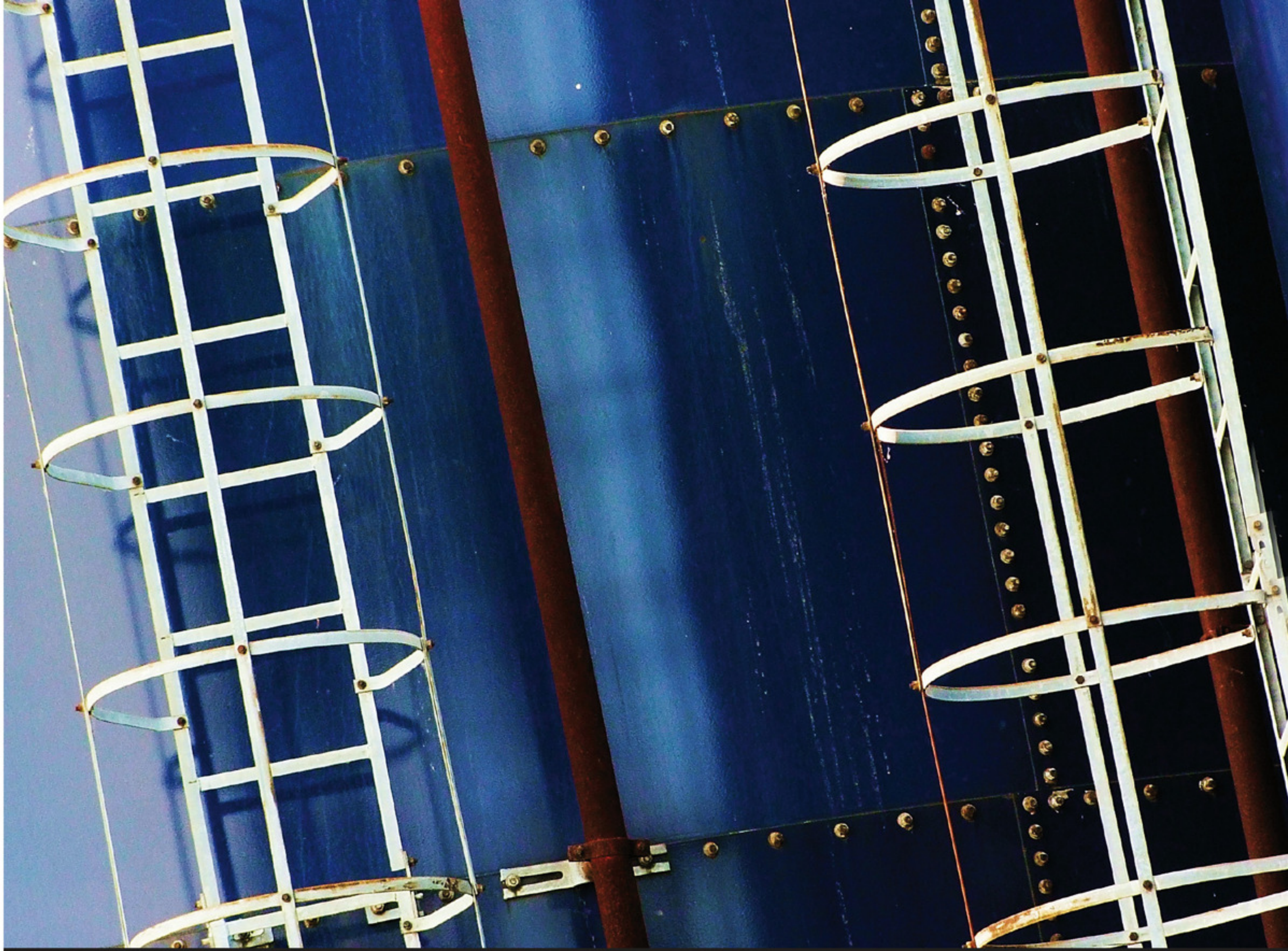


Professional Expertise Distilled

Microsoft Windows PowerShell 3.0 First Look

A quick, succinct guide to the new and exciting features in PowerShell 3.0

Adam Driscoll

[PACKT] PUBLISHING enterprise professional expertise distilled

Microsoft Windows PowerShell 3.0 First Look

A quick, succinct guide to the new and exciting features in PowerShell 3.0

Adam Driscoll

[PACKT] PUBLISHING enterprise professional expertise distilled

BIRMINGHAM - MUMBAI

Microsoft Windows PowerShell 3.0 First Look

First published: October 2012

Production Reference: 1260912

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84968-644-0

www.packtpub.com

Cover Image by Tina Negus (tina_manthorpe@sky.com)

Credits

Author
Adam Driscoll

Reviewers
Chris Dent
Shay Levy
David Karapetyan

Acquisition Editor
Dhwani Devater

Lead Technical Editor
Susmita Panda

Technical Editor
Prasanna Joglekar

Project Coordinator
Yashodhan Dere

Proofreader
Linda Morris

Indexers
Hemangini Bari
Tejal Daruwale

Graphics
Manu Joseph

Production Coordinator
Nilesh R. Mohite

Cover Work
Nilesh R. Mohite

About the Author

Adam Driscoll is a young and enthusiastic Software Developer and Team Lead at Quest Software. Born and raised in Wisconsin, Adam attended Edgewood College in Madison, WI and was hired shortly thereafter by Quest. He has experience in authoring PowerShell modules and providers in both .NET and PowerShell, building PowerShell development tools for developers and administrators and frequently blogs about .NET technologies.

I would like to thank my mother and father for always believing in me and supporting me. I wouldn't have been able to complete this without them. I would also like to thank my family and friends for providing encouragement throughout the process of writing this book. Finally, I would like to thank Carrie Arnold and Shay Levy for providing additional editorial guidance during the authoring of this book. Their help ensured that the quality of the book was far beyond what I could do alone.

About the Reviewers

Chris Dent is an Automation Specialist with over 10 years experience working with Microsoft and networking technologies. Chris has worked with PowerShell since 2007 specialising in network protocols and management.

My thanks should go to my wife Emily, for her endless patience in the face of incessant clattering from my keyboard in the evenings.

Shay Levy is a Systems Engineer working for a government institute in Israel. He has over 20 years of experience, focusing on Microsoft server platforms, especially on Exchange and Active Directory. For his contribution to the community he has been awarded with the Microsoft Most Valuable Professional (MVP) award for four years in a row. As a Microsoft Certified Trainer (MCT) he has taught numerous courses at John Bryce training center, Israel's largest computer training center. He is the Co-Founder and Editor of the `PowerShellMagazine.com` website and a Co-Director of the `PowerShellCommunity.org` website. Shay was also a technical reviewer of several books, including the *Microsoft Exchange 2010 PowerShell Cookbook* from Packt publishing. He often covers PowerShell related topics on his blog `http://PowerShay.com` and you can also follow him on Twitter at `http://twitter.com/ShayLevy`.

Not unlike many programmers **David Karapetyan** stumbled into programming while trying to attain a graduate degree in a related technical field. In his case it happened to be mathematics and while he enjoyed the abstractions and mental gymnastics involved in proving theorems at the end of the day opening up a shell and writing a Ruby script to demonstrate an edge case of some theorem was far more satisfying. So after attaining a Masters degree in mathematics he entered the field of software testing and reliability and hasn't looked back. These days his tools for quickly accomplishing a task are C# and PowerShell. He works at eSolar as a Software Test Engineer verifying the proper functionality of software that controls fields of heliostats at a concentrated solar power plant. The complicated nature of the software and its high reliability requirements mean that there must be a lot of automation in the verification and testing process in order to meet tight release deadlines. As the software runs on Windows the right tool for maintaining the testing and automation framework for verifying the software is PowerShell because it integrates seamlessly with all the components that comprise the testing framework.

Table of Contents

Preface

With PowerShell quickly becoming the de-facto standard for automation, on the Windows platform, it is becoming a necessity to learn and understand the language. *Microsoft Windows PowerShell 3.0 First Look* will ensure that you have a great overview of the numerous new features and changes found in the most recent version of the language. Through simple examples and succinct chapters, this book will quickly bring readers up to speed with need to know information about the newest version of PowerShell.

What this book covers

Chapter 1, *Find what you're looking for*, covers the new Show-Command cmdlet, updatable help and the module auto-discovery feature.

Chapter 2, *Usability Enhancements*, looks at the enhancements to the Where-Object, ForEach-Object, Get-ChildItem, and tab expansion.

Chapter 3, *Improved Administration*, covers improvements to administration including scheduled jobs, improved remote sessions, and an enhanced Restart-Computer cmdlet.

Chapter 4, *Windows Workflow in PowerShell*, examines Windows Workflow and how it has been integrated into Windows PowerShell.

Chapter 5, *Using the Common Informational Model*, looks at the Common Informational Model and the host of new cmdlets that are available to manage Windows and non-Windows devices.

Chapter 6, New and Improved PowerShell Hosts, looks at the new PowerShell Web Access and the greatly enhanced PowerShell Integrated Scripting Environment.

Chapter 7, Windows 8 and Windows Server 2012 Modules and Cmdlets, looks at some of the interesting new modules and cmdlets found in Windows 8 and Windows Server 2012.

What you need for this book

This book requires that you have the Windows Management Framework (WMF) version 3 installed. WMF version 3 requires that you are running Windows 7, Windows Server 2008, or Windows Server 2008 R2.

The final chapter in this book requires an installation of Windows 8 or Windows Server 2012. There are chapters within this book that show the use of Microsoft Visual Studio 2010 but this is not required to take full advantage of this book.

Who this book is for

This book is intended for IT administrators that are looking to quickly come up to speed on the new functionality found in PowerShell 3.0. Some PowerShell experience may be necessary to take full advantage of some of the topics covered in this book.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: The `Show-Command` is an interesting cmdlet to add to an until-now-command line-only shell.

Any command-line input or output is written as follows:

```
PS C:\> Show-Command New-Item
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: When clicking on the **Run** button, the command is executed and output is not displayed in the user interface, but rather written to the console.

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title through the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website, or added to any list of existing errata, under the Errata section of that title.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Find What You're Looking for

Windows PowerShell is a very dynamic language. There are many ways to improve the user experience such as adding new cmdlets (pronounced command-lets) that expose more functionality in the shell, or by adding new providers that allow users to traverse the data, just like one would traverse files and folders on a hard drive. The users can easily add functions to their profiles or load more complex modules that add a host of new features. As more and more vendors including Microsoft, start to pick up the pace of PowerShell support for their products, the more difficult it will be to find and understand all of the new components. Windows Server 2012 alone offers thousands of new cmdlets for interacting with the operating system. Luckily PowerShell features, such as the integrated help and the alias system, provide ways to understand and access cmdlets, and have been an integral part of the language since its inception. These features can help even the most seasoned PowerShell users understand and experiment with cmdlets for the first time.

The **cmdlets** are commands that can have data piped in and out of them. This allows multiple cmdlets to be chained together to form a pipeline of processing, simplifying scripts, and making them read more like English language. As an example, we can select processes based on their memory usage and output their names with a simple pipeline of the cmdlet. The following command explains this:

```
PS C:\> Get-Process | Where-Object WorkingSet -gt 100MB | Select-Object
Name

Name
----
chrome
svchost
```

Simplifying the way users find and use cmdlets has been a major corner stone of PowerShell from the beginning, but there have been certain areas where improvement is necessary. The three areas that Microsoft focused on during the most recent version of PowerShell, have been the user avoidance of the command line, the contextual help system, and cmdlet discovery.

Often users are accustomed to a **Graphical User Interface** (**GUI)** because it offers contextual clues, steps, and more natural feedback. The command line can seem primitive and daunting. To help prevent some of this shell-shock, Microsoft implemented a transitional cmdlet, `Show-Command`, that displays the required parameters and help in a GUI rather than on the command line.

The second area that Microsoft has focused on is the cmdlet discovery. In PowerShell 2.0, modules were introduced. The modules allowed developers to create simple and pluggable bunches of related cmdlets and providers into a single unit that could be loaded and unloaded at will. These modules are starting to become the standard that Microsoft itself has used for developing new product or component based PowerShell support. Although not all their PowerShell support ships as modules, a large majority does. The problem was that, modules needed to be loaded explicitly before any of their functionality could be used. As more and more modules are implemented and installed, it becomes increasingly difficult to find the proper module and cmdlet. Additionally, it requires two commands instead of one: one to load the module and one to call the target cmdlet.

The final area is the contextual help system, which was plagued by the fact that it was static. It only allowed users and vendors to update help when everything else was updated. This was a headache for both parties.

In this chapter we will see the following:

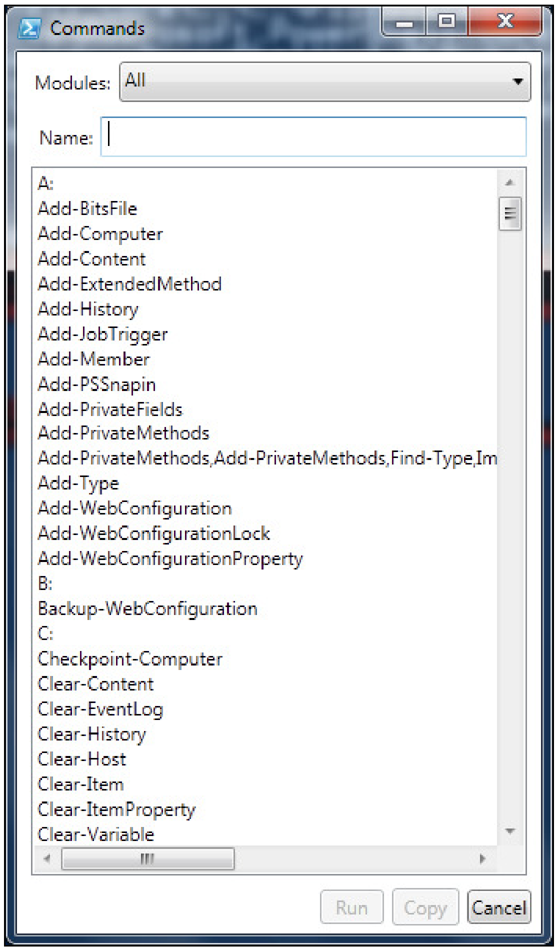
- The new `Show-Command` cmdlet and how it transitions into PowerShell a bit easier
- The updatable contextual help system which ensures that we are up to date with the latest versions of contextual help
- The cmdlet and module auto-discovery system, which makes working with all the cmdlets and modules in a system easier

The Show-Command cmdlet

Often the command line can seem like an unfamiliar and daunting area of the operating system, which new users are reluctant to investigate. The lack of a user interface which can be clicked on often outweighs the benefits that short, quick commands can offer. In PowerShell 3.0, Microsoft has added a simple cmdlet to aid in eliminating this stumbling block. This cmdlet is `Show-Command`.

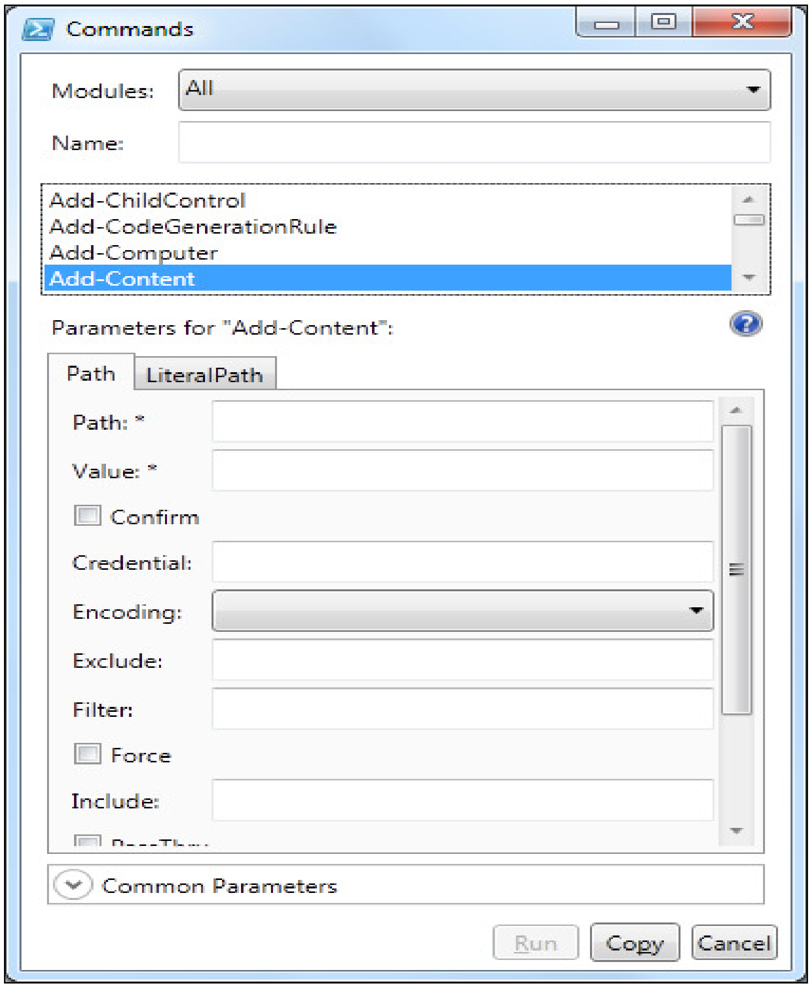
Running `Show-Command` without any parameters opens a user interface with a drop-down list of available modules and a list box full of all the cmdlet, aliases, drives, and functions available in those modules. The commands in the list will depend on what is installed on the machine and the list will differ per machine. This is done as follows:

```
PS C:\> Show-Command
```



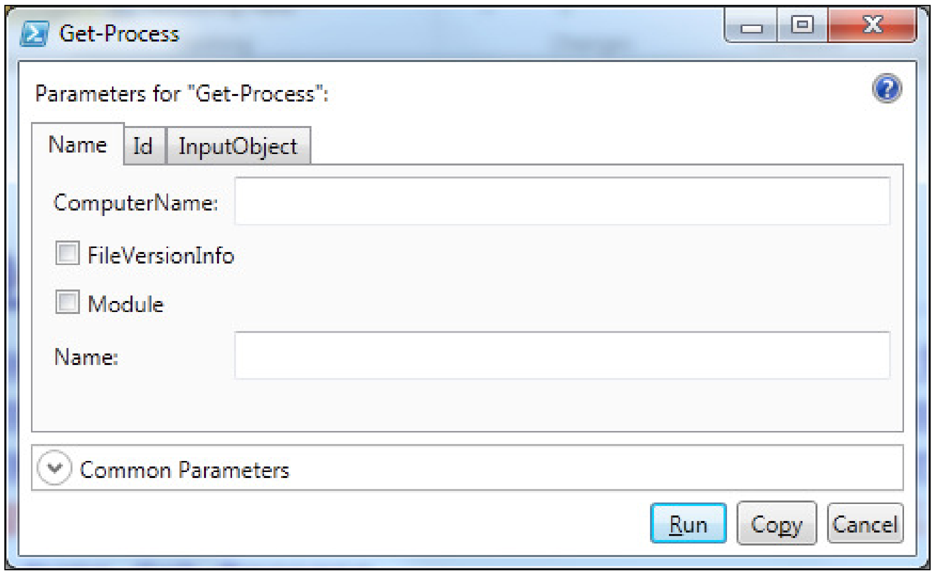
A:
Add-BitsFile
Add-Computer
Add-Content
Add-ExtendedMethod
Add-History
Add-JobTrigger
Add-Member
Add-PSSnapin
Add-PrivateFields
Add-PrivateMethods
Add-PrivateMethods,Add-PrivateMethods,Find-Type,Im
Add-Type
Add-WebConfiguration
Add-WebConfigurationLock
Add-WebConfigurationProperty
B:
Backup-WebConfiguration
C:
Checkpoint-Computer
Clear-Content
Clear-EventLog
Clear-History
Clear-Host
Clear-Item
Clear-ItemProperty
Clear-Variable

There is a textbox for searching the list available under the **Modules** drop-down list. If you select one of the commands, drives, or functions, a new pane will open at the bottom of the window, as shown in the following screenshot, which shows all the parameters for the cmdlet and allows you to run, copy, or view the help for a particular item:



You can pass the name of a cmdlet to the `Show-Command` cmdlet through the **Name** parameter as well. In this mode, the `Show-Command` user interface allows for input into the named cmdlet. All the parameters and parameter sets are available for the user to select from and provide input. For this, you can enter the following command:

