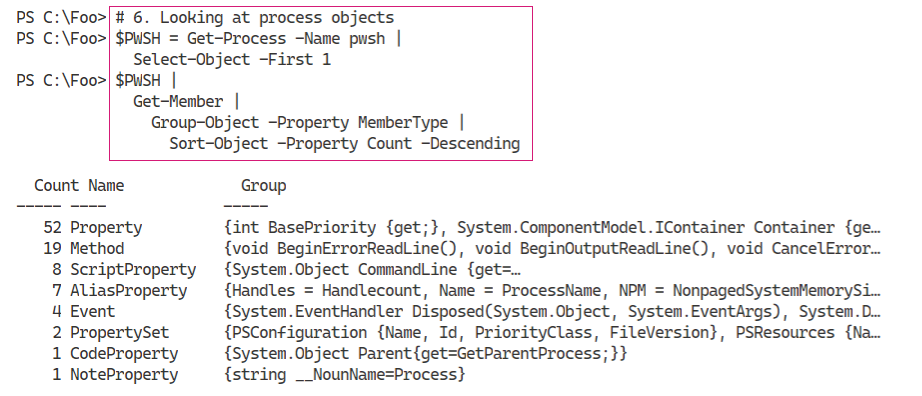
Insert image B42024\_05\_14.png

In step 5, you examine the details of an integer, a System.Int32 data type. You use the GetType() method, which is present on all .NET objects, to return the variable’s type information. Including the full class name of the class. The output looks like this:



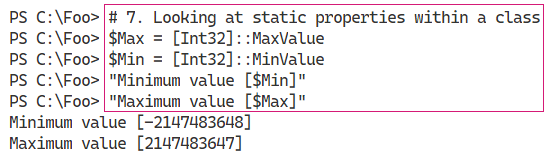
Insert image B42024\_05\_15.png

In step 6, you examine the member types of a System.Diagnostics.Process object. The Get‑Process cmdlet returns objects of this class. The output of this step looks like this:



Insert image B42024\_05\_16.png

In step 7, you examine two static values of the System.Int32 class. These two fields hold the largest and smallest values, respectively, which a 32-bit signed integer can hold. The output looks like this:



Insert image B42024\_05\_17.png

## There's more...

In step 1, you create an object representing pwsh.exe. This object’s full type name is System.IO.FileInfo. In .NET, classes live in namespaces and, in general, namespaces equate to DLLs. In this case, the object is in the System.IO namespace, and the class is contained in the System.IO.FileSystem.DLL. You can discover namespace and DLL details by examining the class’s documentation - in this case, https://docs.microsoft.com/dotnet/api/system.io.fileinfo?view=net-5.0.

As you use your favourite search engine to learn more about .NET classes that might be useful, note that many sites describe the class without the namespace, simply as the FileInfo class, while others spell out the class as System.IO.FileInfo. If you are going to be using .NET class names in your scripts, a best practice is to spell out the full class name.

# Leveraging .NET Methods

With .NET, a method is some action a .NET object occurrence, or the class can perform. These methods form the basis for many PowerShell cmdlets. For example, you can stop a Windows Process by using the Stop-Process cmdlet. The cmdlet then uses the Kill() method of the associated Process object. As a general best practice, you should use cmdlets and only dip down into using .NET classes and methods directly.

.NET methods can be beneficial to perform some operation which has no PowerShell cmdlet. One use of a .NET method is 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 objects Kill() method. Of course, to help IT Professionals, the PowerShell team created the Kill alias to the Stop-Process cmdlet.

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

At 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 methods encrypt and decrypt a file based on

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 Kill. PowerShell then calls the objects Kill() method. In practice, it’s 4 characters to type vs 11 (or is you are using tab completion to best effect 8), At the command line, piping to the kill method (where you don’t even have to use the open/close parentheses!) is just faster and risks fewer typos.

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

## Getting Ready

You run this recipe on SRV1 after loading PowerShell 7.1 and VS Code.