```
PS C:\> Show-Command -Name Get-Process
```

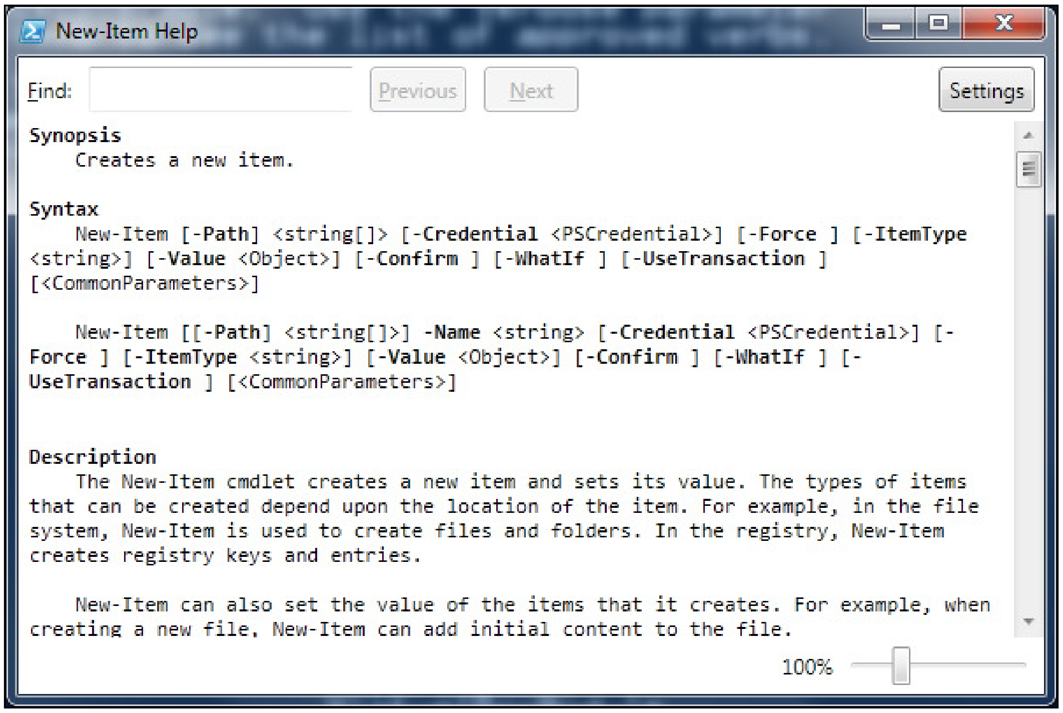


By clicking on the **Copy** button, a string of text is copied to the clipboard, which is the result of the data entered into the user interface. This text could then be pasted into a script editor or the command line for execution. When clicking on the **Run** button, the command is executed and the output is not displayed in the user interface, but rather written to the console. Additionally, the size of the resulting window can be adjusted using the `Height` and `Width` parameters of the cmdlet.

Another interesting feature of the `Show-Command` cmdlet is the **?** button (shown in the previous screenshot). This button is only available when the `Name` parameter is specified or a cmdlet is selected from the first view shown previously. Clicking on the button will open a new window with the help for the current command. This is explained by the following command:

```
PS C:\> Show-Command New-Item
```

The output is shown in the following screenshot:



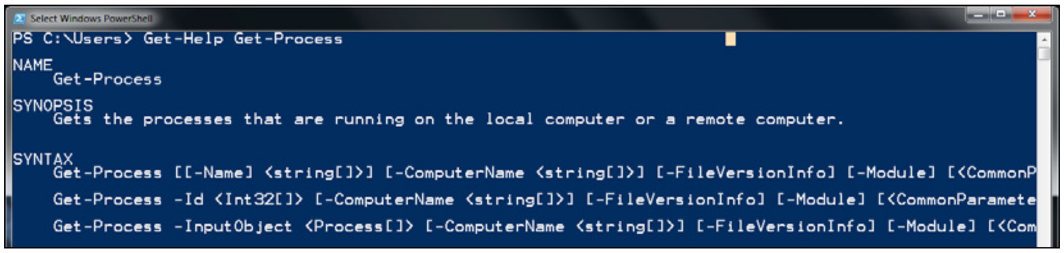
```
Synopsis
    Creates a new item.

Syntax
    New-Item [-Path] <string[]> [-Credential <PSCredential>] [-Force ] [-ItemType
<string>] [-Value <Object>] [-Confirm ] [-WhatIf ] [-UseTransaction ]
[<CommonParameters>]

    New-Item [[-Path] <string[]>] -Name <string> [-Credential <PSCredential>] [-
Force ] [-ItemType <string>] [-Value <Object>] [-Confirm ] [-WhatIf ] [-
UseTransaction ] [<CommonParameters>]


Description
    The New-Item cmdlet creates a new item and sets its value. The types of items
that can be created depend upon the location of the item. For example, in the file
system, New-Item is used to create files and folders. In the registry, New-Item
creates registry keys and entries.

    New-Item can also set the value of the items that it creates. For example, when
creating a new file, New-Item can add initial content to the file.
```

Note that the cmdlet `Show-Command` is blocking the view and you will not be able to interact with the PowerShell console while the window is open.

The `Show-Command` is an interesting cmdlet to add to an until-now-command line-only shell. The benefit is that it offers a simple visual view of many cmdlets and the respective help. This can help users overcome the shell-shock, which is often associated with opening PowerShell for the first time. Additionally, the cmdlets can be of value to administrators looking to shortcut some of the unnecessary typing that must be done for the cmdlets with lots of parameters. Rather than having to type them out, you can now find yourself tabbing through the `Show-Command` user interface.

Updatable contextual help

For a first-time user to the language, `Get-Help` is the most important cmdlet in the bunch (some may argue this is true for all users!). Not only does it offer the syntax definitions, but also examples, detailed explanations, and links to online resources as well. This is shown in the following screenshot:



```
PS C:\Users> Get-Help Get-Process

NAME
    Get-Process

SYNOPSIS
    Gets the processes that are running on the local computer or a remote computer.

SYNTAX
    Get-Process [[-Name] <string[]>] [-ComputerName <string[]>] [-FileVersionInfo] [-Module] [<CommonP
    Get-Process -Id <Int32[]> [-ComputerName <string[]>] [-FileVersionInfo] [-Module] [<CommonParamete
    Get-Process -InputObject <Process[]> [-ComputerName <string[]>] [-FileVersionInfo] [-Module] [<Com
```

In the previous versions of the language, the PowerShell help was a static resource that would only be updated during major releases. For example, in PowerShell 2.0, links integrated into the help would point users to the Internet for additional, possibly updated resources. This required using the `Online` parameter of the `Get-Help` cmdlet, as follows:

```
PS C:\> Get-Help Get-Process -Online
```

The use of the `Online` parameter was cumbersome as it drove the user outside of the shell and into the default browser. Additionally, any documentation errors could not be corrected in the local help and could lead users astray. Microsoft has improved the user experience of the help system in the latest version of PowerShell. Rather than making it a static resource, the help system has become a dynamic feature, much like the rest of PowerShell. This allows users to retrieve the most updated help directly from Microsoft and other developers to ensure the best and the most reliable information.

What help is made of

To understand how to update the help, it makes sense to understand what the help is made out of. The help system is comprised of a collection of text- and XML-like files stored in the PowerShell installation directory or within module directories themselves. Long-winded topic discussions are stored simply in `about_*.txt` files within the PowerShell installation directory. These files are retrieved using `Get-Help` and specifying the full name or a distinguishable part of the title of the help topics, using the following command:

```
PS C:\> Get-Help -Name about_Modules
```

The same command would work by just specifying modules as the `Name` parameter as follows:

```
PS C:\> Get-Help -Name modules
```

Help that is directly related to cmdlet is stored within a unique format called **Microsoft Assistance Markup Language** (**MAML)**. The entirety of cmdlet help is formatted into these files; including descriptions, examples, and detailed parameter explanations. The MAML help files are stored within the module directories themselves. The help system parses the different pieces of the MAML document, as you request them using the `Get-Help` cmdlet.

In the latest version of PowerShell, the help system can now download and install help from the Internet or from the file system. The improved help system now takes advantage of a new key that is a part of the module's manifest. This key, `HelpInfoURI`, points to the location of the updated help files. Module authors can specify this URI by using the `New-ModuleManifest` cmdlet or by editing the `PSD1` manifest file for a module.

There are two new cmdlets that have been added to the help system. The `Save-Help` cmdlet is capable of downloading help files from the Internet and storing them packaged as a combination of XML and cabinet (CAB) files. These files can then be provided to `Update-Help` for installation. `Update-Help` is capable of both downloading help files from the Internet and installing them. It can consume the files downloaded by `Save-Help` as well and install them.

The Save-Help cmdlet

The `Save-Help` cmdlet downloads the help for modules and saves the modules to the local file system or share. As the cmdlet saves the help to the `$PSHome\Modules` directory, if an alternate path is not specified, you will need to be part of the administrators' group and run an elevated PowerShell command prompt by right-clicking and selecting **Run As Administrator**. The `Save-Help` cmdlet uses a `HelpInfo` XML file of a module to gather the correct URL for downloading the help files. When the help files are downloaded, they will be organized into a single `HelpInfo` XML file accompanied by several CAB files, one for each language. The CAB files contain the actual help documentation. You don't have to worry about extracting any of the data from the CAB file as the `Update-Help` cmdlet will do this for you.

The Save-Help cmdlet can specify an alternate path using the LiteralPath or DestinationPath parameters. The LiteralPath parameter will not expand wildcards. Note that the path will not be created by this cmdlet. The following command shows this:

```
PS C:\> Save-Help -DestinationPath C:\MyHelpDirectory
```

You can download help for a single module as well. You cannot specify the help for a single cmdlet. To download help for a single module use the Module parameter as follows:

```
PS C:\> Save-Help -Module Microsoft.PowerShell.Core -DestinationPath
C:\MyHelpDirectory
```

There is a restriction enforced by the Save-Help cmdlet which prevents you from downloading the help more than once in a 24-hour period. To bypass this you can use the Force parameter, as shown in the following command:

```
PS C:\> Save-Help -Module Microsoft.PowerShell.Core -Force -
DestinationPath C:\MyHelpDirectory
```

The Update-Help cmdlet

The Update-Help cmdlet is responsible for installing help downloaded from the Internet or local folder or file share. By default, the Update-Help cmdlet will download files from the Internet and install them if they are out of date. If you run Update-Help without parameters it will update help for each module loaded into the current session. You will need to be part of the administrators' group and run as administrator, to update any of the core PowerShell modules, because they are stored in the system directory. This will update all currently loaded modules directly from the Internet if they are out of date. The following command can be used for this:

```
PS C:\> Update-Help
```

Alternatively, you can update help from a local folder or file share. The Update-Help cmdlet expects that these files are in the format that Save-Help creates. You will need to have both the HelpInfo XML file along with any CAB files in the specified folder or share. To invoke Update-Help in this way, you can use the LiteralPath or SourcePath parameters as shown in the following command:

```
PS C:\> Update-Help -SourcePath C:\MyHelpDirectory
```

Like the Save-Help cmdlet, the Update-Help cmdlet, by default, only allows you to download the help once per day. To bypass this you can use the Force parameter as follows:

```
PS C:\> Update-Help -Module Microsoft.PowerShell.Core -Force
```

Again, like the `Save-Help` cmdlet, the `Update-Help` cmdlet allows you to specify a particular module and culture when updating help. By default, the cmdlet will use the current culture. In this example, it would download and install the help for the `Microsoft.PowerShell.Core` module for the German culture, as follows:

```
PS C:\> Update-Help -Module Microsoft.PowerShell.Core -UICulture de-DE
```

The cmdlet offers a `Recurse` parameter that will search the `SourcePath` directory, recursively, for `HelpInfo` XML and CAB files using the following commands:

```
PS C:\> Update-Help -SourcePath \\fileServer01\HelpFiles -Recurse
```

In addition to accepting strings for module names through the `Module` parameter, the cmdlet `Update-Help` and cmdlet `Save-Help` will accept `PSModuleInfo` objects through the `InputObject` parameter as well. This parameter accepts pipeline input. The `Get-Module` cmdlet returns `PSModuleInfo` objects. This allows you to use the two modules in unison and create a pipeline between the two. For example, you could get all the modules that are available and update them. This is shown in the following command:

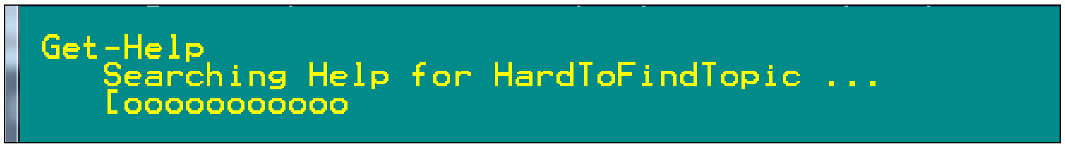
```
PS C:\> Get-Module -ListAvailable | Update-Help
```

In the same way, with `Save-Help`, you could save the help for all modules in a particular directory, as follows:

```
PS C:\> Get-Module -ListAvailable | Save-Help -DestinationPath
C:\MyHelpDirectory
```

The Get-Help cmdlet

The `Get-Help` cmdlet has been updated with a simple usability feature. Microsoft added a progress indicator that is now present when searching for help on a particular cmdlet. Because the number of cmdlets and modules is growing exponentially, and the cmdlet discovery is expanding beyond the session (as you'll soon see), this subtle but necessary feature has been added, as shown in the following screenshot:



```
Get-Help
   Searching Help for HardToFindTopic ...
   [ooooooooooo
```

The auto-discovery cmdlet

Finding cmdlets and functions in PowerShell is very easy. You can simply type `Get-Command` to get a list of all the cmdlets and functions available in the current session. Adding new functions or cmdlets to the session can be done by running scripts, or even writing your own as you go.

In PowerShell 2.0, the concept of a module was introduced. A module is a packaged set of cmdlets, aliases, functions, types and other components that can be loaded and unloaded from a session. Often these modules serve a single purpose, for example `Hyper-V` support or `Diagnostics`. Keep in mind that roles or features may need to be enabled before accessing particular modules. The `Hyper-V` module for example is only available if the `Hyper-V` role is enabled. To access modules in the current session you can simply type `Get-Module` as follows:

```
PS C:\> Get-Module


ModuleType Name                        ExportedCommands
---------- ----                        ----------------
Manifest   Microsoft.PowerShell.Core {Add-History, Add-...
Manifest   Microsoft.PowerShell.M... {Add-Computer, Add-...
Manifest   Microsoft.PowerShell.S... {ConvertFrom-...
Manifest   Microsoft.PowerShell.U... {Add-Member, Add-Type,..
```

To take the command a step further you can issue it with the `ListAvailable` parameter too. This will show all modules that are currently installed in the specified modules directories within the system, as shown in the following command:

```
PS C:\> Get-Module -ListAvailable
```

The `Get-Module` cmdlet is looking for folders within the directories that contain modules manifests. Module manifests outline what is contained in and exported from the module. Notice the previous output the `ExportedCommands` property of each record returned by the `Get-Module` cmdlet. This property shows which commands are to be exported from the module and thus available to the user in the session. Module authors are advised to set the `CmdletsToExport` key in their module manifests. This ensures that the module will not actually have to be loaded in order to view which cmdlets are provided by the module. If this key is not populated, the module may be loaded in order to discover the cmdlets that are exported.

By default, the module directories include the user module path, `%userprofile%\Documents\WindowsPowerShell\Modules`, and the system module path, `%windir%\System32\WindowsPowerShell\v1.0\Modules`. These values are stored within the `$env:PSModulePath` environment variable.

In the previous versions of PowerShell, you would be responsible for knowing which module contained the cmdlets you were attempting to use. You would then have to call the `Import-Module` cmdlet followed by the name of the module. This would make the cmdlet available to you in the command line. This is no longer the case in PowerShell 3.0.

In the newest version of PowerShell, the cmdlet `auto-discovery` cmdlet does that work for you. Simply typing the name of a cmdlet which is available in any module, imported or not, will execute as expected. For example, the `WebAdministration` module contains a `Get-Website` cmdlet. Note that the web server role needs to be enabled to gain access to the `WebAdministration` module.

In PowerShell 2.0, before being able to use `Get-Website` you would first have to import the `WebAdministration` module, using the following commands:

```
PS C:\>Import-Module WebAdministration
PS C:\> Set-Location IIS:
PS ISS:\>Get-Website –Name "Default Web Site"
```

There are two pain points associated with the previous interaction. First, the user must know which module contains the cmdlet they are intending to use. Finding the cmdlet could be done by calling `Get-Module` and then parsing the `ExportedCmdlets` parameter, as follows:

```
PS C:\> Get-Module -ListAvailable | Where-Object { $_.ExportedCommands.
Keys -eq "Get-Website" }
```

The command returns nothing in this case because the `WebAdministration` module does not define the `ExportedCmdlets` key in the module manifest. Instead it specifies the wildcard character, `*`, signifying that all cmdlets are exported. Due to this, we would need to import every module in order to search for the cmdlet. This could be very time consuming and resource intensive if a system had a lot of modules installed.

The second pain point is the need to make sure a module is currently imported into a session before calling any cmdlets. This can be a problem when multiple scripts are chained together. For example, if `Script1.ps1` was run first, and it imported module `x`, and then `Script2.ps1` used cmdlets from module `x` it would work correctly. If `Script2.ps1` was run by itself it would fail because it did not import module `x`.

Understanding cmdlet auto-discovery

The `auto-discovery` cmdlet removes the need to load a module explicitly before calling cmdlets that it includes. For example, rather than performing the previous imports you could simply call it `Get-Website` as follows:

```
PS C:\>Get-Website
```

Right before execution of the `Get-Website` cmdlet, PowerShell imported the module for us. This is not the only circumstance where a cmdlet module will be automatically loaded. There are actually two other actions that will implicitly load a module. The first action is when attempting to load the help for a cmdlet using `Get-Help` as follows:

```
PS C:\> Get-Help Get-Website
```

Again this will load the module right before getting the help for the cmdlet. This is necessary because the module contains the help information, thus `Get-Help` requires that the module be loaded before reading the help. The last circumstance where a module will be implicitly loaded is when you use the `Get-Command` cmdlet to retrieve a `CmdletInfo` object using following command:

```
PS C:\> Get-Command Get-Website
```

Using a `Get-Command` call with a wildcard character will not load the module into the current session. The `about_Modules` help topic explains why. The following sentence supports this, which can be found at

`http://msdn.microsoft.com/en-us/library/windows/desktop/hh847804.aspx`.

Get-Command commands that include a wildcard character (`*`) are considered to be discovery, not use, and do not import any modules.

It is possible to modify how modules are auto-loaded in your PowerShell session. You can adjust the value of the `PSModuleAutoLoadingPreference` variable. The default value for this variable is `All`. `All` signifies that during any of the these actions the module containing the target cmdlet will be auto-loaded. The other two values are `ModuleQualified` and `None`. `ModuleQualified` loads modules only when a command is prefixed by the module name, as follows:

```
PS C:\> Get-Command "MyModule\MyCommand"
```

In any other circumstance the module will not be loaded. Setting `PSModuleAutoLoadingPreference` to `None` will turn off module auto-loading.

There is one other change that affects when modules are imported. In PowerShell 2.0, modules that relied on other modules, could specify the `RequiredModules` property and list the other modules that were required by the module. If the other modules were not already imported when the target module was being imported, an error would be shown and the module would not be imported successfully. This has changed in PowerShell 3.0. Instead of the primary module load failing immediately if the other modules are not loaded, an attempt to load all the dependent modules will be made before loading the target module.

This effectively alleviates the two pain points seen in the previous versions of PowerShell. The cmdlets can be used without having to load their modules. This reduces script dependencies and saves a bit of typing. In addition, it enables administrators to search for cmdlets and the help of cmdlets without having to know the details of where they are stored.

Things to know about auto-discovery

The cmdlet `auto-discovery` is a great new feature but has a few caveats that need to be understood when working with the new version of PowerShell. The first thing to note is that the help system searches the module directories in a specific order. The local user's module directory is the first to be searched, followed by the system module directory. This is very important to understand because PowerShell allows for functions to be overwritten. This means that it is possible to have two separate modules that contain those cmdlets. For example, if you have two virtual machine management modules, say `HyperVModule` and `VirtualBoxModule`, and they both have `Get-VM` cmdlet.

Here is an example of one of the module manifests:

```
#HyperVModule.psd1 - Module Manifest

@{
ModuleVersion = '1.0'
GUID = '256c636c-7e77-493c-971e-fe16cfe0601c'
Author = 'Adam Driscoll'
CompanyName = 'Get-VM'
Copyright = '2011 Adam Driscoll'
Description = 'Test Hyper-V Module'
PowerShellVersion = '3.0'
#Some fields removed …
NestedModules = @('HyperVModule.ps1')
CmdletsToExport = 'Get-VM'
}
```

As you can see the module will export the cmdlet `Get-VM`. The `CmdletsToExport` property is what the system is evaluating to determine where cmdlets are stored during `auto-discovery`. Here is an example of one of the source files:

```
function Get-VM
{
    Write-Host "HyperVModule:Get-VM"
}
```

As you can see, we are a defining a function, `Get-VM`, which simply prints out the name of the module and the name of the cmdlet. In the case of the `VirtualBoxModule`, we have defined a very similar structure, except it will print out `VirtualBoxModule:Get-VM` rather than the string presented in the previous script snippet.

Imagine that the `HyperVModule` is stored in the local user module directory while `VirtualBoxModule` is stored in the system module directory. Typing `Get-VM` in the shell will then load the `HyperVModule` and execute that cmdlet.

Executing the `Get-VM` on the command line would now result in the following:

```
PS C:\> Get-VM
HyperVModule:Get-VM
```

To execute the other version of the cmdlet you will need to first load the `VirtualBoxModule` explicitly, and then execute the cmdlet as follows:

```
PS C:\> Import-Module VirtualBoxModule
PS C:\> Get-VM
VirtualBoxModule:Get-VM
```

The cmdlet `auto-discovery` is great when attempting to locate a particular cmdlet, but as seen previously it behaves differently when there are multiple cmdlets with one name.

Although this is an issue there are several ways to work around it. Even when both the cmdlets are loaded, it is still possible to fully qualify the name of the cmdlet we would like to execute. For example, if we wanted to execute the `Get-VM` cmdlet of `VirtualBoxModule`, we could do the following:

```
PS C:\> VirtualBoxModule\Get-VM
VirtualBoxModule:Get-VM
```

In addition to using the default fully qualified name, we could use the `Prefix` parameter of the `Import-Module` cmdlet as well. This allows us to change the prefix for the cmdlet. For example, we could define a new prefix for the `VirtualBoxModule` as follows:

```
PS C:\> Import-Module VirtualBoxModule –Prefix VBM
PS C:\> VBM\Get-VM
VirtualBoxModule:Get-VM
```

> In PowerShell 2.0, this prefix was defined by the module name. But in PowerShell 3.0 module authors have the option of changing the prefix by specifying the `DefaultCommandPrefix` key in the manifest of the module.

Another problem associated with this circumstance is attempting to find these cmdlets. `Get-Command` is not set up to find cmdlets beyond the first one it finds when there are no wildcards associated with the name. Once it locates the cmdlet it is looking for, it will simply stop looking and return that cmdlet. So executing the following will only return the `HyperVModule` version of `Get-VM` because it looks in the `user module` directory before looking in the `system module` directory:

```
PS C:\ Get-Command Get-VM

CommandType     Name                            ModuleName
-----------     ----                            ----------
Function        Get-VM                          HyperVModule

PS C:\ > Get-Module

ModuleType Name                         ExportedCommands
---------- ----                         ----------------
Manifest   HyperVModule                 Get-VM
```

Notice that it has loaded the module after calling `Get-Command`. Although in most circumstances the `Get-Command` works in our favor, this time we were attempting to search for all the `Get-VM` cmdlets available to us. As you can see, the `HyperVModule` is now loaded into our session. In order to find the other cmdlet we need to get a bit more complex in the way we search the system. The following command will retrieve all the available modules and filter through each one of their exported commands, returning only modules that contain a `Get-VM` cmdlet:

```
PS C:\ > Get-Module -ListAvailable | Where-Object { $_.ExportedCommands.
ContainsKey('Get-VM') }
```

```
    Directory: C:\Users\Adam\Documents\WindowsPowerShell\Modules


ModuleType Name                          ExportedCommands
---------- ----                          ----------------
Manifest   HyperVModule                  Get-VM


    Directory: C:\Windows\system32\WindowsPowerShell\v1.0\Modules


ModuleType Name                          ExportedCommands
---------- ----                          ----------------
Manifest   VirtualBoxModule              Get-VM
```

The reason the previous cmdlet works is that it does not actually force either of the modules to load into the session via cmdlet `auto-discovery`. Instead, internally it reads the manifest files and searches for cmdlet within those manifests and returns the corresponding modules. Instead of using such a complicated command it is possible to once again use the `Get-Command` cmdlet. Albeit, this example may not work in every circumstance, it will work for the previous example as follows:

```
PS C:\> Gt- Command Get-VM*


CommandType     Name        ModuleName
-----------     ----        ----------
Function        Get-VM      HyperVModule
Function        Get-VM      VirtualBoxModule
```

Because we used a wildcard in the call to `Get-Command`, the system did not use the `auto-discovery` cmdlet as explained earlier. The problem with this method is that it may find cmdlets simply starting with `Get-VM` in addition to the cmdlets we are actually searching for.

Another issue associated with cmdlet `auto-discovery` is that it can mask issues associated with the modules it is attempting to auto-load. If, for instance, I had incorrectly typed the `NestedModule` property in the `HyperVModule` manifest, I would receive an error while attempting to load the module with `Import-Module`, shown as follows:

```
#Notice I'm missing a close single quote
CompanyName = 'Get-VM
```

This is what happens when I attempt to load the module explicitly with the `Import-Module` cmdlet as follows:

```
PS C:\Users\Adam> Import-Module HyperVModule
Import-Module : The module manifest 'C:\Users\Adam\Documents\
WindowsPowerShell\Modules\HyperVModule\HyperVModule.psd1'
could not be processed because it is not a valid PowerShell restricted
language file. Please remove the elements that are not permitted by the
restricted language
```

If I were now to attempt to call the `Get-VM` cmdlet, I would receive an error. Rather than continuing on to the next module that contains `Get-VM`, the `VirtualBoxModule`, PowerShell stops processing because of an error in the `HyperVModule` as follows:

```
PS C:\ > Get-VM
The term 'Get-VM' is not recognized as the name of a cmdlet, function,
script file, or operable program. Check the spelling of the name, or if a
path was included, verify that the path is correct and try again.
```

This is important to note for both duplicate cmdlet scenarios and for module development, as the issue will not be obvious. The cmdlet `auto-discovery` is a very helpful feature but understanding the few caveats associated with it are a fundamental part of using it.

Summary

As seen in this chapter, Microsoft has taken some big steps forward in providing a better user experience for PowerShell users. We learned that the updatable help system allowed developers and users the benefit of the most up-to-date examples and descriptions for all the cmdlets and topics that a module may expose. The `Show-Command` cmdlet ensured that users, that may be weary of the command line, will feel at home with a simple graphical user interface to enter parameter values and view the help. The improvements to cmdlet discovery ensured that as the list of modules grow, the users will still be able to easily find and use the cmdlets that they need to.

In the next chapter we will look at some of the new usability enhancements to some of the core components common to every PowerShell user.

2
Usability Enhancements

In the previous chapter we examined what Microsoft has done to make some of the core features better. In this chapter we will look at some simple, but very important changes to a few of the usability aspects of PowerShell. First, we will look at the changes to the `Where-Object` cmdlet and the `ForEach-Object` cmdlet. The first version of these cmdlets worked well but proved confusing for the first time users and required excessive syntactical noise to be used. Next, we will explore the changes to the tab completion behavior in PowerShell 3.0 that makes it much easier to use. Additionally, a bug was fixed that any regular PowerShell user would find very annoying. Finally, we will investigate the changes to the `Get-ChildItem` cmdlet for the file system provider. The latest version adds new dynamic parameters that make it easier to find the types of files and directories you are looking for with a terse, self-documenting syntax.

In this chapter we will cover the following:

- The changes to the `Where-Object` and `ForEach-Object` cmdlets to improve usability
- The new and improved tab expansion
- The enhancements to the `Get-ChildItem` cmdlet for the file system provider

The problem with Where-Object and ForEach-Object

The PowerShell pipeline is a very, if not the most, important feature of the language. Piping from one cmdlet to another offers numerous benefits including removing unnecessary storage variables and eliminating the need to create loops. Most of the time cmdlets offer a very robust piping mechanism. Often related cmdlets are developed in unison so that piping is as effective and fluent as possible.

For example, here is a script that gets the `Themes` service and starts it. This is an example of two well paired cmdlets. There is very little coding necessary and the fluency of the pipeline seems natural:

```
PS C:\> Get-Service –Name Themes | Start-Service
```

Expectedly, cmdlets cannot fit every situation. Sometimes the parameter or parameter set cannot be determined in piping situations. Other times the cmdlets simply do not offer a good piping or filtering mechanism. As more third-party vendors become involved in PowerShell, this most likely will become more common amongst cmdlet developers that may not have the rigorous standards that Microsoft does.

There are two utility cmdlets that have been available in PowerShell since the beginning: `Where-Object` and `ForEach-Object`. These cmdlets allow the user to adjust the pipeline between cmdlets. `Where-Object` can limit the objects returned by `Get` cmdlet when they do not offer the filtering mechanism that matches the user's circumstance. The `Where-Object` cmdlet filters objects in the pipeline like the SQL where clause filters records.

For example, here is a script that filters all the `Process` objects returned by `Get-Process` that have a handle count over 1001 and stops them. This is with the aid of `Where-Object`:

```
PS C:\ > Get-process | Where-Object { $_.Handles -gt 1001 } |
Stop-Process
```

Historically, `Where-Object` and `ForEach-Object` both suffered from excessive syntactical noise as we saw previously. There was no way to shortcut the amount of typing necessary to execute these cmdlets. The use of a **script block**, a script snippet surrounded by curly brackets, required a lot more typing. In an effort to shorten the syntax, an alias was created for the `Where-Object` cmdlet that is simply a question mark. The following example returns all the processes that have started in the last minute:

```
PS C:\> Get-Process | ? { $_.StartTime -gt (Get-Date).AddMinutes(-1)}
```

Although this saved some typing it made it a whole lot more confusing. Additionally, the use of the `$_` variable (current object) was unclear to new users. It can be a very difficult concept to grasp because the variable is not self-descriptive. There were ways around this issue. For example, using the `foreach` statement it was possible to forgo the `$_` variable and use a variable that is domain-specific, as shown in the following commands:

```
$processes = Get-Process -Name PowerShell
foreach($process in $processes)
{
$processs.Stop()
}
```

In this script we are getting all the processes named `PowerShell` and storing them in the `$processes` variable. We then are looping through each process and stopping it. Although the `foreach` operator statement offers a much more understandable syntax, it is even worse when it comes to the amount of typing. In the end, it's necessary to define a variable and introduce braces and parentheses, all to call a single method on each one. Selecting multiple properties from an object was a bit easier as the `Select-Object` cmdlet could do this without the use of a script block as follows:

```
PS C:\> Get-Process -Name PowerShell | Select-Object -ExpandProperty ID
```

This command simply gets all the `PowerShell` processes and returns a list of the process ID (PID), values of each, and writes them to the pipeline. This is much easier than creating a script block to select the properties we want to send to the pipeline. Although each of these methods offered a reasonable workaround, it is not the ideal scenario.

Fixing the problem with Where-Object

PowerShell 3.0 fixes the problem with `Where-Object` and `ForEach-Object` by simplifying how they work. Rather than having to deal with the `$_` variable and the script block mechanism, Microsoft added numerous new parameter sets to `Where-Object` and a new parameter to `ForEach-Object`. The more complicated `Where-Object` script presented earlier can be now written in a much more simplified syntax as follows:

```
PS C:\> Get-Process | Where-Object Handles -GT 1001 | Stop-Process
```

The script still returns all the processes that have more than 1001 handles and stops them (be careful running this script locally! Consider using the `WhatIf` parameter to ensure that only the processes you are looking to stop will be stopped.). The `Where-Object` no longer needs the braces or the `$_` variable. Notice how, like a well-designed pipeline cmdlet, the command now reads very much like English. The pipeline is no longer interrupted by an ugly, confusing, and non-descriptive script block. Utilizing the default `Where-Object` alias, `Where`, makes it read even more like spoken language, as shown in the following command line:

```
PS C:\> Get-Process | Where Handles -GT 1001 | Stop-Process
```

To better understand how to use this cmdlet correctly, it is good to examine how the cmdlet was developed to behave this way. In the original `Where-Object` cmdlet, there was a single parameter set, as explained by the following command:

```
Where-Object [-FilterScript] <scriptblock> [-InputObject <psobject>]
[<CommonParameters>]
```

A **parameter set** is a collection of parameters that are intended to be used together. Often parameters can appear in one parameter set but not another, while other parameters can appear in both. This allows cmdlets to adapt to different intents and to expand the amount of functionality without too much complication.

The original parameter set simply took a collection of objects from the pipeline, or otherwise, and filtered them based on the script block provided. It is required that the script block returns a boolean value. Any object matching that boolean logic would then be written back to the pipeline, as seen in previous examples.

In an effort to simplify the use of the cmdlet, many new parameter and parameter sets have been added. Each of the new parameters and parameter sets match one of the PowerShell comparison operators. You can use the following command to get more information about each one of the comparison operators:

```
PS C:\> Get-Help about_Comparison_Operators
```

The comparison operator similarity can be seen in the previous example in which we were filtering processes by the number of handles that they possess. In that example we used the `-GT` (greater than) parameter of the `Where-Object` cmdlet. This parameter was designed to look much like its accompanying comparison operator but in reality are two different things.

As an example, you will see, the following command uses the `-GT` comparison operator:

```
PS C:\> $MyVariable -GT 100
```

In this case, we are using the `-GT` operator rather than the parameter. There are many keywords in PowerShell, such as `if`, `else` and `foreach`. Unlike these keywords, the `Where-Object` cmdlet exposes parameters that have the same name as their related comparison operator. It is quite an ingenious way to deal with the problems with `Where-Object`. Rather than overcomplicating the already confusing cmdlets, Microsoft reused the same syntactical construct in a completely different way. This allows scripts to become much more readable and easier to understand for novices and experts alike.

The following is a table of the `Where-Object` parameter sets and parameters used for filtering pipelined objects. The following list of parameters can be located in the contextual help system for the `Where-Object` cmdlet:

Parameter set	**Parameter**
`EqualSet`	`EQ`
`ScriptBlockSet`	`ScriptBlock`
`CaseSensitiveGreaterThanSet`	`CGT`
`CaseSensitiveNotEqualSet`	`GNE`
`LessThanSet`	`LT`
`CaseSensitiveEqualSet`	`CEQ`
`NotEqualSet`	`NE`
`GreaterThanSet`	`GT`
`CaseSensitiveLessThanSet`	`CLT`
`GreaterOrEqualSet`	`GE`
`CaseSensitiveGreaterOrEqualSet`	`CGE`
`LessOrEqualSet`	`LE`
`CaseSensitiveLessOrEqualSet`	`CLE`
`LikeSet`	`Like`
`CaseSensitiveLikeSet`	`CLike`
`NotLikeSet`	`NotLike`
`CaseSensitiveNotLikeSet`	`CNotLike`
`MatchSet`	`Match`
`CaseSensitiveMatchSet`	`CMatch`
`NotMatchSet`	`NotMatch`
`CaseSensitiveNotMatchSet`	`CNotMatch`
`ContainsSet`	`Contains`
`CaseSensitiveContainsSet`	`CContains`
`NotContainsSet`	`NotContains`
`CaseSensitiveNotContainsSet`	`CNotContains`
`InSet`	`In`

Parameter set	Parameter
`CaseSensitiveInSet`	`CIn`
`NotInSet`	`NotIn`
`CaseSensitiveNotInSet`	`CNotIn`
`IsSet`	`Is`
`IsNotSet`	`IsNot`

Notice that each of these items is in its own parameter set so they cannot be specified together. That makes it really easy for the cmdlet to determine what we are trying to accomplish. In addition, notice that there are no `-AND` or `-OR` parameters, so complex queries will still need to be done (the old way) within a script block parameter set. When utilizing this cmdlet with one of the comparison operator parameters (all parameters with exception to the `ScriptBlock`), you are required to provide a left-hand and right-hand argument for the comparison. So if you are providing a list of `ProcessInfo` objects, for example returned by `Get-Process`, you must specify one of the properties on that object for comparison, as shown in the following command:

```
PS C:\> Get-Process | Where-Object Id -EQ 1000
```

If any of these properties are not provided, an error will be shown as follows:

```
PS C:\> Get-Process | Where-Object Name -EQ
Where-Object : The specified operator requires both the -Property and
-Value parameters. Supply both parameters and
retry.
```

Additionally you cannot specify properties of the properties. Unfortunately this will not result in an error. Instead, the query will simply return `$null`. The following command explains this:

```
PS C:\> Get-Process | Where-Object StartTime.DayOfWeek -EQ Thursday
```

The same thing will happen when you attempt to query a property that does not exist on the object you are filtering through, as follows:

```
PS C:\> Get-Process | Where-Object MyProperty -EQ $true
```

This may be a stumbling point for some. From first glance, it does appear that no objects are matching the specified filtering criteria. If you look at it literally, none of the objects are matching the criteria, but not due to them failing the query, but rather to them not containing the specified property.

Fixing the problem with ForEach-Object

The `ForEach-Object` has got a much simpler treatment but still reaps the benefits of simplicity. In PowerShell 2.0 there were two ways to gather the properties of pipeline objects. The first and most common way of accomplishing this would be to use the `Select-Object` cmdlet as follows:

```
PS C:\> Get-Service | Select-Object DisplayName,Status
DisplayName                        Status
-----------                        ------
Application Experience             Stopped
Application Layer Gateway Service  Stopped
Windows All User Install Agent     Stopped
Application Identity               Stopped
Application Information            Running
```

Although less common, this can be done by using the `ForEach-Object` cmdlet as well, as follows:

```
PS C:\> Get-Service | ForEach-Object { @{DisplayName=$_.
DisplayName;Status=$_.Status} }


Name                     Value
----                     -----
DisplayName              Application Experience
Status                   Running
DisplayName              Application Layer Gateway Service
Status                   Stopped
```

For obvious reasons, in this circumstance, using the `Select-Object` cmdlet would be the better mechanism. The output from `ForEach-Object` is not as nicely formatted and takes much more typing. The one benefit that `ForEach-Object` has over `Select-Object` is that it can do anything within the script block. Typically, when a method needs to be called rather than a property, it is necessary to use the `ForEach-Object` script block as follows:

```
PS C:\> Get-Service Hyper*  | ForEach-Object { $_.Pause() }
```

The problem with this `ForEach-Object` call is the excessive overhead of the syntactical components. From a usability perspective, we are attempting to get each service that starts with Hyper and call the `Pause` method on them. The parenthesis, `$_` variable and script block braces are confusing and pretty unnecessary for such a simple command. In the newest version of PowerShell, Microsoft made it much easier to achieve what we are trying to do. The new syntax allows the user to simply specify the name of a method that they wish to call on the pipelined objects as follows:

```
PS C:\> Get-Service Hyper* | ForEach-Object -MemberName Pause
```

Much like the `Where-Object` syntax presented earlier, the `ForEach-Object` now can read much more like the English language. By utilizing the default `ForEach-Object` alias `foreach`, you can see this as follows:

```
PS C:\> Get-Service Hyper* | foreach Pause
```

Note that the member name is case insensitive so specifying "Pause" or "pause" would yield the same result.

The `ForEach-Object` is not limited to only methods without parameters. There is a new `ArgumentList` parameter as well. This allows you to provide the arguments of a method as an array of values. Each value must either be of the expected type or be convertible to the expected type. For example the `String` class has a method `Insert` as follows:

```
PS C:\> "AString" | Get-Member Insert


   TypeName: System.String


Name   MemberType Definition
----   ---------- ----------
Insert Method     string Insert(int startIndex, string value)
```

Piping an array of strings to the `ForEach-Object` cmdlet, specifying the `Index` method of `String` class and providing a list of arguments could yield this effect as follows:

```
PS C:\> "a Bird!","a Plane!","Superman!" | ForEach-Object Insert
-ArgumentList 0,"It's "
It's a Bird!
It's a Plane!
It's Superman!
```

Just as with the `Where-Object` cmdlet, specifying a property of a property is not supported by this cmdlet and will fail, returning `$null` as the following command shows:

```
PS C:\> Get-Service Hyper* | ForEach-Object MachineName.Length
```

Enhancements to tab completion

Tab completion is the feature that allows you to begin typing a command, parameter, or paths, press the *Tab* key and have the PowerShell engine recommend possible matches to complete what you are typing. This saves thousands of key strokes, for a typical administrator a day. It can be helpful for discovery purposes as well, for example, when attempting to determine the contents of directories or finding related cmdlets.

In PowerShell 3.0, Microsoft enhanced the experience and fixed an issue that caused the most administrator grief. The annoying issue that was fixed is referred to as **midline tabbing**. If the user were to start typing a command, or possibly copy the command into a command prompt and move the cursor to the middle of the line, the command prompt would delete anything after the completed parameter.

Take into consideration the following command line:

```
PS C:\> Start-job -ScriptBlock { Get-WmiObject -Class Win32_Process }
```

This command will start a job that will return all instances of the `Win32_Process` WMI class. Now, let's assume that we weren't satisfied with an unnamed job and we wanted to add the name parameter to the call, but we wanted it to come before the `ScriptBlock` parameter for better readability. If we began typing the parameter, pressed the *Tab* key (as denoted by the `<tab>` syntax in the following command) the command line would have deleted the rest of the line, after the tab completed parameter:

```
PS C:\> Start-job –N<tab> -ScriptBlock { Get-WmiObject -Class
Win32_Process }
```

The previous command would result in the following in PowerShell 2.0:

```
PS C:\> Start-job –Name
```

This can be quite annoying and time consuming as we would have to write the entire command over again. Doing the same operation in PowerShell 3.0 works as you would expect, resulting in the following:

```
PS C:\> Start-job –Name -ScriptBlock { Get-WmiObject -Class Win32_Process
}
```

This simple usability improvement was much needed. The same tab completion behavior is available for the nested script block found in our `Start-Job` command line call in this example. For example, adding a computer name to the `Get-WMIObject` call before the `Class` parameter does not result in the deletion of the rest of the command line as follows:

```
PS C:\> Start-job –Name -ScriptBlock { Get-WmiObject –Co<tab> -Class
Win32_Process }
```

The earlier command line results in the following:

```
PS C:\> Start-job –Name -ScriptBlock { Get-WmiObject –ComputerName -Class
Win32_Process }
```

In PowerShell 2.0 it was possible to customize the tab completion behavior. You could simply override the default function `TabExpansion` with your own version. In PowerShell 3.0 there is no longer a default `TabExpansion` function but rather a default `TabExpansion2` function. We can see this by calling the following command:

```
PS C:\> Get-Command TabExpansion*

CommandType      Name

----------      ----

Function          TabExpansion2
```

It is still possible to override the tab completion behavior through the use of the original `TabExpansion` function, rather than overriding the new `TabExpansion2` function. PowerShell will call the custom, user-defined `TabExpansion` function if it exists in the current session. The tab completion function needs to return an array of possible values or `$null` as follows:

```
PS C:\> function TabExpansion { "Get-ChildItem" }
```

This tab completion function is a bad idea because all tab expansion will result in the cmdlet `Get-ChildItem` written to the command line but it demonstrates how easy it is to create our own tab completion behavior. Using our new, poorly designed tab completion function we could begin typing a set command as follows:

```
PS C:\> Set<tab>
```

This would result in the following being written to the command line:

```
PS C:\> Get-ChildItem
```

Not the most useful tab expansion behavior! To remove our custom tab completion we can use the function provider that comes built into PowerShell. The command sequence will reset the tab completion behavior back to before we started. The following command shows this:

```
PS C:\> Remove-Item function:\TabExpansion
```

It is possible to remove the pre-defined `TabExpansion2` function using the same method, but this will leave us with no tab completion at all. In addition to the enhancements for midline tabbing the core modules now support better completion of parameter values, even when no parameters are specified. Previously, for a cmdlet such as `Stop-Process`, you would have to type out the name of a running process completely before sending it in as a parameter value as follows:

```
PS C:\> Stop-Process Notepad
```

In PowerShell 3.0, many of the core cmdlets support parameter value completion. You could now type the following:

```
PS C:\> Stop-Process N<tab>
```

This would result in the tab completion system returning back a process starting with an `N` as we press the *Tab* key. If `Notepad` was the only running process, or other processes starting with `N` fell after it alphabetically, the following would result:

```
PS C:\> Stop-Process Notepad
```

This is a really handy tool to have when we know what we are trying to achieve on the command line and want to save a boat load of typing. The cmdlets that do support it can support the feature on multiple properties. For instance, the `Get-Job` cmdlet supports both `Id` and `Name` parameter value expansion. Some other cmdlets that support this type of expansion include `Get-EventLog` for log names and `Get-Service` for service names as follows:

```
PS C:\ > start-job -Name test -ScriptBlock { Get-Process }


Id      Name            State         HasMoreData     Location
--      ----            -----         -----------     --------
2       test            Running       True            localhost
PS C:\> Get-Job <tab>
```

This would yield the following:

```
PS C:\> Get-Job 2
```

The reason the `Id` is used in the previous circumstance is that it is marked as position zero. This information can be located in the help for the cmdlet. If we were to specify the `Name` parameter the name would then be expanded as follows:

```
PS C:\> Get-Job –Name <tab>
PS C:\> Get-Job –Name test
```

The final enhancement to tab completion is the ability to tab-expand enumerated values for the parameters. Along the same lines of the previous parameter expansion, this type of expansion happens for the values of cmdlet parameters. In this case though it will work on more than just the core cmdlet. Any cmdlet parameter that exposes a set enumeration type, for instance `ExecutionPolicy`, will benefit from this, as shown in following command:

```
PS C:\> Set-ExecutionPolicy –ExecutionPolicy <tab>
PS C:\> Set-ExecutionPolicy –ExecutionPolicy AllSigned
```

Pressing the *Tab* key again, in this scenario, will cycle through the possible values, just as `Get-Process` or `Stop-Service` would do. Script authors can provide this feature by using the `ValidateSet` validation attribute. This attribute ensures that only certain values can be provided to a parameter. The tab expansion will enumerate these values as well.

Improvements to Get-ChildItem

`Get-ChildItem` is a great cmdlet because it traverses multiple providers. For instance, it will work with both the file system and the registry provider. Using it with the file system provider, will return files and directories from the current location as follows:

```
PS C:\> Get-ChildItem
    Directory: C:\
Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----        12/28/2009   9:40 PM            Backup
d----         11/6/2010   1:45 PM            Builds
d----         9/18/2010   9:42 AM            Development
```

Running the same cmdlet within the context of the registry will have vastly different results as follows:

```
PS HKLM:\software> Get-ChildItem

    Hive: HKEY_LOCAL_MACHINE\software

Name                           Property
----                           --------
7-Zip                          Path : C:\Program Files\7-Zip\
AGEIA Technologies             PhysX Version  : 9110621
                               PhysX BuildCL  : 1
                               HwSelection    : GPU
                               PhysXCore Path : C:\Program Files (x86)\
NVIDIA Co…
ASUS
Atheros
```

The goal of the cmdlet is to return the children of the current item or items specified by the `Path` or `LiteralPath` parameters. Depending on the provider, this item and children mean different things. The `Get-ChildItem`, along with other provider-based cmdlets, can expose new parameters based on the current provider context. In PowerShell 3.0, the file system provider has added several useful parameters to the `Get-ChildItem` cmdlet to make it easier to search for files holding particular attributes.

The problem with Get-ChildItem

To understand the need for something like this, it is a beneficial exercise to see what it took in PowerShell 2.0 to look for files or directories that contain only particular attributes. First, let's see how to return only files in a particular directory as follows:

```
PS C:\> Get-ChildItem | Where-Object { -not $_.PSIsContainer }

    Directory: C:\

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---          3/1/2010  11:37 PM      40054 temp.jpg
-a---         6/29/2010   6:32 PM         10 testScript.ps1
```

PowerShell's extensible type system added the `PSIsContainer` `NoteProperty` to the object type, `FileInfo` and `DirectoryInfo`, returned by the `Get-ChildItem` cmdlet.

We then use the `Where-Object` cmdlet to find all items that are not containers. This is a lot of typing for a seemingly common and simple task. To retrieve files that match a particular attribute set is even more complex:

```
PS C:\> Get-ChildItem -Force | Where-Object { -not $_.PSIsContainer -and
$_.Attributes -band [IO.FileAttributes]::Archive }
    Directory: C:\
Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a-hs        12/11/2011  10:10 AM 3220037632 hiberfil.sys
-a-hs        12/11/2011  10:10 AM 4293386240 pagefile.sys
-a---          3/1/2010  11:37 PM      40054 temp.jpg
-a---         6/29/2010   6:32 PM         10 testScript.ps1
```

This command returns all child items of the `C:\` path and filters them by not being a container and by containing the attribute `Archive`. The use of the `Force` parameter instructs `Get-ChildItem` to return hidden files. This is a complex command to say the least! This is especially confusing for users transitioning from the Window's command prompt where the `dir` command could take care of this much easier, as shown in the following command:

```
C:\>dir /a:a *.*
 Volume in drive C has no label.
 Volume Serial Number is B43D-0A38
 Directory of C:\
12/11/2011  10:10 AM     3,220,037,632 hiberfil.sys
12/11/2011  10:10 AM     4,293,386,240 pagefile.sys
03/01/2010  11:37 PM            40,054 temp.jpg
06/29/2010  05:32 PM                10 testScript.ps1
               4 File(s)  7,513,463,936 bytes
               0 Dir(s)  209,774,366,720 bytes free
```

Now that we have an understanding of the problem with the previous version of `Get-ChildItem`, let's look at the new version to see how it improves the user experience and changes the grade of the extremely steep learning curve.

A better Get-ChildItem

If you look at the syntax definition for the `Get-ChildItem` cmdlet we will see all the new parameters that are available to us as follows:

```
Get-ChildItem [[-Path] <string[]>] [[-Filter] <string>] [-Include
<string[]>] [-Exclude <string[]>] [-Recurse]
[-Force] [-Name] [-UseTransaction] [-Attributes <FlagsExpression[FileAttr
ibutes]>] [-Directory] [-File] [-Hidden]
[-ReadOnly] [-System] [<CommonParameters>]


Get-ChildItem [[-Filter] <string>] -LiteralPath <string[]> [-Include
<string[]>] [-Exclude <string[]>] [-Recurse]
[-Force] [-Name] [-UseTransaction] [-Attributes <FlagsExpression[FileAttr
ibutes]>] [-Directory] [-File] [-Hidden]
[-ReadOnly] [-System] [<CommonParameters>]
```

These additional parameters can be used to select files and directories that match specified attribute criteria. For example, if we wanted to select all the files in the root `C` drive we could do the following:

```
PS C:\> Get-ChildItem -File
    Directory: C:\
Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---          3/1/2010  11:37 PM      40054 temp.jpg
-a---         6/29/2010   6:32 PM         10 testScript.ps1
```

Notice that only files were selected within the `C` drive. These parameters are not mutually exclusive and can be specified together. If we were to add on the `Hidden` parameter we would see the following:

```
PS C:\> Get-ChildItem -File -Hidden
    Directory: C:\
Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a-hs         12/8/2011   7:37 PM 3220037632 hiberfil.sys
-a-hs         12/8/2011   7:37 PM 4293386240 pagefile.sys
```

As you can see, only files that are hidden were returned. This is very useful as it lets us easily specify, whether our child item is a file and whether it is hidden or even a system file as well. Taking it a step further, assume that we may want to find both hidden and visible files. Using or omitting the `Hidden` parameter will not work because we want both file types. In this circumstance we will want to use the new `Attributes` parameter. This parameter allows us to specify a formatted special string that enables us to filter files and directories based on attributes in a more complex manner. If we wanted to achieve the result of both hidden and unhidden files, we would do the following:

```
PS C:\> Get-ChildItem -Attribute !Directory+Hidden,!Directory

    Directory: C:\

Mode                LastWriteTime     Length Name

----                -------------     ------ ----

-a-hs         12/8/2011   7:37 PM 3220037632 hiberfil.sys

-a-hs         12/8/2011   7:37 PM 4293386240 pagefile.sys

-a---          3/1/2010  11:37 PM      40054 temp.jpg

-a---         6/29/2010   6:32 PM         10 testScript.ps1
```

In this command, we filtered all the child items that were not directories and hidden, and then again by objects that were not directories. This provided us with both hidden and unhidden files in the `C` drive. The `Attributes` parameter uses several special characters to denote different parts of the filter. The special operators are as follows:

- !: This is the `NOT` operator. Inserting it before a particular value specifies that we do not want that particular type
- +: This is the `AND` operator. Is specifies that we want the latter flag to be honored as well as the former
- ,: This is the `OR` operator. It allows us to specify filters where if the child item matches the latter flag or the former flag, it will be returned as a match

In plain English, the filter we specified `(!Directory+Hidden,!Directory)` reads: Not a directory which is hidden or not a directory. As there are only two types of entities within a files system, files, or directories, the English could be simplified to read: hidden and unhidden files. Thus, all files, hidden or not, will be returned after the command. The filtering syntax of the `Attributes` parameter supports more attributes than the ones specified previously. It supports all values of the `System.IO.FileAttributes` enumeration. These values are as follows:

- `ReadOnly`
- `Hidden`
- `System`
- `Directory`
- `Archive`
- `Device`
- `Normal`
- `Temporary`
- `SparseFile`
- `ReparsePoint`
- `Compressed`
- `Offline`
- `NotContentIndexed`
- `Encrypted`

Several of the attributes support a short hand alias too. If we were to use one of the default aliases for `Get-ChildItem`, `dir`, along with the short hand alias for the attributes, we could have a very terse command. The following example returns those files as the previous example in much shorter syntax:

```
PS C:\> dir -af
    Directory: C:\
Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---          3/1/2010  11:37 PM      40054 temp.jpg
-a---         6/29/2010   6:32 PM         10 testScript.ps1
```

Summary

We learned that Microsoft has spent some time enhancing a few of the key cmdlets and features to be easier to use and easier to understand. Both novice and advanced users alike will benefit from the multitude of changes. The changes to `Where-Object` and `ForEach-Object` will have a huge impact on how scripts are written and the command line used. The elimination of the need to use the often confusing, and sometimes buggy, `$_` variable make working with the cmdlets less of a chore and make it read much more like a spoken language.

The changes to tab completion will no doubt make many long-time PowerShell users happy. The simple fix to midline tabbing may seem like a trivial fix to new users but having lost many command lines to midline tabbing in the past, existing users will enjoy the change. In addition the ability to discover enumeration values simply through tab completion will make writing scripts much easier.

Finally the changes to the `Get-ChildItem` cmdlet for the file system provider will make working with the file system that much easier. Short and effective commands can now accomplish very powerful queries and locate exactly what files or directories you are looking for.

In the next chapter we will look at how PowerShell 3.0 improves administration through features such as scheduled jobs and better remote sessions.

3
Improved Administration

PowerShell has always been about administration. From the different provider systems to the numerous modules, the system is a Windows administrator's dream. Even the first version was miles ahead of VB script. The rate of community acceptance alone is starting to prove that PowerShell is the de facto standard for administrators. The two previous versions simplified and exposed many integration points for administrators to easily automate their environments. With the power of .NET hiding behind it, PowerShell allows an everyday administrator to develop their own solutions without having to involve a developer.

With the third version of PowerShell, Microsoft weighed some of the feedback it received from the community and further enhanced the administrators' experience. We'll see how PowerShell is taking a step forward in several areas that will undoubtedly offer even more flexibility to the system.

In this chapter we will cover the following:

- Working with scheduled jobs
- Delegated administration
- Robust and resilient sessions
- Changes to the `Restart-Computer` cmdlet

Scheduling jobs

PowerShell 2.0 introduced the concept of a job. A job is a unit of work which can be run independently of what is going with the command line or the rest of the script. This allows for simultaneous units of work to run at the same time. Many jobs can be scheduled; all completing work in the background while either the command line continues to function or the script continues to run. This was a huge improvement over what Windows administrators had at their fingertips in the past. Rather than having to wait for a single unit of work to complete, we could trigger numerous actions at once. This saved time and allowed the administrator to work on something else. Additionally, jobs could be run remotely, spanning the entire infrastructure.

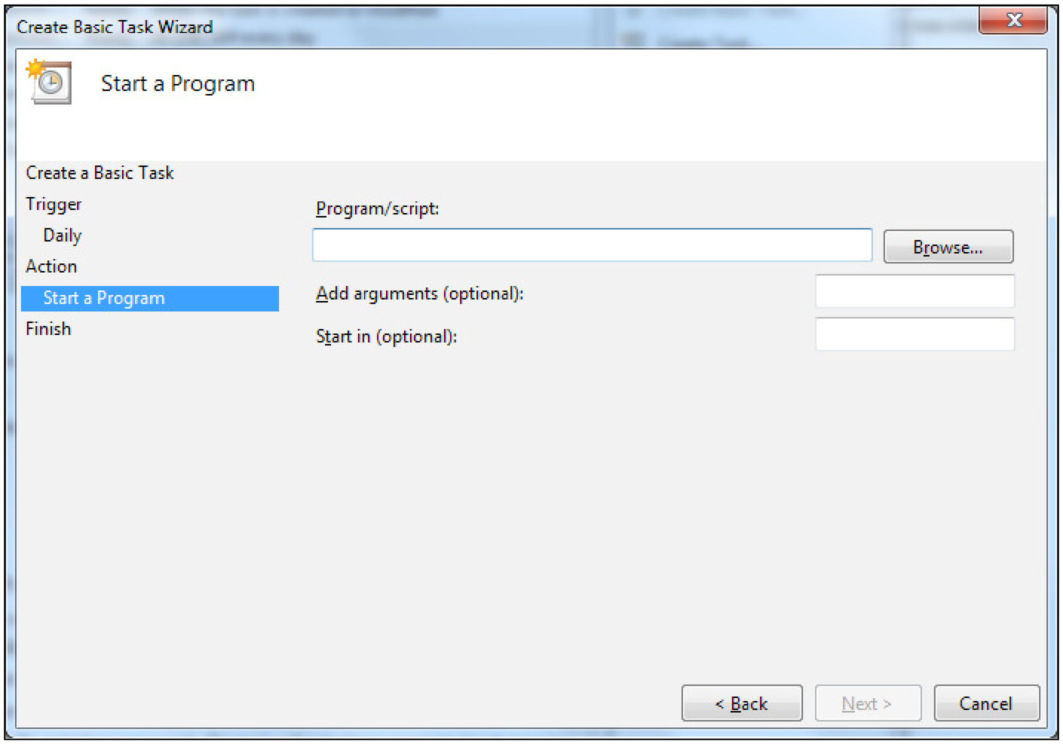
Rather than the concept of threads typically found in full-fledged programming languages, jobs were designed to be easy to understand and work with. Instead of having to deal with the implementation details of an independent unit of work, administrators can just worry about getting the job done in the simplest most effective way possible.

More often than not, an administrator will author a script with the goal of re-using that script later. There would be no reason to write a script if this was not the case. Imagine cleaning out contract worker accounts from Active Directory or finding mailboxes of Exchange users that are reaching their quota. These types of actions typically require more than a single command and typically require a full script to be written. Not only will scripts like these be repeated, they will need to be repeated on a regular basis. Before the current version of PowerShell, scripts such as this could be scheduled. The Windows Task Scheduler is a handy tool which can be used to run particular actions on a schedule, so they do not need to be done by hand.

In PowerShell 2.0 it was possible to run these scripts using the task scheduled with a little bit of finesse. It is possible to schedule the execution of a script using the PowerShell executable using the following command:

```
powershell.exe -File  C:\CleanOldUsers.ps1' "
```

This command could be added to the Windows Task Scheduler as a basic task. Using the task wizard, we could set a schedule, such as daily or weekly at 1:00 a.m., to run the task. This is much better than us having to run the task by hand at 1:00 a.m. This is shown in the following screenshot:



Although this was a fine solution, Microsoft took this concept and baked it right into PowerShell 3.0. Rather than having to manually add the task via the Windows Task Scheduler, cmdlets have been developed to take care of this right at the command line.

Creating a job trigger

The first step in scheduling a new job is to create an object that defines the schedule upon which the job will start. This is known as the **job trigger** as well. The `New-JobTrigger` cmdlet creates a `ScheduledJobTrigger` job that then can be passed to the `Register-ScheduledJob` cmdlet. If we wanted to use our previous example, we could setup a job trigger for 1:00 a.m. daily as follows:

```
PS C:\> New-JobTrigger -At 1:00AM -Daily


Id          Frequency       Time                      DaysOfWeek
--          ---------       ----                      ----------
0           Daily           12/14/2011 1:00:00 AM
```

Notice that our newly created object is set up for the current date. This is when the job will first start. It will then be triggered daily at 1:00 a.m. In addition to scheduling jobs daily, it is possible to schedule jobs weekly or just once. This can be done with the `Weekly` or `Once` parameters. `Once`, `Daily`, and `Weekly` all fall into their own parameter set so they cannot be specified together. To create a schedule of jobs, that requires more than one job trigger, simply run `New-JobTrigger` more than once, creating more than one object.

The `New-JobTrigger` cmdlet requires that the `At` parameter be specified. The `At` parameter accepts a `DateTime` object. Any valid date and time string will correctly format into a `DateTime` object. In the following examples, the `At` parameter takes the values on the left-hand side. The results of these values are explained on the right-hand side:

1:00 PM: Job starts today at 1:00 PM

1/12/12: Job starts January 1, 2012 at 12:00 AM

1/12/12 1:00 PM: Job starts January 1, 2012 at 1:00 PM

To offset jobs slightly we can even specify the `RandomDelay` parameter. This parameter accepts a `TimeSpan` object. A `TimeSpan` object is, as it sounds, a span of time. Specifying this parameter causes the job to randomly trigger from the time specified by the `At` parameter plus the `RandomDelay` parameter as follows:

```
PS C:\> New-JobTrigger -Daily -At 1:00PM -RandomDelay (New-TimeSpan
-Minutes 1)
```

This example will cause the job to be triggered daily, sometime between 1:00 p.m. and 1:01 p.m. This allows for many jobs to start around 1:00 p.m. without having them all start at exactly 1:00 p.m., possibly causing performance or network issues.

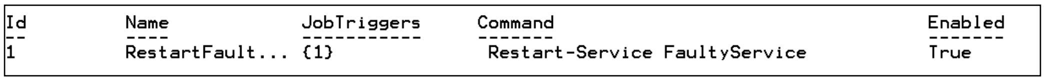
To make the `New-JobTrigger` cmdlet useful, it is necessary to pass the result of the cmdlet to the `Register-ScheduledJob` cmdlet. This is most easily done by utilizing a variable as follows:

```
PS C:\ > $DailyTrigger = New-JobTrigger -At 1:00AM -Daily
```

The `$DailyTrigger` variable can now be passed to the `Register-ScheduledJob` cmdlet in order to schedule our job using following command:

```
PS C:\ > Register-ScheduledJob -Name RestartFaultyService -ScriptBlock {
Restart-Service FaultyService }
-Trigger $DailyTrigger
```

This produces the output shown in screenshot:



```
Id          Name             JobTriggers      Command                                  Enabled
--          ----             -----------      -------                                  -------
1           RestartFault...  {1}               Restart-Service FaultyService           True
```

The same scheduling can be done by the following two methods. In this scheduled job, we have set up a script block which runs each day at 1:00 a.m. Its goal is to restart a service which does not work as expected. Notice the braces around the `Triggers` property. This is because the `Register-ScheduledJob` cmdlet accepts multiple triggers for an individual scheduled job.

In the first method, we could add additional triggers that would cause this particular job to trigger. If we run the `Get-ScheduledJob` cmdlet, our newly created job will be returned using the following command:

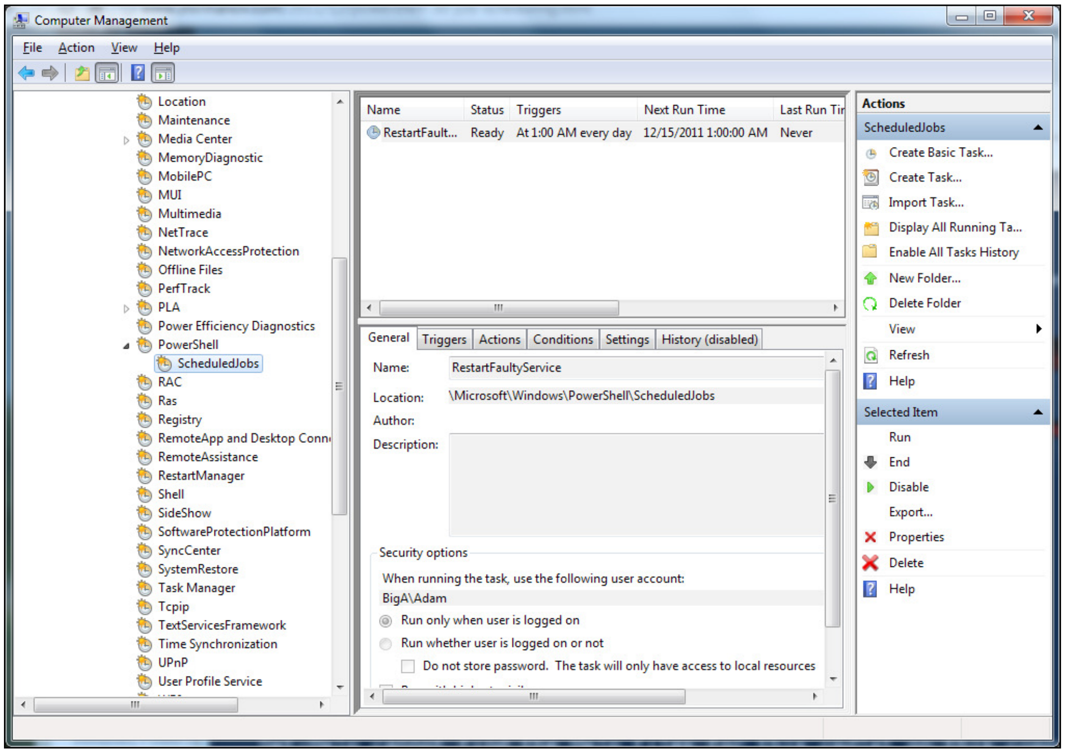
```
PS C:\ > Get-ScheduledJob
```

In the second method, the `Get-ScheduledJob` cmdlet accepts both name and ID as input, so that only particular jobs are returned as follows:

```
PS C:\ > Get-ScheduledJob -Name RestartFaultyService
```

Viewing scheduled jobs in the Windows Task Scheduler

In addition to finding the job using the `Get-ScheduleldJob` cmdlet, we can look in the Windows Task Scheduler window to see that our job has been created there as well. This is shown in the following screenshot:



Name	Status	Triggers	Next Run Time	Last Run Ti
RestartFault...	Ready	At 1:00 AM every day	12/15/2011 1:00:00 AM	Never

Name: RestartFaultyService

Location: \Microsoft\Windows\PowerShell\ScheduledJobs

Author:

Description:

Security options

When running the task, use the following user account:

BigA\Adam

[✓ Run only when user is logged on]

Run whether user is logged on or not

☐ Do not store password. The task will only have access to local resources

All scheduled PowerShell jobs can be found under the path `Microsoft\Windows\PowerShell\ScheduledJobs`.

Upon inspecting what comprises the scheduled task we can see that it is merely calling PowerShell with a few special arguments. This is very similar to the way which we would have to set up a scheduled job in PowerShell 2.0, except it utilizes a new module, the `PSScheduledJob` module, to take care of all the execution and state management for us. Refer to the following commands:

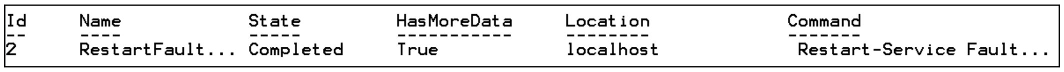
```
-NoLogo -NonInteractive -WindowStyle Hidden -Command "Import-Module
PSScheduledJob; $jobDef = [Microsoft.PowerShell.ScheduledJob.Schedule
dJobDefinition]::LoadFromStore('RestartFaultyService', 'C:\Users\Adam\
AppData\Local\Windows\PowerShell\ScheduledJobs'); $jobDef.Run()"
```

If we were to navigate to the path found in these arguments, we will find an XML file that is a serialized version of the `ScheduledJobDefinition` object. This object contains all the information necessary for defining the job. This is how the job is stored for future reference. Additionally we find an `Output` folder that, whenever the job runs, contains the result of the said job.

This is really just the internals of the scheduled job. It is not required to directly interact with the data found in this folder. To interpret this information, we can use the pre-existing job cmdlets such as `Get-Job` and `Receive-Job`. The `Get-Job` cmdlet will return the completed or running scheduled jobs as follows:

```
PS C:\ > Get-Job
```

This will create the following output:

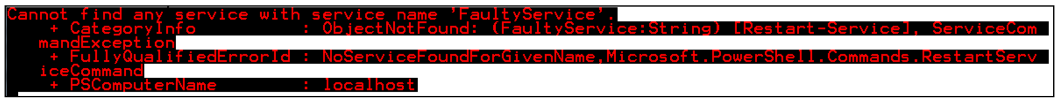


```
Id     Name             State         HasMoreData     Location             Command
--     ----             -----         -----------     --------             -------
2      RestartFault...  Completed     True            localhost             Restart-Service Fault...
```

Just as with an in-session job, we can receive the result of the specified job. This command reads the results of the scheduled job from the `Output` folder and writes it to the console for us to view as follows:

```
PS C:\ > Receive-Job 2
```

In this case, I really did not have a `FaultyService` running on my machine so the job failed with the error message, as shown in the following screenshot. This makes evaluating whether or not a job was completed much easier!



```
Cannot find any service with service name 'FaultyService'.
    + CategoryInfo          : ObjectNotFound: (FaultyService:String) [Restart-Service], ServiceCom
   mandException
    + FullyQualifiedErrorId : NoServiceFoundForGivenName,Microsoft.PowerShell.Commands.RestartServ
   iceCommand
    + PSComputerName        : localhost
```

Just as with the Windows Task Scheduler, it is possible to disable jobs using the command line. This can be accomplished using the `Disable-ScheduledJob` cmdlet. Passing in the name or ID of the job will disable it, as shown in the following command:

```
PS C:\> Disable-ScheduledJob -Name RestartFaultyService
```

If we want to re-enable the job we just use the `Enable-ScheduledJob` cmdlet as follows:

```
PS C:\> Enable-ScheduledJob -Name RestartFaultyService
```

When we no longer need a job we can remove it with the `Unregister-ScheduledJob` cmdlet as follows:

```
PS C:\> Unregister-ScheduledJob -Name RestartFaultyService
```

This will remove the job from both the Windows Task Scheduler and the file system, as shown earlier.

Working with scheduled jobs is as easy as working with jobs themselves. The seamless integration between the regular and scheduled jobs makes configuring running and monitoring them nearly identical. Additionally, as the Windows Task Scheduler is used to schedule and execute the jobs, there is no additional overhead incurred by the new functionality.

Delegated administration

Authentication and permissions are vital when using any type of computer system. Identity management and delegation is becoming an even bigger hurdle to overcome in the age of globally distributed systems. In Windows, permissions and authentication are typically done with access control lists and user credentials. Users are authenticated based on a user name, password, and given particular rights based on that account and which groups it is a part of. Just like everything else in Windows, PowerShell plays by the same rules. If a user does not have access to a particular file or folder, PowerShell will show an `access denied` error. In Microsoft operating systems such as Vista and beyond, user account control was added to ensure that the user could not be spoofed into providing control to malicious software, when it attempted to do something that may cause system instability. Registry access for example, requires that a PowerShell prompt be elevated by the `Run As Administrator` command. You may have encountered this when attempting to enable remoting via the `Enable-PSRemoting` cmdlet as follows:

```
PS C:\> Enable-PSRemoting
Enable-PSRemoting : Access is denied. To run this cmdlet, start Windows
PowerShell with the "Run as administrator"
option.
```

The cmdlet ensures that the current session is elevated enough to access the correct registry key when enabling this feature. Locally, permissions are pretty well defined. Local users and groups are easily queried and sets of permissions are applied. Domain systems, such as Active Directory, provide distributed administration amongst a trusted set of computers or other authenticating end points. This is evident in cmdlets that accept the `Credential` parameter such as `Enter-PSSession`. The following command explains this:

```
PS C:\> Enter-PSSession –ComputerName driscoll-host –Credential
ard\adriscoll
```

My credentials have traversed the computer system, and have been used to authenticate me against the target system. Now that we are in the target system, we can use the handy `whoami.exe` application that comes pre-installed on Windows 7 systems as follows:

```
[driscoll-host]: PS C:\> whoami
ard\adriscoll
```

The application simply returns the user name of the person who has executed the command. As you can see that my username comes back, as the user I have authenticated against the target system. This is expected behavior and ensures that the permissions that apply to the user throughout the domain apply on the target system as well.

The need for a more robust system

Although this configuration is ideal for most use cases, it is sometimes necessary to allow a user more control than they would typically be allowed. For example, permissions could be granted to reset users' passwords without actually having the privileges on those user accounts. Sometimes this requires privileges above and beyond those which are granted by default. Connecting to the `driscoll-host` machine with the `adriscoll` user may not provide all the necessary permissions required to, for example access the registry. In PowerShell 2.0, the user that `Enter-PSSession`, or related remoting cmdlets, would run as would always be the user that was authenticated via the `Credential` parameter.

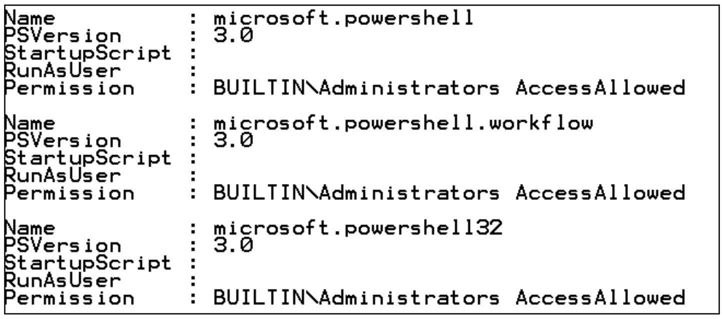
Providing additional permissions may require that the user authenticate as a different user all together. This requires that the user know the credentials for that secondary login. Microsoft has added an interesting feature to the third version of the **Windows Remote Management** (**WinRM)**, system to allow users to change which user a remote session is running under. This feature is called **delegated administration**. It is the ability to run a remote PowerShell session as a different user without knowing the credentials for that user. To understand how this works in PowerShell 3.0, it is necessary to understand how PowerShell session configurations work.

Examining PowerShell session configurations

PowerShell remoting is built upon WinRM to provide remote access to machines using the `Enter-PSSession` and `Invoke-Command` cmdlets. Whenever a user connects to a remote machine using one of these cmdlets, the WinRM service creates a new instance of a process to host the session. The cmdlets and objects are serialized back and forth to the remote computer. From a user perspective, it all is seamless. In order to better customize remote sessions such as this, Microsoft exposed the concept of a **PowerShell Session Configuration**. A session configuration allows for numerous customizations to take place when connecting to a remote machine via the named configuration. The configuration acts as an end point to which a user connects. A user can decide which end point they'd like to connect to, or use the default end point. When no end point is specified, users will connect to the `Microsoft.PowerShell` session configuration.

> Note that on a 64-bit system users will connect to the 64-bit endpoint by default. On 32-bit systems there will be a dedicated endpoint rather than two separate ones.

```
PS C:\ > Get-PSSessionConfiguration
```



```
Name          : microsoft.powershell
PSVersion     : 3.0
StartupScript :
RunAsUser     :
Permission    : BUILTIN\Administrators AccessAllowed

Name          : microsoft.powershell.workflow
PSVersion     : 3.0
StartupScript :
RunAsUser     :
Permission    : BUILTIN\Administrators AccessAllowed

Name          : microsoft.powershell32
PSVersion     : 3.0
StartupScript :
RunAsUser     :
Permission    : BUILTIN\Administrators AccessAllowed
```

Notice that in the first session configuration, **microsoft.powershell**, the **Permission** is set to allow `ard\adriscoll` all access. This effectively makes the user an administrator with all possible permissions. These configurations are defined on the local machine we are running this cmdlet on.

In order to use one of the session configuration end points it requires us to utilize the `Enter-PSSession` cmdlet, much like we were doing previously. In this circumstance we will specify the end point, or session configuration that we wish to connect to. Depending on how the session configuration is set up, our experience may differ as follows:

```
PS C:\> Enter-PSSession -ComputerName driscoll-host -Credential ard\
adriscoll -ConfigurationName Microsoft.PowerShell
```

Although issuing the command with this configuration name does the same as not specifying the configuration name at all, it illustrates the point that we can easily connect to a session that was set up for an entirely different use case. For example, running Active Directory cmdlets or maybe enabling Exchange access.

Session configurations are not a new concept to PowerShell, as they were introduced in the second version. The new feature is the ability to change the context within which the remote session is executing. In PowerShell 2.0, the remote session would execute under the delegated credentials of the user that authenticated via the remote session. This can now be configured using the session configuration cmdlets. Notice that the previous `Get-PSSessionConfiguration` cmdlet returned several `SessionConfiguration` objects. One of the properties of these objects is the `RunAsUser` property as follows:

```
Name          : microsoft.powershell32
PSVersion     : 3.0
StartupScript :
RunAsUser     :
Permission    : BUILTIN\Administrators AccessAllowed
```

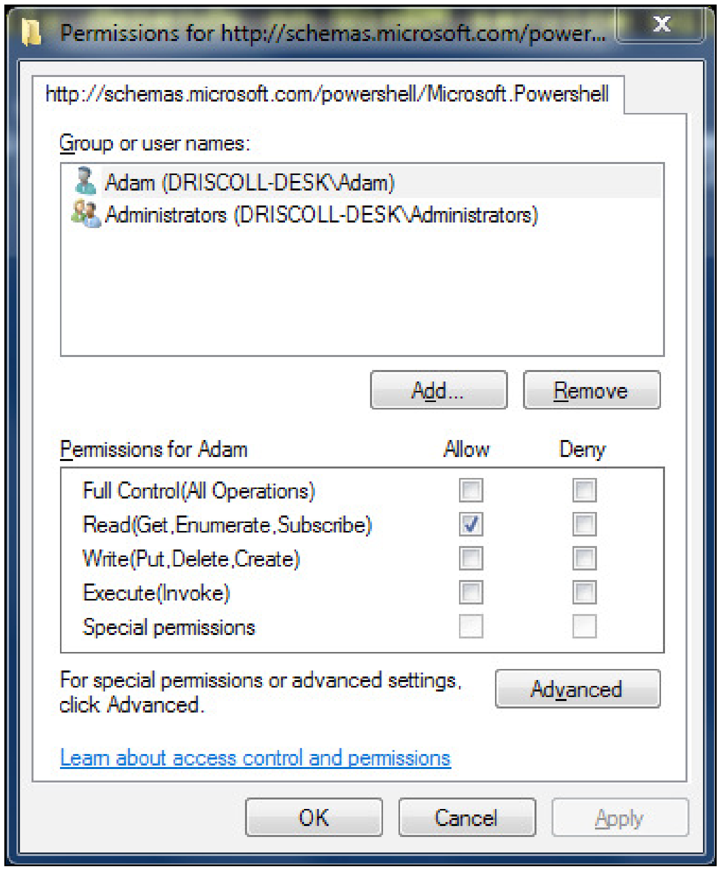
By default, the value is `$null` for the pre-defined session configurations. When this is the case, users connected to that session configuration will run as themselves. Any permission on that remote box will apply directly to that user. We can adjust this very simply by modifying an existing session configuration or creating a new one.

Modifying existing session configurations

First, let's see what it takes to modify one of the existing sessions. If for example, we wanted to give a user access to the default PowerShell session, who was not already a part of the local administrators' group, we could use the `Set-PSSessionConfiguration` cmdlet. This cmdlet modifies existing session configurations. There is a handy parameter for the cmdlet that will display a Windows access control list dialog to us as follows:

```
PS C:\ >  Set-PSSessionConfiguration Microsoft.Powershell –
ShowSecurityDescriptorUI
```

After executing this command, we will be prompted with a confirmation prompt. The prompt notifies us that the WinRM service will be restarted. Before issuing the command it is pertinent to realize that all remote or local sessions will be disconnected. After providing `Y` to the prompt we will then see the familiar Windows ACL dialog as shown in the following screenshot:


Permissions for http://schemas.microsoft.com/power...
http://schemas.microsoft.com/powershell/Microsoft.Powershell
Group or user names:
Add...
Remove
For special permissions or advanced settings, click Advanced.
Advanced
Learn about access control and permissions
OK
Cancel
Apply

Adam (DRISCOLL-DESK\Adam)
Administrators (DRISCOLL-DESK\Administrators)

Permissions for Adam	Allow	Deny
Full Control(All Operations)	☐	☐
Read(Get,Enumerate,Subscribe)	✓	☐
Write(Put,Delete,Create)	☐	☐
Execute(Invoke)	☐	☐
Special permissions	☐	☐

Notice the fine grained permissions that can be set for particular users and groups. Depending on the type of cmdlet, PowerShell can limit what the user has access to. They are broken down into three groups: **Read**, **Write**, and **Execute**. This dialog lets us specify which users or groups will have access to this particular session configuration or end point. This helps us to limit access to the machine via PowerShell remoting. Let's take it a step further and create a new session configuration that utilizes the new delegated administration capabilities available in PowerShell 3.0.

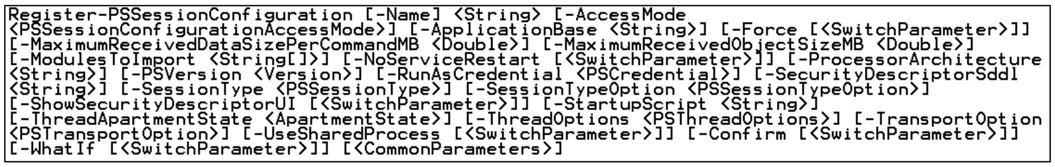
Creating a delegated session configuration

The cmdlet which is used for creating session configurations is the `Register-PSSessionConfrigurration` cmdlet. In order to run this cmdlet, we must run the PowerShell command prompt as an administrator. Although PowerShell will notify us before continuing, it is important to know the WinRM will be restarted after registering a new session configuration. This will cause any local or remote WinRM session to terminated.

Registering new session configurations

Let's take for instance, that certain users need to access a Team Foundation Build server. The services on the machine run as a trusted Build service user that only a few selected domain administrators know the credentials to. In order to provide access to a few developers that may need to access some resources on the build machine, without knowing the credentials to the build service account, we can set up a session configuration that delegates those users to run as the build service account.

For our example, we are trying to allow the `mdnvdi\developers` Active Directory group to run as the `mdnvdi\tfsbuilder` account. We will be configuring the session configuration to run in this manner. First, let's elevate a command prompt and examine the `Register-PSSessionConnfiguration` cmdlet parameter definition as follows:



```
Register-PSSessionConfiguration [-Name] <String> [-AccessMode
<PSSessionConfigurationAccessMode>] [-ApplicationBase <String>] [-Force [<SwitchParameter>]]
[-MaximumReceivedDataSizePerCommandMB <Double>] [-MaximumReceivedObjectSizeMB <Double>]
[-ModulesToImport <String[]>] [-NoServiceRestart [<SwitchParameter>]] [-ProcessorArchitecture
<String>] [-PSVersion <Version>] [-RunAsCredential <PSCredential>] [-SecurityDescriptorSddl
<String>] [-SessionType <PSSessionType>] [-SessionTypeOption <PSSessionTypeOption>]
[-ShowSecurityDescriptorUI [<SwitchParameter>]] [-StartupScript <String>]
[-ThreadApartmentState <ApartmentState>] [-ThreadOptions <PSThreadOptions>] [-TransportOption
<PSTransportOption>] [-UseSharedProcess [<SwitchParameter>]] [-Confirm [<SwitchParameter>]]
[-WhatIf [<SwitchParameter>]] [<CommonParameters>]
```

One side note is that there is actually a second parameter set, but it is intended for advanced developers looking to create session configurations using binary files. From the previous parameter definition we can see that there are a lot of things that can be configured for remote sessions. The most important parameters in our case are `SecurityDescriptorSddl`, `ShowSecurityDescriptorUI`, and `RunAsCredential`. The first two parameters define what user will be allowed to access our newly defined session configuration. The final parameter defines under which user the session configuration will execute.

To create our new configuration we can simply execute the following command:

```
PS C:\> Register-PSSessionConfiguration -Name TfsDevelopers
-RunAsCredential mdnvdi\tfsbuilder -ShowSecurityDescriptorUI
```

The command will ask us for the users that will be able to access the session configuration. From here, we can select the developers' group and give them full permissions. Next, the command will ask us for the credentials for the `mdnvdi\tfsbuilder` account. All this information will be stored securely in WinRM. The WinRM service will then be restarted.

Connecting to a newly registered session configuration

At this point, developers that are part of the previously specified group will be able to access this end point and run as the `tfsbuilder` account. This could be accomplished with the `Enter-PSSession` cmdlet as follows:

```
PS C:\> Enter-PSSession -ComptuerName mdnvdi-builder -ConfigurationName
TfsDevelopers -Credential mdnvdi\adriscoll
[mdnvdi-builder] PS C:\> whoami
Mdnvdi\tfsbuilder
```

In this command we connected to the `mdnvdi-builder` machine as the user `mdnvdi\adriscoll`. Once we were authenticated and connected we ran the `whoami` process which returns the user that we are running as. As you can see, we are running as the `mdnvdi\tfsbuilder` account. In the previous version of Windows PowerShell this would require additional and often very involved steps. Now the selected users will be able to execute the tasks under that delegated permissions of the `tfsbuilder` user. This type of scenario could be handy for all kinds of applications where a user may need a different set of permissions when connecting to a machine. In a different circumstance, it may be necessary to remove permissions from a user, rather than grant them. The `RunAsCredential` could be a user who did not have as much permission as the user who is authenticating against the end point.

In addition to running against a remote machine, it would be possible to set up session configurations for a local machine as well. This could allow users to accomplish the same types of tasks as we have seen but locally rather than remotely.

PowerShell session configurations are a very helpful but little known feature. Their usefulness has been expanded with the addition of delegated administration. As well adding this helpful feature, Microsoft solidified the behavior of remote sessions when dealing with real-world connection problems or client migration. The addition of robust and resilient sessions makes the work of an administrator much less frustrating.

Robust and resilient remote sessions

Remote sessions in PowerShell were a great feature but suffered from a few key issues that could cause an administrator issues when working in the field. Often, the goal of a remote session is to administer a machine which is physically not located within the office or even the general geographic location of the administrator. The further away the machine, the greater the chance that network related issues will cause connection failure. In the event of a connection failure, PowerShell does not behave as many interconnected technologies will. Rather than recovering, the session would merely be lost and any work would need to be recovered or repeated. Furthermore, an administrator can now disconnect and reconnect to a remote session. This allows for us to move from one machine to another while the session is still in progress on the remote machine.

A long running example

This could be helpful in long running job scenarios. To illustrate how this can be useful we will use an example of a script which is checking for a memory leak in a process. It runs over the course of ten minutes and gathers several memory statistics of the process to see if there is a gradual increase in overall usage over that time. The script waits two seconds between each reading and reports the output directly to the host. An administrator will evaluate the data after the process has completed. It is done as follows:

```
$stopTime = Get-Date
$stopTime.AddMinutes(10)
Start-Job -Name MemoryJob {
  while ((Get-Date) -lt $stopTime)
  {
 Get-Process BadService | Select-Object   PrivateMemorySize, WorkingSet,
VirtualMemorySize
 Start-Sleep -Seconds 2
 }
}
```

This script sets the `stopTime` variable to the current date and time returned by `Get-Date` and adds ten minutes to it. The loop checks to see if the current time is still less than the stop time. The `Start-Sleep` pauses the iteration of the loop for two seconds. The entire content of the loop is inside a local job because we want the script to return control to the console so that we can disconnect our session. The private memory size of `BadService`, working set, and virtual memory size will be written to the host. For the remainder of this example, we will assume that the previous script is saved in the file `$HOME\CheckMemoryUsage.ps1`.

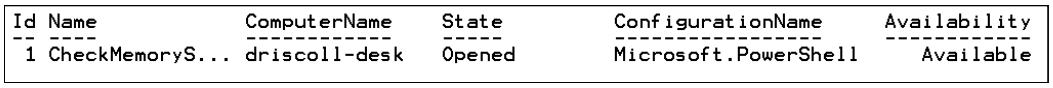
The different memory size metrics are useful for different reasons so getting all of them helps to tell a better story of memory usage of a process.

In our scenario, the script described previously will need to be run on a remote machine which has the suspect `BadService`. As the script takes ten minutes, it is expected that the administrator will not want to simply wait and watch the script as it executes. Before PowerShell 3.0, it would be necessary to keep the PowerShell command line open while this script was executing. Closing the console or experiencing even a slight network disconnection would result in the session being terminated and the script would need to be re-run.

Using resilient sessions

The first step to creating a resilient session is using the `New-PSSession` cmdlet to create a new named session. Without a name it will be difficult to locate the session we are attempting to reconnect in the future. This is done with the following command:

```
PS C:\> New-PSSession -ComputerName driscoll-desk -Name
CheckMemorySession
```



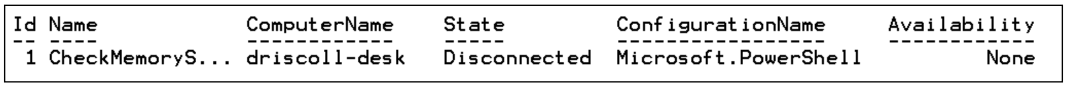
```
Id Name            ComputerName    State         ConfigurationName     Availability
-- ----            ------------    -----         -----------------     ------------
 1 CheckMemoryS... driscoll-desk   Opened        Microsoft.PowerShell     Available
```

Previously, you will have noticed that we are connected to `driscoll-desk` on the default `Microsoft.PowerShell` session configuration. Our session state is opened and we are ready to begin executing the script. We can retrieve our session and store it into a variable so that we can use it with `Invoke-Command` as follows:

```
PS C:\> $s = Get-PSSession -ComputerName driscoll-desk-Name
CheckMemorySession
PS C:\> Invoke-Command -Session $s -FilePath $Home\CheckMemoryUsage.ps1
```

As the script was designed in a way that it runs asynchronously, the previous command will return right away. At this point the script is running on the remote `driscoll-desk` machine. We can now experiment with disconnecting the session and connecting from another PowerShell console. To disconnect a session explicitly, we use the `Disconnect-PSSession` cmdlet as follows, which gives the output as shown in the following screenshot:

```
PS C:\> Disconnect-PSSession $s
```



```
Id Name            ComputerName    State         ConfigurationName     Availability
-- ----            ------------    -----         -----------------     ------------
 1 CheckMemoryS... driscoll-desk   Disconnected  Microsoft.PowerShell          None
```

Our session is now disconnected, but is still running on the remote host. We can now exit the console completely. After exiting the console, we can now re-open the PowerShell console. The first thing we can check after launching a new console is whether the remote machine is still running our remote session. This is done using following command:

```
PS C:\> Get-PSSession -ComputerName driscoll-desk -Name
CheckMemorySession
```

Notice that we are in the same state, in which we were when we first disconnected from the session in the previous console. We now have the opportunity to reconnect to the session using the `Connect-PSSession` cmdlet as follows:

```
PS C:\> Connect-PSSession ComputerName driscoll-desk  -Name
CheckMemorySession
```

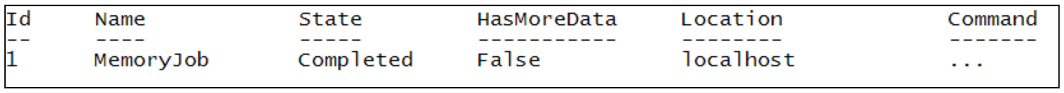
This reconnects the session with computer driscoll-desk.

Once the connection is established, we are free to interact with the session as if it was a newly created session within our console. The ability to execute commands with `Invoke-Command` or enter the session explicitly using `Enter-PSSession` both behave exactly the same as they did in the previous version of PowerShell. For example, we can now evaluate the progress of the job we started in the last PowerShell console as follows:

```
PS C:\> Enter-PSSession -Id 6
```

```
[driscoll-desk] PS C:\> Get-Job
```

This gives us the following output:



```
Id     Name            State         HasMoreData     Location             Command
--     ----            -----         -----------     --------             -------
1      MemoryJob       Completed     False           localhost            ...
```

The job has completed and we could now utilize the data stored within it to evaluate whether or not the process we were monitoring was in fact leaking memory. Once we have finished using our session we need to remove it using the following command:

```
[driscoll-desk] PS C:\> Exit-PSSession
```

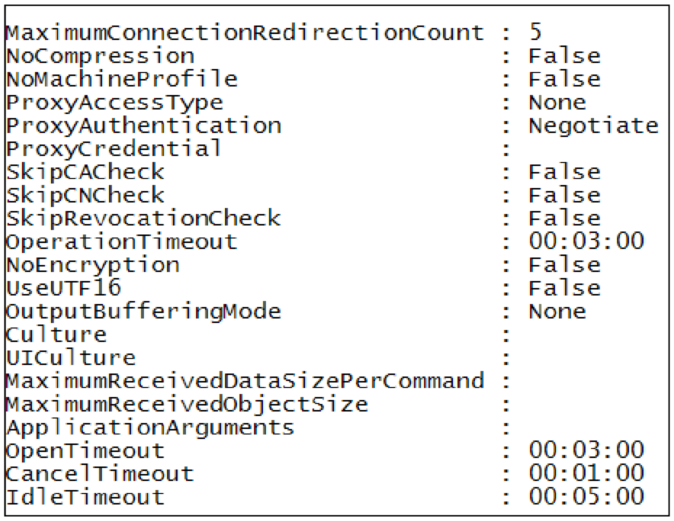
```
PS C:\> Remove-PSSession -ComputerName driscoll-desk -Name
CheckMemorySession
```

Once the session has been removed we can no longer connect to it. The data that was stored in the remote job is lost as well, along with the session as follows:

```
PS C:\> Get-PSSession -ComputerName driscoll-desk
```

Notice that in this `Get-PSSession` command the `CheckMemorySession` is no longer listed. Using the `Remove-PSSession` cmdlet is not the only way a session will be recycled on a remote machine. There is an idle timeout setting too, which is set on a remote session, so that orphaned sessions are not left behind. By default, the idle timeout is set to four minutes. To adjust this time out value we can use the `New-PSSessionOption` cmdlet. The `IdleTimeout` parameter accepts milliseconds as follows:

```
PS C:\> New-PSSessionOption -IdleTimeout 300000
```



```
MaximumConnectionRedirectionCount : 5
NoCompression                     : False
NoMachineProfile                  : False
ProxyAccessType                   : None
ProxyAuthentication               : Negotiate
ProxyCredential                   :
SkipCACheck                       : False
SkipCNCheck                       : False
SkipRevocationCheck               : False
OperationTimeout                  : 00:03:00
NoEncryption                      : False
UseUTF16                          : False
OutputBufferingMode               : None
Culture                           :
UICulture                         :
MaximumReceivedDataSizePerCommand :
MaximumReceivedObjectSize         :
ApplicationArguments              :
OpenTimeout                       : 00:03:00
CancelTimeout                     : 00:01:00
IdleTimeout                       : 00:05:00
```

Notice that the final value for the session option is the `IdleTimeout` and it is set to five minutes or 300,000 milliseconds. To use the command effectively we would need to store the session option in a variable and pass it to the `New-PSSession` cmdlet as follows:

```
PS C:\> $o = New-PSSessionOption -IdleTimeout 300000
PS C:\> New-PSSession -ComputerName badservice-host -Name
CheckMemorySession -SessionOption $o
```

Additionally, the default session option can be set using the `PSSessionOption` preference variable, as shown in the following commands:

```
PS C:\> $PSSessionOption.IdleTimeout = 300000
PS C:\> New-PSSession -ComputerName driscoll-desk -Name
CheckMemorySession
```

The `PSSessionOption` variable value will be used whenever a new session is created or entered and the session option is not explicitly specified.

We can even use the new inline syntax to define the parameters for the session options. This syntax utilizes a hash table to define the property values as follows:

```
PS C:\> New-PSSession -ComputerName driscoll-desk -SessionOption @
{NoMachineProfile=$true; idleTimeout="00:05:00"}
```

Note that the remote computer could specify an `IdleTimeout` value as well. If this value is specified, the value that is lesser will be used.

The real benefit of this feature is not in the ability to interact with a single machine, as we have just experimented with, but instead to interact with many machines. Rather than having to manage many connected sessions, and worrying about the state of that connection, it is possible for an administrator to simply connect, start a task, disconnect, and check on the result of that task at a later time from another machine. This ensures a much better level of scalability of remote PowerShell interaction.

Experimenting with resilient sessions

In addition to the carefully designed disconnected architecture that was implemented, Microsoft took steps towards ensuring that the WinRM protocol behaves much better under unexpected connection issues as well. Rather than losing a session to a bad network, a session can be reconnected after failure. PowerShell will even attempt to reconnect automatically. This can best be shown with an example. The Windows PowerShell release notes have by far the simplest and most effective example available. It demonstrates exactly how the PowerShell console behaves.

The example requires two PowerShell consoles to be running. The first console is responsible for starting a remote session with the local machine. The second console will remove the WinRM end point which the first console is connecting to. This simulates a loss in connectivity.

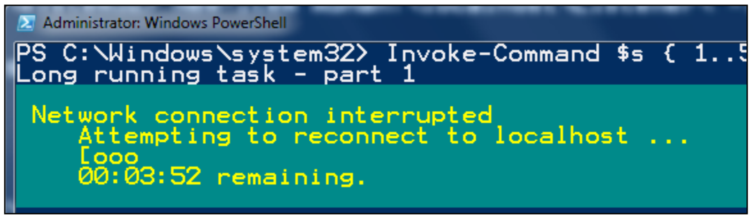
First start two PowerShell consoles. We can refer to them as console one and console two. Console one will start the remote operation and console two will disconnect that operation. Execute the following commands in console one. The console will begin listing the iteration number that it is currently on. Console one will block up to 500 seconds as follows:

```
PS C:\> $s = New-PSSession -ComputerName localhost
PS C:\> Invoke-Command $s { 1..500 | % {echo "Long running task - part
$_"; sleep 1} }
```

Now, in console two, execute the following command, causing console two to disconnect:

```
PS C:\> Remove-Item WSMan:\localhost\Listener\* -Recurse
```

As we switch back to console one we can see that the console has lost connection to its remote session. The PowerShell console provides a convenient count down timer that displays how much time before the session will be marked in a disconnected state. The session will not be removed and any data processed during the session will still be available until the session is restored. This is shown in the following screenshot:

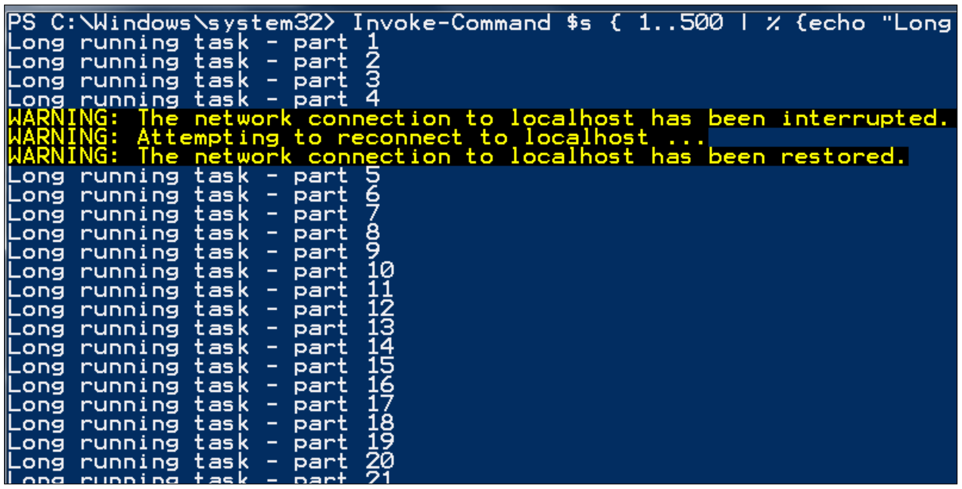


```
PS C:\Windows\system32> Invoke-Command $s { 1..5
Long running task - part 1

 Network connection interrupted
    Attempting to reconnect to localhost ...
    [ooo
    00:03:52 remaining.
```

Now move back into console two and type the following command:

```
New-Item WSMan:\localhost\Listener\ -Transport HTTP -Address * -Force
```

The command recreates the end point which we have previously removed. This allows console one to continue to run. One interesting note is that console one will output all the parts that were completed from the time we disconnected until the time we were reconnected. The following screenshot explains this:



```
PS C:\Windows\system32> Invoke-Command $s { 1..500 | % {echo "Long
Long running task - part 1
Long running task - part 2
Long running task - part 3
Long running task - part 4
WARNING: The network connection to localhost has been interrupted.
WARNING: Attempting to reconnect to localhost ...
WARNING: The network connection to localhost has been restored.
Long running task - part 5
Long running task - part 6
Long running task - part 7
Long running task - part 8
Long running task - part 9
Long running task - part 10
Long running task - part 11
Long running task - part 12
Long running task - part 13
Long running task - part 14
Long running task - part 15
Long running task - part 16
Long running task - part 17
Long running task - part 18
Long running task - part 19
Long running task - part 20
Long running task - part 21
```

Rather than the session being terminated on the server end, the session continued to run, even though the client lost connection. This is a vast improvement from the previous version of PowerShell. In environments where spare connectivity is an everyday issue, this functionality will make PowerShell a much more viable resource.

Improved Restart-Computer

The `Restart-Computer` cmdlet is very handy. Restarting a whole host of computers can be a big pain but using this cmdlet makes it much easier. Typically, a computer restart is a side effect of another action. We simply do not restart computers for fun. Rather, a piece of software has been installed or a Windows update has been applied. Although the cmdlet is very useful, it did have a few caveats in the previous version of PowerShell. The main issue was that there was no way to ensure, that the computer was running again without constantly attempting to ping the machine using a mechanism as follows:

```
PS C:\> Restart-Computer $serverName
PS C:\> while (-not (Test-Connection –ComputerName $serverName –Quiet))
{Start-Sleep 10}
```

These commands will restart the named computer and ping it every ten seconds until it responds. Once the computer responds to the ping, the script will continue on. This excess scripting made this cmdlet a bit of a pain to use in actual scripts. Additionally, the `Restart-Computer` cmdlet can be used quite often by administrators, so having to deal with this every time is annoying.

The enhanced Restart-Computer

Rather than using this mechanism, we can now take advantage of the `Wait` parameter that has been added to the `Restart-Computer` cmdlet. Rather than having to write any additional waiting logic, we can just specify the `Wait` parameter as follows:

```
PS C:\> Restart-Computer $serverName –Wait
```

The execution of the script will be blocked until the `Restart-Computer` cmdlet returns from its restart operation. This makes it much more practical to utilize the cmdlet in your scripts. Although it helps all administrators the most, administrators new to PowerShell will find this parameter the most helpful. It may be very difficult to come up with the waiting logic correctly without knowing a lot more about PowerShell.

The length and type of `Wait` can be specified with a few additional parameters that have been added. The amount of time to wait for the computer to restart, before considering the operation timed out can be specified using the `Timeout` parameter. This parameter accepts the number of seconds to wait. The next restart will wait up to five minutes, or 300 seconds, for the computer to restart as follows:

```
PS C:\> Restart-Computer $serverName –Wait –Timeout 300
```

The logic which we developed ourselves relied on the network to be available rather than particular services. This could cause issues if we typed to invoke PowerShell remotely on the server which we are restarting. Although the logic would stop waiting the WinRM service may not be ready to accept our request for a remote session. This could cause the rest of the script to fail. In the following script the `Invoke-Command` call would most likely fail as although the network is ready, the WinRM service is most likely not.