## How to do it...

1. Starting Notepad

notepad.exe

1. Obtaining methods on the Notepad process

$Notepad = Get-Process -Name Notepad

$Notepad | Get-Member -MemberType method

1. Using the Kill() method

$Notepad |

  ForEach-Object {$\_.Kill()}

1. Confirming Notepad process is destroyed

Get-Process -Name Notepad

1. 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"

}

1. Viewing files in $Path folder

$Files = Get-ChildItem -Path $Path

$Files | Format-Table -Property Name, Attributes

1. Encrypting the files

$Files| ForEach-Object Encrypt

1. Viewing file attributes

Get-ChildItem -Path $Path |

  Format-Table -Property Name, Attributes

1. Decrypting and viewing the files

$Files| ForEach-Object {

  $\_.Decrypt()

}

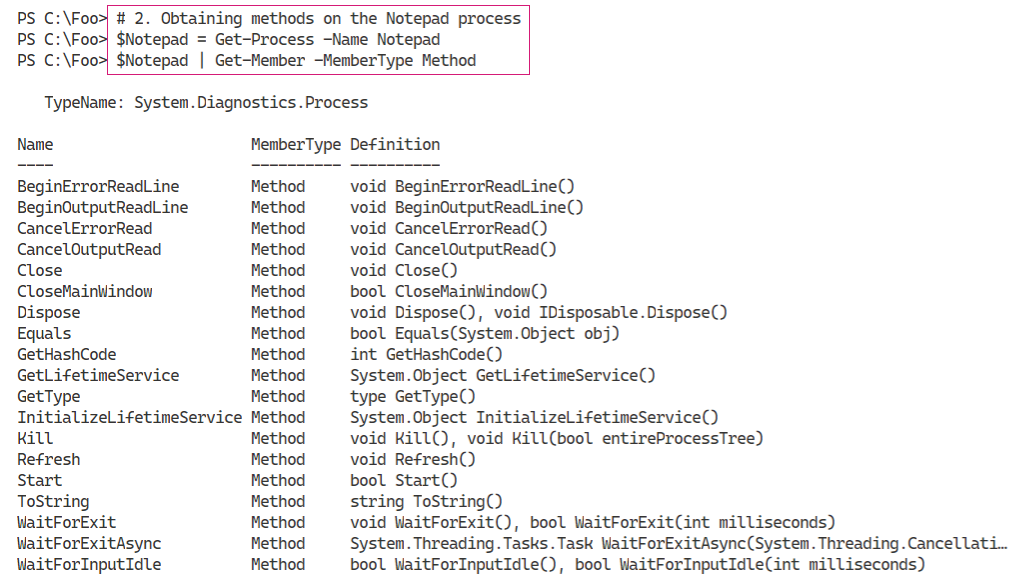
Get-ChildItem -Path $Path |

  Format-Table -Property Name, Attributes

## How it works...

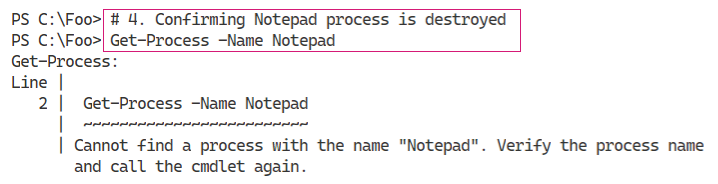
In step 1, which produces no output, you start Notepad.exe. This creates a process which you can examine and use.

In step 2, you obtain Notepad’s Process object and examine the methods available to you. The output looks like this:



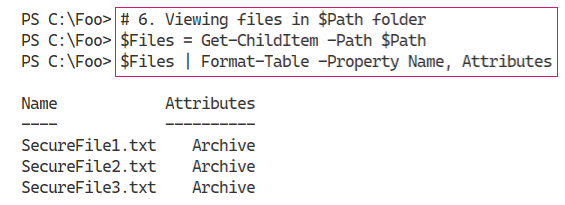
Insert image B42024\_05\_18.png

In step 3, you use the Kill() method in the System.Diagnostics.Process object to stop the Notepad process. This step produces no output. In step 4, you confirm that you have stopped the Notepad process with output like this:



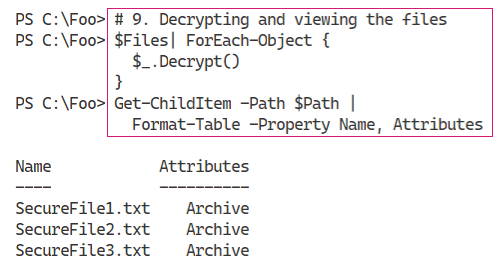
Insert image B42024\_05\_19.png

To illustrate other uses of .NET methods, you create a folder and three files within the folder in step 5. Creating these files generates no output. In step 6, you use the Get-ChildItem cmdlet to retrieve details about these three files, including all file attributes, which looks like this:



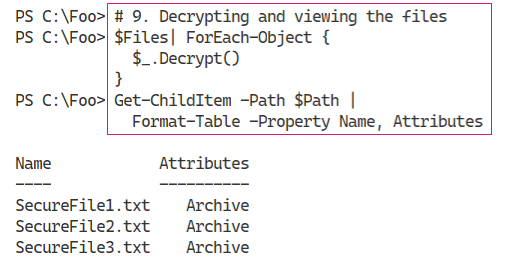
Insert image B42024\_05\_20.png

In step 7, you use the encrypt method to encrypt the files, generating no output. In step 8, you view again the attributes of the files, which looks like this:



Insert image B42024\_05\_21.png

Finally, with step 9, you decrypt the files using the Decrypt() method. You then use the Get‑ChildItem cmdlet to view the file attributes which allows you to determine that the files are not encrypted. The output of this step looks like this:



Insert image B42024\_05\_22.png

## There's more...

The output from step 2, you view the methods you can invoke on a process object. 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 slightly different output although with the same actual error message.

In step 7 and step 9, you use the FileInfo objects (created by Get-ChildItem) and call the Encrypt() and Decrypt() methods. These steps demonstrate using .NET methods to achieve some objective in the absence of specific cmdlets. Best practice suggests always using cmdlets where you can.

# 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. In some cases, as you saw in “Leveraging .NET Methods”, commands do not exist to achieve your goal. In those cases, you can make use of the methods provided by .NET.

There are also cases where you need to perform more complex operations without PowerShell cmdlet or direct .NET support. You may, for example, have a component of an ASP.NET web application you wish to repurpose for administrative 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 class that you can use in the same way you use .NET methods (and using virtually the same syntax). To do this, you use the Add-Type cmdlet and specify the C# code for your class/type(s). PowerShell compiles the code and loads the resultant class into your PowerShell session.

An essential aspect of .Net methods is that a method can have multiple definitions, or calling sequences. Multiple definitions allow you to invoke a method using different sets of parameters. This is not dissimilar to PowerShell’s use of parameter sets. For example, the System.String class, which PowerShell uses to hold strings, contains the Trim() method. You use that method to remove extra characters, usually space characters, from the start or end of a string (or both). The Trim() method has three different definitions, which you view in this recipe. Each overloaded definition trims characters from a string slightly differently. To view more details on this method, and the three overloaded definitions, see https://docs.microsoft.com/dotnet/api/system.string.trim?view=net-5.0/

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

## Getting Ready

You run this recipe on SRV1, a workgroup system running Windows Server Datacenter edition. You must have loaded PowerShell 7 and VS Code onto this host

## How to do it...

1. Examining the overloaded method definition

("a string").Trim

1. Creating a C# class definition in here string

$NewClass = @"

namespace Reskit {

   public class Hello {

     public static void World() {

         System.Console.WriteLine("Hello World!");

     }

   }

  }

"@

1. Adding the type into the current PowerShell session

Add-Type -TypeDefinition $NewClass

1. Examining method definition

[Reskit.Hello]::World

1. Using the class’s method

[Reskit.Hello]::World()

1. 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 + "!");

    }

  }

}

"@

1. Adding the type into the current PowerShell session

Add-Type -TypeDefinition $NewClass2 -Verbose

1. Viewing method definitions

[Reskit.Hello2]::World

1. Calling with no parameters specified

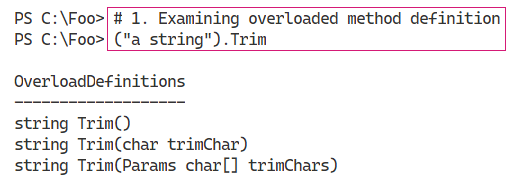
[Reskit.Hello2]::World()

1. Calling new method with a parameter

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

## How it works...

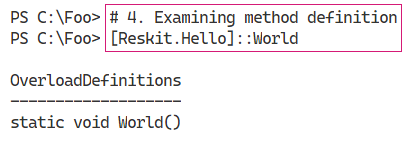
In step 1, you examine the different method definitions for the Trim() method. The output, showing the different overloaded definitions, looks like this:



Insert image B42024\_05\_23.png

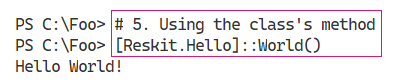
In step 2, you create a class definition and store the definition in a variable. In step 3, you add this class definition into your current PowerShell workspace. These two steps produce no output.

In step 4, you use the new method to observe the definitions available to you, which looks like this:



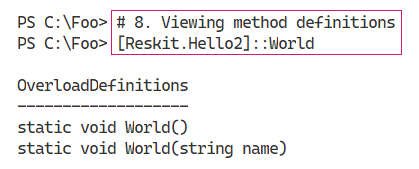
Insert image B42024\_05\_24.png

In step 5, you use the World() method, whose output is as follows:



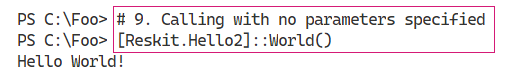
Insert image B42024\_05\_25.png

In step 6 and step 7, you create another new class definition, this time with two overloaded definitions to the World() method. These two steps produce no output. In step 8, you view the definitions for the World() method within the Reskit.Hell02 class, with output like this:



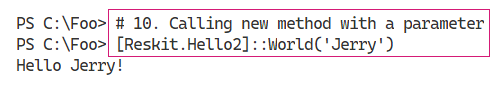
Insert image B42024\_05\_26.png

In step 9, you invoke the World() method, specifying no parameters, which produces this output:



Insert image B42024\_05\_27.png

In step 10, you invoke the World() method, specifying a single string parameter, which produces this output:



Insert image B42024\_05\_28.png

## There's more...

In step 1, you examine the different definitions of a method, in this case, the Trim() method of the System.String class. There are several ways you discover method overloads. In this step, you create a pseudo-object containing a string and then look at the method definitions. In this step, PowerShell creates an unnamed object (in the .NET Managed heap) which it immediately destroys after the statement completes. At some point later in time, .NET runs the GC process to reclaim the memory used by this temporary object and re-organises the managed heap.

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

In step 3, you 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 which then enabled you to use the classes. If you intend to use this add-in regularly, then you could add step 2 and step 3 into your PowerShell profile file or a production script. Alternatively, if you use this method within a script, you could add the steps into the script.

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

# Creating a PowerShell Cmdlet

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

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

As in the “Creating a C# Extension”, an important question to ask is why and when should you create a cmdlet? Aside from the perennial “because you can” excuse, there are reasons why an extension or a cmdlet might be a good idea. You can create a cmdlet to improve performance - for some operations a compiled cmdlet is just faster than doing operations using a PowerShell script. For some applications using A cmdlet is to perform actions that are difficult or not possible directly in PowerShell, including asynchronous operations and use of Language Independent Query (LINQ).

In this recipe, you load the SDK, and create a new PowerShell cmdlet. The installation of the .NET SDK requires you to navigate to a web page, download, and then run the SDK installer. There appears no simple way to automate this with PowerShell.

## Getting Ready

You run this recipe on SRV1, a workgroup system running Windows Server Datacenter edition. You must have loaded PowerShell 7 and VS Code onto this host

## How to do it...

*# 5.5 Create New Cmdlet*

1. Installing the .NET SDK. Open your browser and navigate to https://dotnet.microsoft.com/download/ and follow the GUI
2. Creating the cmdlet folder

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

Set-Location C:\Foo\Cmdlet

1. Creating a new class library project

dotnet new classlib --name SendGreeting

1. Viewing contents of new folder

Set-Location -Path .\SendGreeting

Get-ChildItem

1. Creating and displaying global.json

dotnet new globaljson

Get=Content .\global.json

1. Adding PowerShell package

dotnet add package PowerShellStandard.Library

1. Creating the cmdlet source file

$Cmdlet = @"

using System.Management.Automation;  // Windows PowerShell assembly.

namespace Reskit

{

  // Declare the class as a cmdlet

  // Specify verb and noun = Send-Greeting

  [Cmdlet(VerbsCommunications.Send, "Greeting")]

  public class SendGreetingCommand : PSCmdlet

  {

    // Declare the parameters for the cmdlet.

    [Parameter(Mandatory=true)]

    public string Name

    {

      get { return name; }

      set { name = value; }

    }

    private string name;

    // Override the ProcessRecord method to process the

    // supplied name and write a greeting to the user by

    // calling the WriteObject method.

    protected override void ProcessRecord()

    {

      WriteObject("Hello " + name + " - have a nice day!");

    }

  }

}

"@

$Cmdlet | Out-File .\SendGreetingCommand.cs

1. Removing unused class file

Remove-Item -Path .\Class1.cs

1. Building the cmdlet

dotnet build

1. Importing the DLL holding the cmdlet

$DLLPath = '.\bin\Debug\net5.0\SendGreeting.dll'

Import-Module -Name $DLLPath

1. Examining at Module

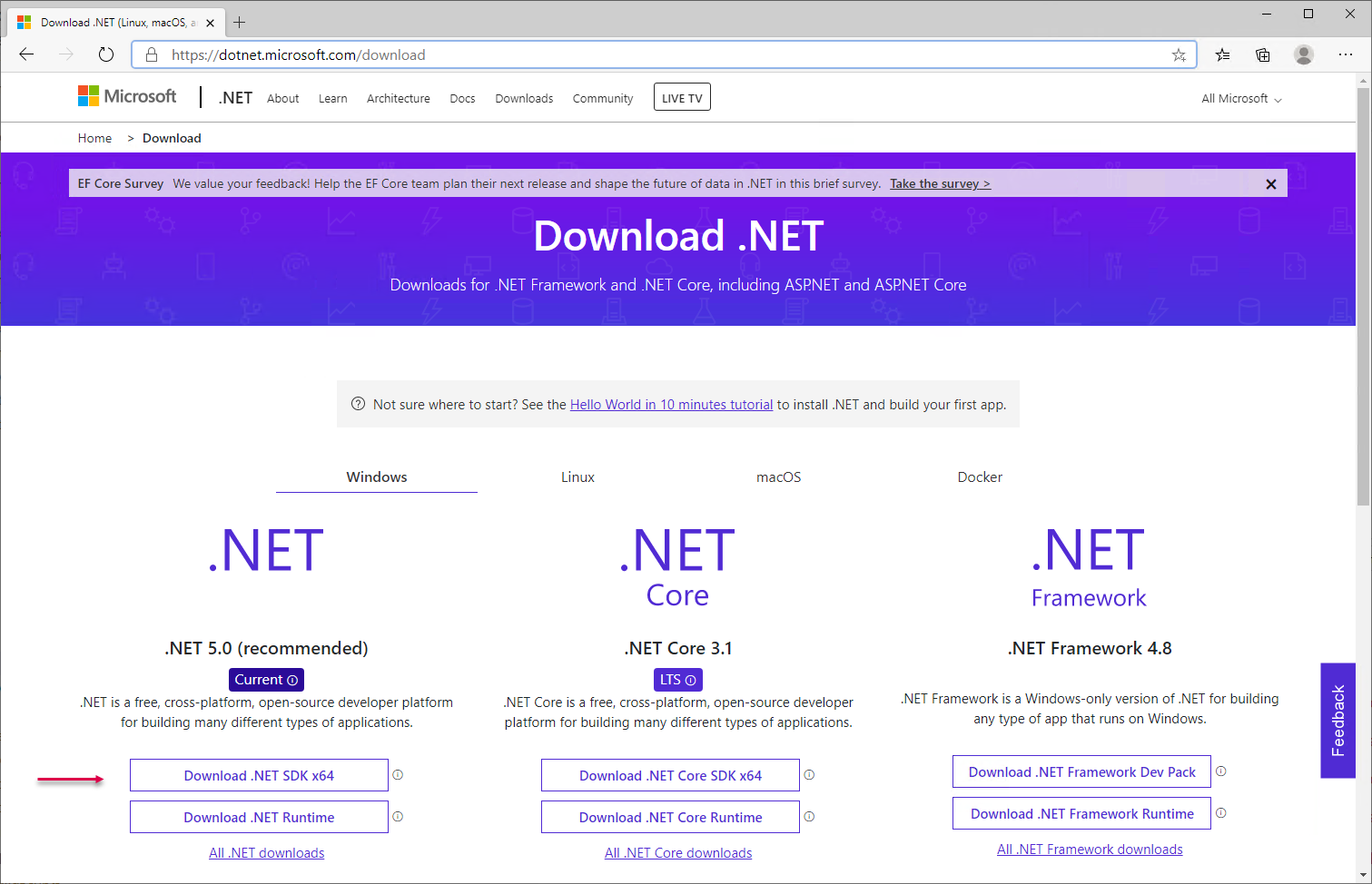
Get-Module SendGreeting

1. Using the cmdlet

Send-Greeting -Name "Jerry Garcia"

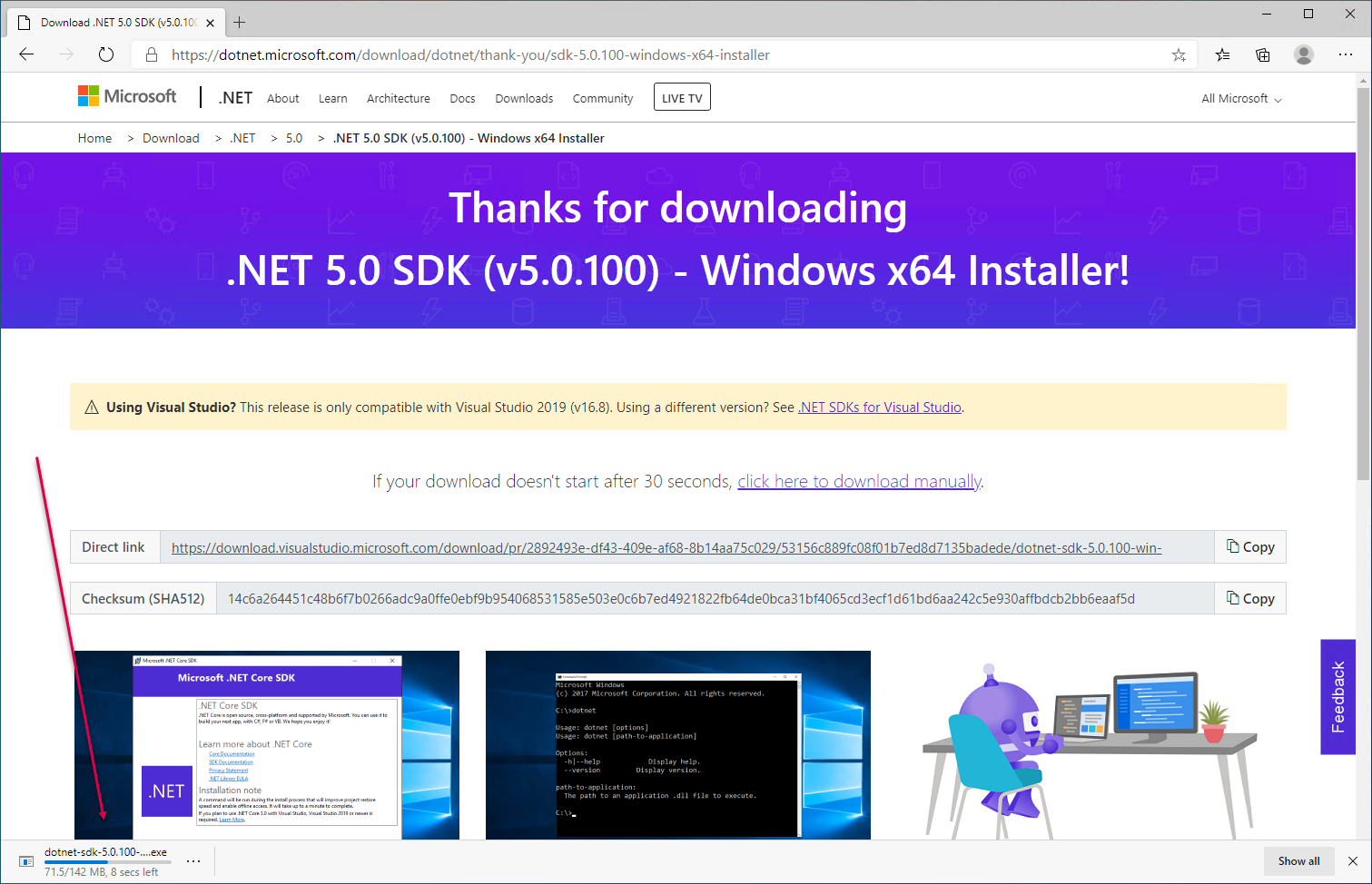
## How it works...

In step 1, you open your browser and navigate to the .NET Download page which looks like this:



Insert image B42024\_05\_29.png

Click, as shown, on the link to “Download the .NET SDK X64” to download the .NET SDK installer program, which looks like this:



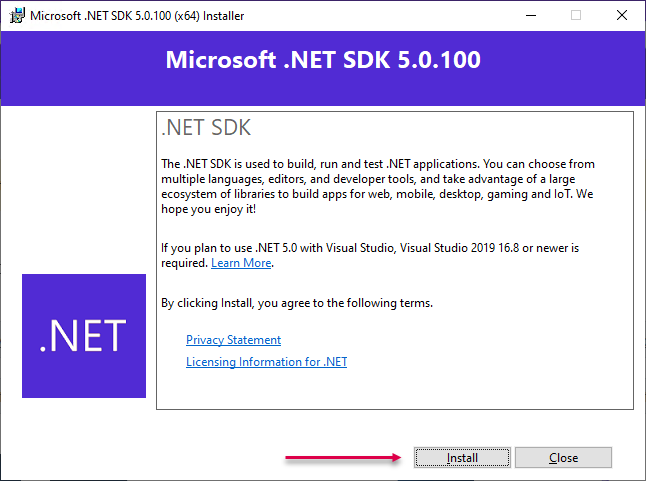
Insert image B42024\_05\_30.png

The installer programm is around 150mb so may take a while to download, depending on your internet connect5ion speed. When the download is complete, you see the download indication look like this:



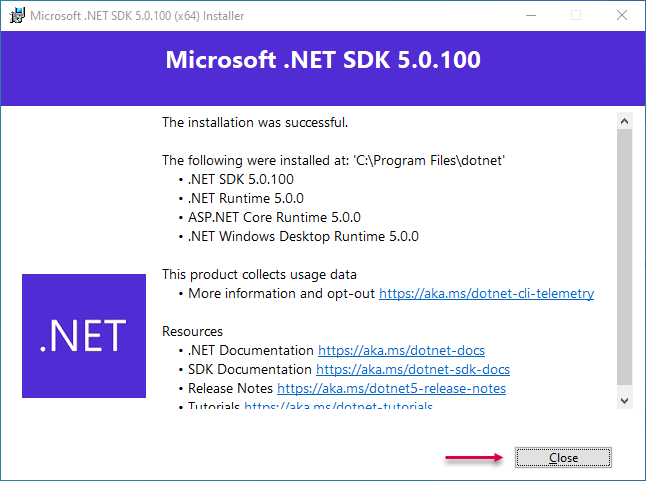
Insert image B42024\_05\_31.png

If you click on the Open file link, you begin the installer, which displays the installer’s opening page, which looks like this



Insert image B42024\_05\_32.png

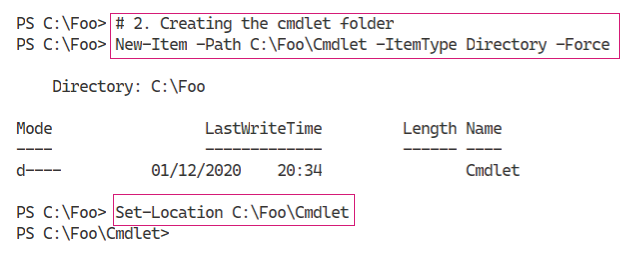
When you click on the Install button, as shown, the installer program completes the setup of the .NET SDK. When setup is complete, you see the final screen, like this:



Insert image B42024\_05\_33.png

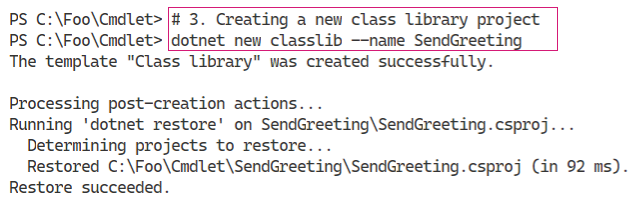
Click on Close to complete the setup.

In step 2, you create a new folder to hold your new cmdlet and use Set-Location to move into that folder. The output from this step looks like this:



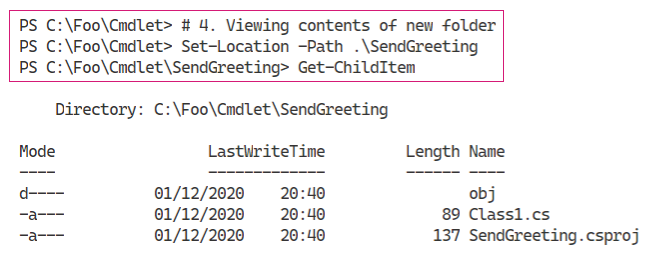
Insert image B42024\_05\_34.png

In step 3, you create a new class library project using the dotnet.exe command, which looks like this:



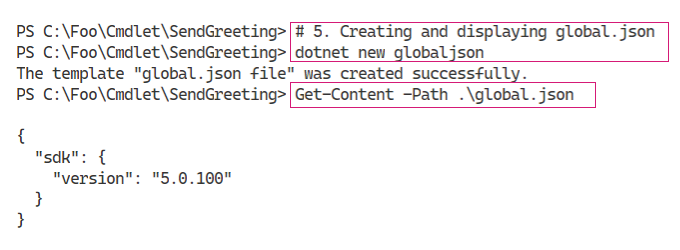
Insert image B42024\_05\_35.png

In step 4, you view the contents of the folder contents created by the previous step. The output looks like this:



Insert image B42024\_05\_36.png

In step 5, you create and then view a new global.json file. This JSON file, amongst other things, specifies which version of .NET is to be used to build your new cmdlet. The output of this step looks like this:



Insert image B42024\_05\_37.png

In step 6, you add the PowerShell package to your project. This step enables you to use the classes in the package to build your cmdlet. The output looks like this:

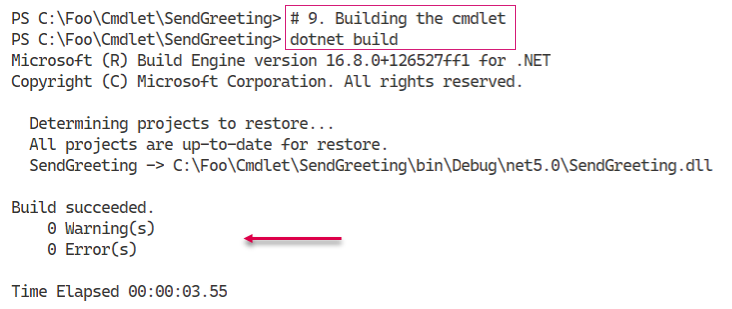


Insert image B42024\_05\_38.png

In step 7, you use PowerShell and a ‘here string’ to create the source code file for your new cmdlet. This step produces no output.

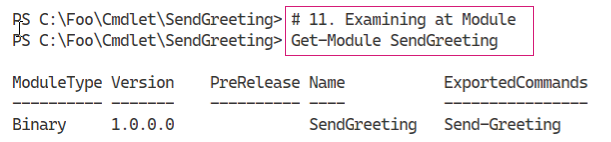
In step 8, you created your new project. This step also creates a file Class1.cs which is not needed for this project. In step 8, which generates no output, you remove this unneeded file.

In step 9, you build the cmdlet, which produces output like this:



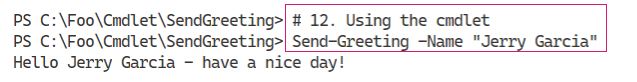
Insert image B42024\_05\_39.png

In step 10, you use Import-Module to import the DLL, which produces no output. With step 11, you use Get-Module to discover the imported module (containing just a single command), which looks like this:



Insert image B42024\_05\_40.png

In the final step in the recipe, step 12, you execute the cmdlet, which produces the following output:



Insert image B42024\_05\_40.png

## There's more...

In step 1, you download the .NET SDK, which you need in order to create a cmdlet. There appears no obvious way to automate the installation, so you need to run the installer and click through it’s GUI in order to install the .NET SDK.

In step 4, you create a new class library project. This step also creates the SendGreeting.csproj file as you can see. This file is a Visual Studio .NET C# project file that contains about the files included in your Send-Greeting cmdlet project, assemblies referenced from your code and more. For more details on the project file, see

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 Core loaded on your system. Using this file tells the project build process to use a specific version of .NET Core (such as version 5.0). For an overview to this file, https://docs.microsoft.com/dotnet/core/tools/global-json.

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

In step 11, you can see that the module, SendGreeting, is a binary module. A binary module is one just loaded directly from a DLL. This is fine for testing, but in production, you should create a manifest module by adding a manifest file. You should also move the module and 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 to an internal repository.

# Chapter Summary

In this chapter, you took a look at the .NET Framework as used by PowerShell 7. You look at the assemblies which contain the code implementing the classes within the .NET Base Class Libraries. You also looked at some classes and methods which can extend what you can do with PowerShell and to cater for situations where there are no cmdlets to support you. You also looked at creating C# extensions and implementing additional classes with methods. These extensions allow you to repurpose other existing C# (or other .NET Language) code. In the final recipe, you created a new PowerShell cmdlet.