```
PS C:\> Restart-Computer $serverName
PS C:\> while (-not (Test-Connection -ComputerName $serverName -Quiet))
{Start-Sleep 10}
PS C:\> Invoke-Command { Get-Service } -ComputerName $serverName
```

To alleviate this issue the `Restart-Computer` cmdlet has the addition of the `For` parameter. It accepts three values: `WinRM`, `WMI`, and `PowerShell`. Specifying this parameter with one of the stated values will cause the `Restart-Computer` cmdlet to wait for that particular feature to become ready on the target machine before releasing the WinRM service from the wait. If `For` is not specified, the default is PowerShell. It is done as follows:

```
PS C:\> Restart-Computer $serverName -Wait -For WinRM
```

This script will wait for `WinRM` to be ready on the target machine indefinitely. This can be very handy if we only need to access a particular component such as WMI.

Installing a product which requires a restart

For example, if we were interested in ensuring that the installation of an MSI package was successful after the computer restarts, we could wait for only WMI to become ready rather than the entire WimRM infrastructure. It is done by using the following commands:

```
PS C:\> Invoke-Command { Start-Process "MSIExec" -ArgumentList " /s /qn
C:\MyProduct.msi" -Wait } -ComputerName $serverName
PS C:\> Restart-Computer $serverName -Wait -For WMI
PS C:\> Get-WmiObject -Class Win32_Product -Filter "Name='MyProduct'" -
ComputerName $serverName
```

In this script we start `MSIExec`, which is the application responsible for installing MSI packages, on the remote machine with the flags for a silent installation of the MSI package `MyProduct.msi`. The `Start-Process` command will wait for the installation to finish. After it is completed, we restart the computer and wait for WMI to become ready after the restart. Once restarted, we query WMI to ensure that `MyProduct` is correctly registered in the installed products on the target machine. The `Restart-Computer` cmdlet takes care of all the pinging of the remote machine to check to see if WMI is ready. Because the `Restart-Computer` cmdlet must constantly ping the remote machine until it is available, Microsoft exposed another related parameter which allows this functionality to be customized.

Using the delay parameter

The `Delay` parameter is used to specify the amount of time taken to make remote machine ready. The value is given in seconds as follows. It defaults to five:

```
PS C:\> Restart-Computer $serverName –Wait –For WinRM –Delay 10
```

This command will query the remote machine's readiness of WinRM every ten seconds until it is ready.

Using different communication protocols

In addition to the new `Wait` parameter Microsoft changed how the `Restart-Computer` cmdlet works internally. In the past it relied solely on the **Distributed Component Object Module** (**DCOM**), to communicate to the remote computers. In many circumstances DCOM is blocked by firewalls, as it can be seen as a security concern. This prevented the `Restart-Computer` cmdlet from working for machines blocking the DCOM ports. To alleviate some of the issues with this, the ability to use the `Restart-Computer` across the WSMan protocol was added. The WSMan protocol is used by the WinRM service and handles PowerShell remoting.

Several new parameters have been added to allow for this. The first is the `Protocol` parameter. This parameter changes the protocol which is used by the cmdlet. We can either specify WSMan or DCOM as follows. DCOM is the default value:

```
PS C:\> Restart-Computer $serverName –Protocol WSMan
```

In addition to being able to change the protocol used, we can specify the type of authentication too used for each type of protocol.

WSMan supports the following parameters: `Default`, `Basic`, `Negotiate`, `CredSSP`, `Digest`, and `Kerberos`.

DCOM supports the following parameters `Default`, `None`, `Connect`, `Call`, `Packet`, `PacketIntegrity`, `PacketPrivacy`, and `Unchanged`.

To change the authentication types, use either the `WSManAuthentication` or `DCOMAuthentication` parameters now available on the `Restart-Computer` cmdlet as follows:

```
PS C:\> Restart-Computer $serverName -Protocol WSMan -WSManAuthentication
CredSSP
```

Summary

In this chapter we learned that the new features within the latest version of PowerShell, had greatly extended and simplified the day to day tasks of an administrator. The scheduled jobs feature made it much easier to setup repeatable tasks so that less time is spent having to repeat the automation. It instead automated the automation, leaving us to spend more time working on more important tasks. The added delegated administration system allowed administrators with less permission to run as different users. This allowed better control or who has access to which resources and enabled more people to take care of tasks that they might not have been able to do before. Finally, the improvements to the `Restart-Computer` cmdlet made it much easier to work correctly into scripts and work with network infrastructure. This made the cmdlet much more practical to use in scripts.

In the next chapter we will look at the new Windows Workflow for PowerShell and how it takes scripts to an entirely new level.

4 Windows Workflow in PowerShell

Scripting and programming can be difficult. Understanding the number of requirements for developing any piece of software can be a tremendous task to undertake. Typically, developers break down the larger pieces of the puzzle into manageable chunks, hoping that the chunks will fit together in the end. Often, these components may be poorly designed, not meet requirements, or are very rigid. As programming and scripting languages advance, more and more of the implementation details are pushed onto the computer and more of the business rule development is assigned to the application developer. PowerShell and programming languages such as C# provide a lot of benefits over some of the lower level languages that require much more care, when it comes to memory and state management.

Microsoft has started taking steps to move beyond programming as we know it. Rather than having to deal with code, they are starting to take a much more visual approach with products such as Visual Studio LightSwitch and Windows Workflow. The latter product enables developers, designers, or even product managers to develop product workflow through a visual designer, requiring little to no actual programming, in a traditional sense. Rather than working with `if` blocks and `while` loops, they worry more about what happens if a user cannot login to their desktop.

Thinking in business rules rather than source code has many advantages. It attacks the true need for the software. Rather than figuring out how to complete a task, we look at what the outcome of that task should be. In PowerShell 3.0, Microsoft added the ability to harness Windows Workflow. Now it is possible to create re-usable workflows in script that can expose higher level business logic, rather than merely a function or module.

In this chapter we will cover the following:

- An overview of Windows Workflow
- How to integrate Windows Workflow into PowerShell
- The limitations of Windows Workflow in PowerShell
- Working with custom activities and activities in Visual Studio

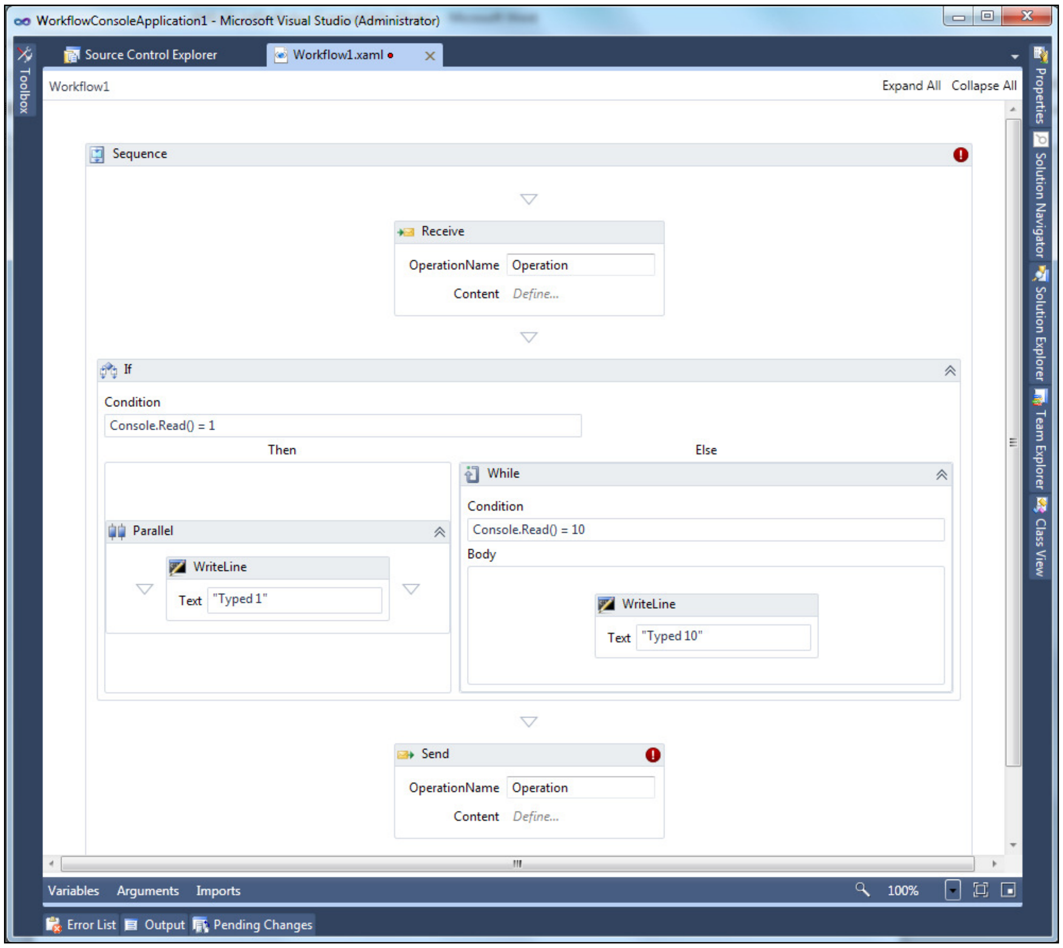
Understanding Windows Workflow

Windows Workflow is a vast product. It encompasses many other products released by Microsoft such as Visual Basic .NET, Visual Studio, and Microsoft .NET. Workflows can be created that do any number of tasks. Each individual task defined in a workflow is referred to as an **Activity**. Activities can be binary, written in languages such as C#, or can be composed of other activities. With the ability to build activities on top of each other it enables developers to create complex systems that seem to serve a single purpose and are re-usable.

The main goal of Windows Workflow is to provide:

- Scalable solutions that can be easily run sequentially or in parallel
- Complex activities that are repeatable, parallelizable, interruptible, and recoverable

The primary way to define a workflow typically has been with Visual Studio. Visual Studio offers a visual editor designed to edit a workflow definition, similar to a flow chart. There are activities for making choices, performing actions, or even running tasks in parallel. Each of these activities can be dragged onto the designer's canvas and into the workflow. This visual type of feedback makes it much easier to understand. This is explained in the following screenshot:



OperationName Operation
Content Define...

Condition
Console.Read() = 1

Text "Typed 1"

Condition
Console.Read() = 10

Text "Typed 10"

OperationName Operation
Content Define...

A **Windows Workflow** is defined as a large **Extensible Application Markup Language** (**XAML)** file. XAML is very similar to the XML syntax. As a workflow is defined in XAML before it is compiled, it offers a standard language which all workflows speak rather than the multitude of programming languages that are available.

In addition to creating workflows in Visual Studio, we can create custom workflow designers for specialized users as well. Workflow designers for product managers or even end users could be developed to create custom workflows that integrate directly into a product. Workflows can then be compiled and hosted in a .NET application for use just like any other source code-written component.

Microsoft's goal with Windows Workflow, is to create business logic easily that is re-usable, repeatable, parallelized, restartable, stoppable, interruptible, and frequent. Exposing these types of characteristics natively makes workflows ideal for use in business and IT administration. This is one of the main reasons that workflow was built into Windows PowerShell 3.0.

Integrating Windows Workflow with PowerShell

PowerShell has always been about simplicity in much the same way as workflow. Being able to intertwine the two together is a natural progression. Windows PowerShell workflow integration offers a lot of benefits in terms of re-usability. Users can create and re-use workflows from Visual Studio directly in PowerShell. This allows many of the pre-existing workflows to be consumed by PowerShell, offering a host of new, higher level functionality which was not found in PowerShell before.

Additionally, workflows created in PowerShell can be used by the Visual Studio Workflow designer. This alleviates much of the code necessary for hosting the PowerShell runtime and running commands. Higher level scripts can be developed into workflows to accomplish larger tasks and simply dropped into an existing Visual Studio Workflow.

By design, PowerShell workflows expose numerous standard parameters that allow for remote execution, parallelization (jobs), and data persistence. The ability to harness these types of features without having to write any additional logic is immensely beneficial. This allows for the workflow to be run almost anywhere. This is what makes workflows so re-usable.

First steps in workflows

The first step into Windows Workflow in PowerShell is to begin defining a workflow. A workflow looks very similar to a function but has some very different traits that must be taken into account to use the workflow concept correctly. To define a workflow, we can use the new `workflow` keyword, followed by a name and a script block as follows:

```
PS C:\> workflow Invoke-String { "String" }
```

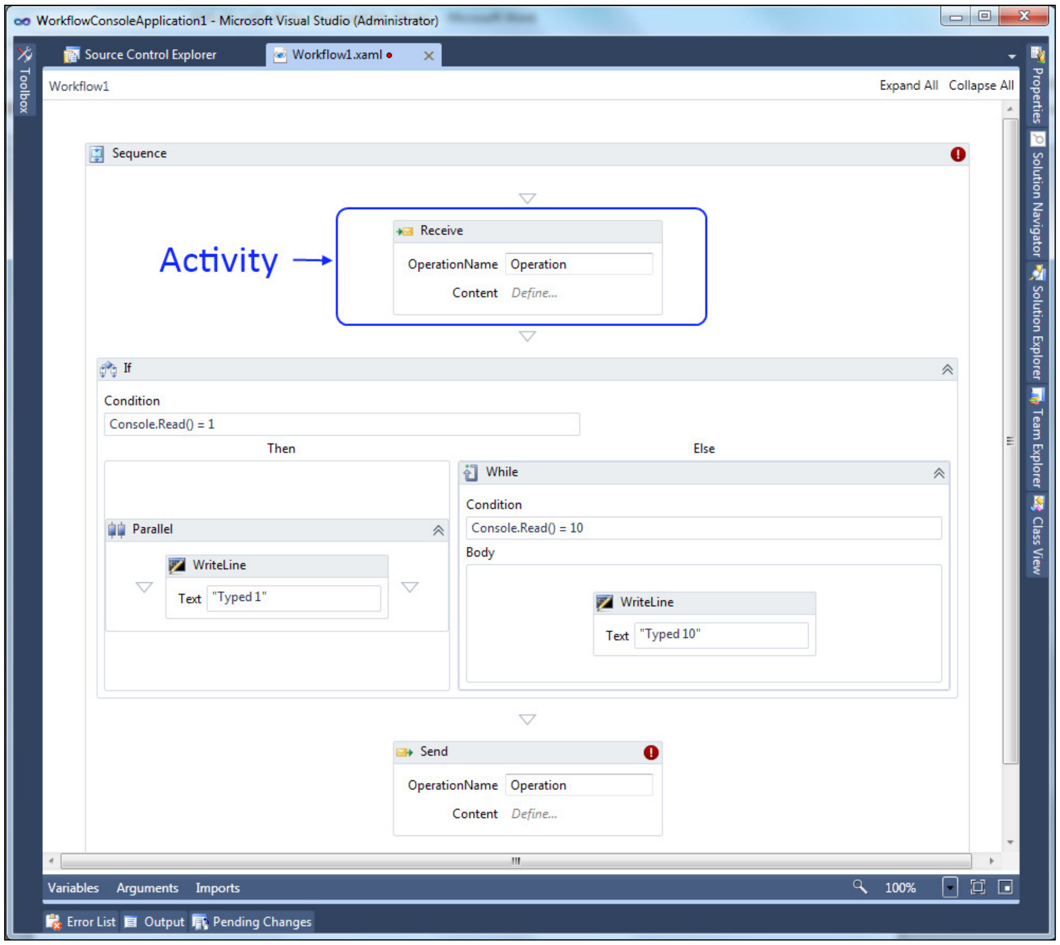
Running this command will define the workflow in our session. Now it is possible to execute the workflow, just like we would execute a cmdlet or function as follows:

```
PS C:\ > Invoke-String
String
```

Running the workflow behaves much like running a command would, aside from some noticeable speed differences. The speed of execution is due to the fact that the workflow is actually compiled when we define it and must be loaded the first time we run it. We can see that, that command is in fact a workflow by using `Get-Command` as follows:

```
PS C:\ > get-command Invoke-String
Capability      Name
----------      ----
Workflow        Invoke-String
```

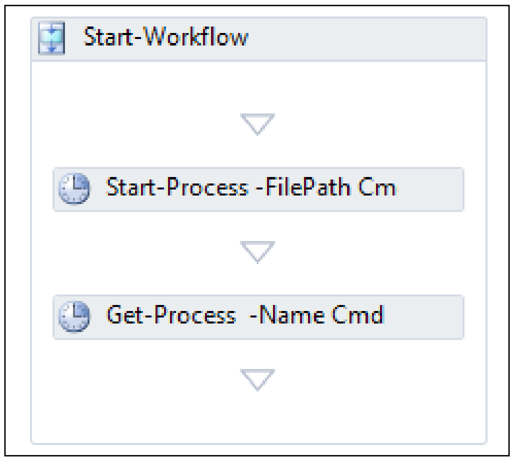
Now that we have created a basic workflow, let's take a few minutes to better understand the concept of a workflow. As PowerShell uses the same workflow system like the Visual Studio workflow designer, it is good to compare the two representations to see how they fit together. In a visual workflow, activities are dragged onto the designer surface to add an individual piece of logic or activity. Each of these activities can take parameters and provide output data. This is explained in the following screenshot:



In PowerShell, activites are broken up by the commands within the workflow. Each command thus becoming its own activity. Let's take for instance the following workflow activity defined in PowerShell. The `Start-Workflow` is the outer activity containing the inner two activites for starting and getting the `Cmd` process. Each of the activities runs independently of the other. Just like the visual designer, the `Start-Process` and `Get-Process` activities accept parameters. We can see this in the following example:

```
PS C:\> Workflow Start-Workflow {
 #Activity 1
 Start-Process -FilePath Cmd
 #Activity 2
 Get-Process  -Name Cmd
}
```

The visual representation of the PowerShell workflow would look as shown in the following screenshot:



The `Start-Process` flows into the `Get-Process`. The `Start-Workflow` is an over arching sequence that contains the two activities. Taking a step back to understand the overall concept is necessary before moving forward, because the idea of a workflow is much different than a regular PowerShell script. As we move forward, we will examine these differences and find the similarities between a regular PowerShell script.

Activites are independent of each other! They are in their own scope and thus cannot even share variables amongst each other. Think of each line, or activity as its own entity.

Workflow common parameters

If we run the `Get-Help` cmdlet against the `Invoke-String` workflow activity we defined earlier, we can evaluate the syntax for the command. In addition to the standard common parameters found with all cmdlets, we find another set of parameters as well. These are the **workflow common parameters**. Refer to the following commands:

```
PS C:\Users\Adam> Get-help Help Invoke-String
NAME
  Invoke-String
SYNTAX
  Invoke-String [<WorkflowCommonParameters>] [<CommonParameters>]
```

The workflow common parameters expose much of the functionality which is expected in all workflows. These types of properties were described earlier, such as parallelism and data persistence. Although there is a significant list of workflow common parameters, some of the most important ones are listed as follows. Each of these parameters will significantly alter how the workflow is run.

- `PSComputerName`: This is a list of computer names to run the workflow against.
- `PSPersist`: This forces the workflow to checkpoint the workflow state and data after each activity.
- `AsJob`: It runs the workflow as an asynchronous job on the local computer.

These listed parameters give a good feel as to some of the capabilities of the workflow. First, it can be run remotely on any number of computers. Secondly, it can persist state and data between activities. Finally, it can be run asynchronously on the local computer. Each of these traits makes a workflow much more robust than a typical job or script.

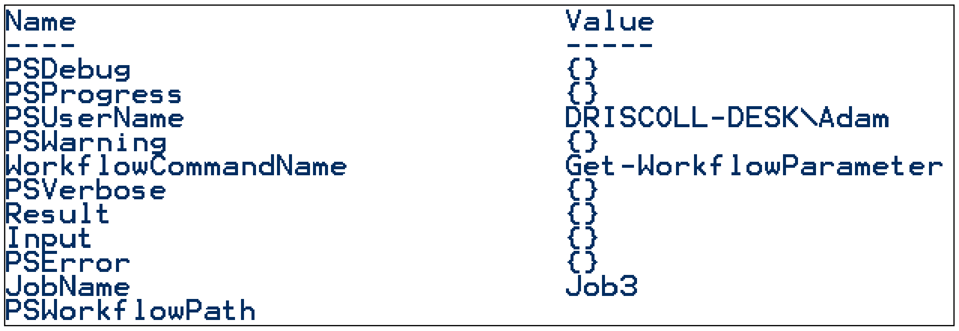
In addition to altering how the workflow is run, we can access the different common workflow parameters from within a workflow. There are several different ways of doing this. To simply access a common workflow variable we can reference it much like any other parameter found in a regular script or function. For example, if we want to access the name provided to the `PSComputerName` parameter, we could use the `$PSComputerName` variable from within the workflow as follows:

```
PS C:\> workflow Get-ComputerName {
"ComputerName: $PSComputerName"
}
PS C:\> Get-ComputerName -PSComputerName localhost
ComputerName: localhost
```

In order to iterate or modify the workflow common parameters we can use the `Get-PSWorkflowData` and `Set-PSWorkflowData` activities. The `Get-PSWorkflowData` command is a generic activity and it is required that we specify the type we are looking to return. If, for example, we wanted to return the list of parameters, we could use the following workflow:

```
PS C:\> workflow Get-WorkflowParameter {
Get-PSWorkflowData[Hashtable] -VariableToRetrieve All
}
PS C:\> Get-WorkflowParameter
```

This will show us the list as shown in the following screenshot:

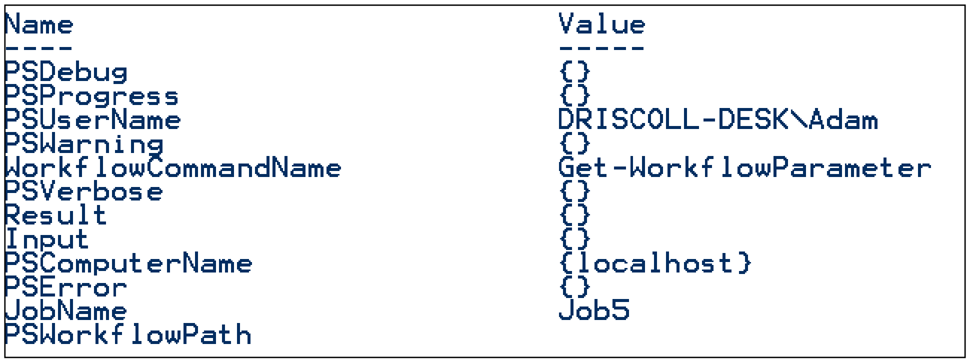


```
Name                           Value
----                           -----
PSDebug                        {}
PSProgress                     {}
PSUserName                     DRISCOLL-DESK\Adam
PSWarning                      {}
WorkflowCommandName            Get-WorkflowParameter
PSVerbose                      {}
Result                         {}
Input                          {}
PSError                        {}
JobName                        Job3
PSWorkflowPath
```

Only the parameters that are currently being used are returned for the activity. We will not see the computer name or persist value unless they are specified. If we run the same command again but with the use of the `PSComputerName` parameter we will see it in the hash table that is returned as follows:

```
PS C:\> Get-WorkflowParameter -PSComputerName localhost
```

The hash table is shown in the following screenshot:



```
Name                           Value
----                           -----
PSDebug                        {}
PSProgress                     {}
PSUserName                     DRISCOLL-DESK\Adam
PSWarning                      {}
WorkflowCommandName            Get-WorkflowParameter
PSVerbose                      {}
Result                         {}
Input                          {}
PSComputerName                 {localhost}
PSError                        {}
JobName                        Job5
PSWorkflowPath
```

The `All` value for the `VariableToRetrieve` is a keyword. To retrieve another parameter we can specify that parameter name along with the type specifier for a string array as follows. The name of the parameter is case sensitive:

```
PS C:\> workflow Get-WorkflowParameter {
Get-PSWorkflowData[string[]] -VaraibleToRetrieve       PSComputerName
}
PS C:\> Get-WorkflowParameter -PSComputerName localhost
Localhost
```

Finally, it is also possible to set the parameters from within the workflow. To do this, we will use the Set-PSWorkflowData.

```
PS C:\> workflow Get-WorkflowParameter {
Set-PSWorkflowData -PSComputerName driscoll-host
Get-PSWorkflowData[String[]] -VariableToRetrieve PSComputerName
}
PS C:\> Get-WorkflowParameter -PSComputerName localhost
Driscoll-host
```

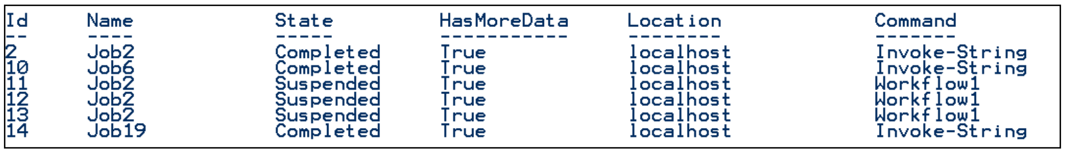
Pay attention to the use of the full parameter names in cmdlets throughout this chapter. Positional parameters are not supported in workflows.

Workflow as a job

Jobs are really helpful when attempting to run a long running activity or multiple activities at once. Executing a workflow as a job, is just as easy as executing a script block as a job using `Invoke-Command`. When running a workflow as a job on the local computer, we can use all the job cmdlets for interacting with the workflow job as follows:

```
PS C:\Users\Adam> $StringJob = Invoke-String -AsJob
PS C:\Users\Adam> Get-Job

PS C:\Users\Adam> Receive-Job $StringJob
String
```

This produces the result as shown in the following screenshot:



```
Id     Name          State       HasMoreData    Location         Command
--     ----          -----       -----------    --------         -------
2      Job2          Completed   True           localhost        Invoke-String
10     Job6          Completed   True           localhost        Invoke-String
11     Job2          Suspended   True           localhost        Workflow1
12     Job2          Suspended   True           localhost        Workflow1
13     Job2          Suspended   True           localhost        Workflow1
14     Job19         Completed   True           localhost        Invoke-String
```

Remote execution

The workflow can be run on a remote computer using the `PSComputerName` parameter as well. There are several other parameters that are used with the `PSComputerName` parameter that allow us to utilize different authentication mechanisms, ports, and even session configuration end points. To run our simple `Invoke-String` workflow on another computer we could do the following:

```
PS C:\> Invoke-String -PSComputerName $serverName
String
```

Just like an `Invoke-Command` call, the workflow returns the string back to the local command window. There are several common workflow parameters that can be used to modify how the workflow activity will connect to a remote machine. Some of these parameters include `PSAuthentication`, `PSUseSSL`, and `PSPort`. In addition to being able to specify these types of parameters, it is possible to specify a session configuration as well. A workflow PowerShell session configuration is very similar to a regular remoting end point.

We still use the `Register-PSSessionConfiguration` cmdlet to create the new session configuration. The only difference is that we need to use the new `SessionType` parameter to specify that the session configuration is a workflow session type. In addition to being able to create an end point that is designated for workflows, we can configure that end point with several workflow specific configuration options. To create a new session configuration option for a workflow we use the `New-PSWorkflowExecutionOption` cmdlet.

Some of the interesting options available for this cmdlet include the maximum size of the persistence store, the maximum number of concurrently executing workflows, and maximum suspended workflows. To create a simple session configuration we could do the following:

```
PS C:\> $Option = New-PSWorkflowExecutionOption -MaxRunningWorkflows 5 -
MaxPersistenceStoreSizeGB 10 -MaxSuspendedWorkflows 10

PS C:\> Register-PSSessionConfiguration -Name Driscoll.Workflow -
SessionType Workflow -SessionTypeOption $Option
```

Custom workflow parameters

Workflows, much like functions or scripts, can use the `param` keyword to define parameters that are accepted by the workflow. The use of a type specifier supported just as in functions and naming the variable will expose the variable as a parameter on the workflow. The syntax looks almost identical to that of a function using the following commands:

```
PS C:\> workflow Write-String {
  param([string]$string)
  "String to write is: $string"
}
```

In addtion, very similar to a function, getting the help for the workflow will show that the workflow exposes the parameter defined by the `param` keyword. This means that there is tab completion available as well, for the parameter on the workflow. This makes it very easy to discover which additonal parameters a workflow supports as follows:

```
PS C:\> Get-Help Write-String
NAME
  Write-String
SYNTAX
  Write-String [[-input] <string>] [<WorkflowCommonParameters>]
```

The `param` block supports the `Parameter` attribute as well, which can be used to adorn parameters of advanced functions. For example, we could use it to mark a particular parameter as mandatory as follows:

```
PS C:\> workflow Write-String {
  param(
    [Parameter(Mandatory)]
    [string]$input)
  "String to write is: $input"
}
```

Scripts in workflows

In order to make a workflow behave more like a regular PowerShell script, we can use the `inlineScript` keyword. The `inlineScript` keyword executes everything within the block just as it would a regular PowerShell script. This is done using the following commands:

```
PS C:\> workflow Invoke-PowerShellScript {
 inlineScript {
  $Process = Start-Process Notepad
  Get-Process $Process
 }
}
```

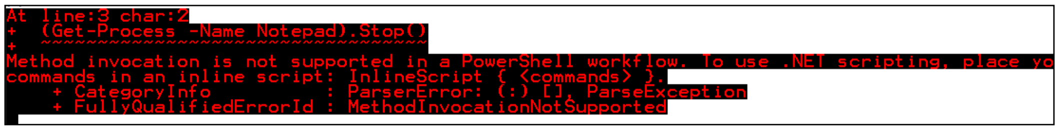
In this inline script we can see that the script is not following the rules of workflow. When using workflow activities, like `Start-Process`, outside of an inline script they support neither positional parameters nor dynamic variable scoping. In contrast, the inline script behaves more like PowerShell and allows for these types of language features. Even though this is the case, running this workflow will behave as it is written. The notepad process will be started and the process information will be returned to the console.

> Note that inline scripts, like workflows in general, do not support user interaction. Using commands like `Write-Host` will fail anywhere within a workflow; including inline scripts.

The inline scripts can be helpful for other reasons as well. One of the very important reasons is the ability to call `.NET` methods. It is not possible to call a `.NET` method within the regular scope of a workflow. On the other hand, this is supported by the inline script block as follows:

```
PS C:\> workflow Stop-Notepad {
 Start-process -FilePath notepad
 (Get-Process -Name Notepad).Stop()
 }
```

This produces the following output:



```
At line:3 char:2
+  (Get-Process -Name Notepad).Stop()
+  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Method invocation is not supported in a PowerShell workflow. To use .NET scripting, place yo
commands in an inline script: InlineScript { <commands> }.
    + CategoryInfo          : ParserError: (:) [], ParseException
    + FullyQualifiedErrorId : MethodInvocationNotSupported
```

In order to call the .NET method we need to place the call to stop within an inline script as follows:

```
PS C:\> workflow Stop-Notepad {
 Start-process -FilePath notepad
  inlineScript {
    (Get-Process –Name Notepad).Stop()
  }
 }
```

Working between workflows

Just like with the visual workflow designer it is possible to allow one workflow to call another workflow. This is simply done by calling the workflow just like you would from the command line within the other workflow using the following commands. The workflow must be defined before it is called:

```
Workflow Invoke-Jump {
 "Jumps"
}
Workflow Invoke-Johnny {
 "Johnny"
 Invoke-Jump
}
PS C:\> Invoke-Johnny Jumps
```

If the `Invoke-Jump` workflow were to change, the `Invoke-Johnny` workflow would remain the same. Johnny would still jump. This is because the activities are directly referenced when the activity is defined. Thus, `Invoke-Jump` becomes part of `Invoke-Johnny`. If we wanted Johnny to run rather than jump, we would need to redefine `Invoke-Johnny`.

It is important to note that workflows nested within another workflow cannot contain a `param` block as well. This is because the `param` block is not natively supported by workflow and a wrapped function is placed around the outermost workflow. The following error would be shown if we tried to nest a workflow which contained a `param` block in another workflow:

```
Unable to find type [Mandatory]: make sure that the assembly containing
this type is loaded.
```

In addition to merely referencing other workflows, workflows can be defined within a workflow. When this is the case, the inner workflow is not visible to anyone outside of the parent workflow. To nest a workflow, use the workflow keyword with the script block of another workflow using the following commands:

```
Workflow Invoke-TopLevel {
 "This is the top level"
 Workflow Invoke-LevelOne{
  "This is level one"
  Workflow Invoke-LevelTwo {
   "This is level two"
  }
  Invoke-LevelTwo
}
Invoke-LevelOne
}
PS C:\> Invoke-TopLevel
This is the top level
This is level one
This is level two
```

If we were to attempt to call one of the inner workflows directly from the command line, we would receive an error. The only workflow visible to us is the `Invoke-TopLevel` workflow. Finally it is possible to define a re-usable function within a workflow. A function in terms of workflow behaves in a similar way to an inline script, but is named and can be called multiple times as follows:

```
Workflow Invoke-MyAge {
$MyAge = 25
Function WriteMyAge {
$MyAge = 26
"MyAge is $MyAge"
 }
"MyAge is $MyAge"
WriteMyAge
}
PS C:\> Invoke-MyAge
MyAge is 25
MyAge is 26
```

Parallel execution

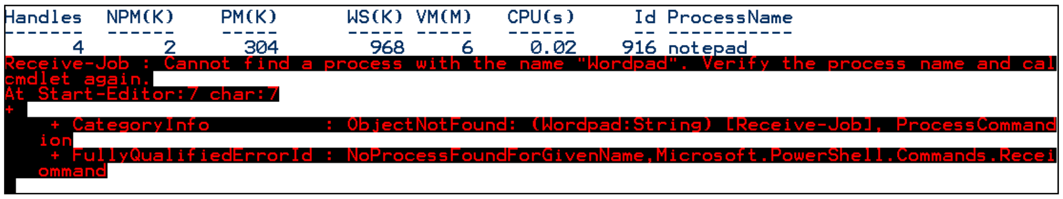
Because each activity could be independent of the next, it is possible to run activities in parallel. To do this we must utilize the `Parallel` keyword followed by a script block as follows:

```
PS C:\> Workflow Start-Editor {
 Parallel {
   Start-Process -FilePath Notepad
   Start-Process -FilePath Wordpad
 }
}
```

Both `Start-Process` commands will start nearly at the same time. Starting activities in parallel can be very beneficial in terms of performance. It fully utilizes the multicore processors that are now standard in most machines. Although this type of functionality was somewhat possible with PowerShell jobs, it was not nearly as easy to accomplish as it is now with the `Parallel` keyword. At any point during parallel processing, it is possible to execute a particular part of the workflow in sequence as well. The `Sequence` keyword forces the processing to take place in order, rather than parallel as follows:

```
PS C:\> Workflow Start-Editor {
      Parallel {
       Start-Process -FilePath Notepad
       Start-Process -FilePath Wordpad
       Sequence {
          Get-Process -Name Notepad
          Get-Process -Name Wordpad
       }
      }
}
PS C:\> Start-Editor
```

The output is as shown in the following screenshot:



```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
      4       2      304        968     6     0.02    916 notepad
Receive-Job : Cannot find a process with the name "Wordpad". Verify the process name and cal
cmdlet again.
At Start-Editor:7 char:7
+
    + CategoryInfo          : ObjectNotFound: (Wordpad:String) [Receive-Job], ProcessCommand
   ion
    + FullyQualifiedErrorId : NoProcessFoundForGivenName,Microsoft.PowerShell.Commands.Recei
   ommand
```

Notice the error in execution of the workflow. This is because the `Sequence` section was started in parallel with the other activities or commands found in the workflow. Due to the variablity of parallel execution, the section will not execute the same way every time. Depending on the speed the other two statements execute at, this error may not be shown every time. To ensure that the `Get-Process` is called only after each process is started, but to ensure that each editor is started in parallel, we could modify the previous script as follows:

```
PS C:\> Workflow Start-Editor {
Parallel {
Sequence {
   Start-Process -FilePath Notepad -Wait
   Get-Process -Name Notepad
 }
 Sequence {
   Start-Process -FilePath Wordpad -Wait
 Get-Process -Name Wordpad
 }
 }
}
```

During the execution of the previous workflow, each sequence is started in parallel, but the activities within the sequence are executed in order.

Running the previous workflow would start each process and return the process information back to the command line. In addition to executing individual activities in parallel, we can iterate over collections of objects in parallel too. This is because the activities within the loop are not related to each other.

Assume the contents of `UserNames.txt` looks something like as follows:

```
Mdnvdi\adriscoll
Prod\oprime
Mdnvdi\sfire
Prod\tstark
mdnvdi\bmarley
…
```

The following workflow iterates over the lines within the text file and outputs them in parallel:

```
PS C:\> workflow Get-User {
 $lines = Get-Content -FilePath C:\UserNames.txt
 foreach -parallel ($line in $lines)
 {
   if ($line -match "mdnvdi")
   {
    "User: $line"
   }
 }
}
PS C:\> Get-User
```

Running the workflow will yield different results depending on the size of the file, but in theory, any line could be output because the loop is executing in parallel. Parallel execution is very powerful but must be used with caution and in the right circumstances. Workflows makes it very easy to create parallel activities.

Persisting data in workflows

Data persistence between activities is natively supported in workflows. This is a corner stone to the workflow idea. To persist data when calling a workflow we can use the `PSPersist` workflow common parameter. This will cause the workflow to persist data to the data store path in between activities. This allows for the workflow to be paused, stopped, and restarted without affecting the state of execution.

In addition to explicitly forcing a workflow to save its state, a workflow can be defined as well when it saves its state. This can be done by utilizing the `Persist` or `Checkpoint-Workflow` activities. This will cause the workflow state to be saved to the default store location.

> The default store location is `C:\Users\<username>\AppData\Local\Microsoft\Windows\PowerShell\WF\PS\default\<user SID>`

Limitations of PowerShell workflows

There are many limitations of PowerShell workflows that are not set on regular scripts. The reason for much of it is to ensure that PowerShell follows the fundamental principles of Windows Workflow. It is required to lose some functionality to gain a whole lot more.

Variable scope

The first rule we need to examine is the workflow variable scope. This is done using the following commands. As already stated, activities are treated as independent units. So the PowerShell state, including variables, is not shared among the activities:

```
PS C:\> Workflow Start-Workflow {
  $process = Start-Process -FilePath Cmd -PassThru
  Get-Process -InputObject $process
}


PS C:\> Start-Workflow


Handles  NPM(K) PM(K) WS(K) VM(M) CPU(s)    Id ProcessName
-------  ------ ----- ----- ----- ------    -- -----------
     47       6  1388  3752    58   0.02  20384 notepad
```

The workflow correctly called `Get-Process` with the `$process` variable. This is because the `$process` variable was defined with workflow level scope. This, much like script level scope, means that the `$process` variable is defined for all lower level blocks of the current workflow. The best way to understand this, is to think of a one-way mirror where the inside can see out but the outside cannot see in. The variables defined on the outside are visible to the inside but not the other way around. As the `$process` variable was defined in the outermost layer, the `Get-Process` has access to it.

Note that workflow level variables are only available to the current workflow and related blocks but not nested workflows.

To examine this point further, we can modify the example using the following commands to use the new `inlineScript` keyword. The `inlineScript` keyword allows us to write a script which is not made of activities but is rather a regular PowerShell script as a single activity:

```
PS C:\> Workflow Start-Workflow {
   inlineScript {
   $process = Start-Process -FilePath Cmd -PassThru
   Get-Process -InputObject $process
  }
  Get-Process -InputObject $process
}

Start-Workflow
Handles  NPM(K) PM(K)  WS(K) VM(M)   CPU(s)  Id ProcessName
-------  ------ -----  ----- -----   ------  -- -----------
     47       6  1388  3752     58     0.02  20384 notepad

Receive-Job : Cannot bind argument to parameter 'InputObject' because it
is null.
```

As we can see, the first `Get-Process` call succeeded. This is because it is part of the `inlineScript`, which is treated as a single activity with its own scope. The second `Get-Process` call fails, because the `$process` variable has not been defined at the workflow scope. Although this works as expected, there are some caveats to variable scope in workflows that require the use of new keywords. The first is using the `scope` keyword. This keyword is used, to ensure that the workflow level variable is used rather than the `inlineScript` level variable as follows:

```
PS C:\> workflow Write-MyAge {
$MyAge = 25
InlineScript {
  $MyAge = 18
  "My age in the PowerShell Script: $MyAge"
  "My age in the Workflow: $using:MyAge"
  }
}

PS C:\> Write-MyAge
My age in the PowerShell Script: 18
My age in the Workflow: 25
```

Notice that the `using` modifier brought in the variable from the outer scope, When the `using` modifier is not specified, the local variable was used instead. It is not possible to set the workflow variable from within an inline script.

No advanced function blocks

Workflows do not support `Begin`, `Process`, or `End` blocks that are commonly found in advanced script functions. Workflows are intended to flow from one activity to another, so defining a start or end block does not make sense in this context. Additionally, workflows do not accept pipeline input so there is no need for a process block.

Activities that run only on the local machine

There are numerous cmdlets that are used to manipulate objects within the pipeline. Because these cmdlets are very heavy in resource usage, it was a decision to make them only run locally. To execute these cmdlets it is necessary to use them within an `inlineScript` block within the workflow.

Some of the cmdlets include the following:

- `Add-Member`
- `Where-Object`
- `Write-Host`
- `Get-Random`
- `ConvertTo-Xml`
- `Measure-Object`

The cmdlets that have no activity implementation

Because the general concept of workflow is slightly different than that of PowerShell of a regular PowerShell script, the numerous cmdlets that do not make sense are run as workflow activities. Microsoft has outlined each of these cmdlets and provided a reasonable explanation, as to why they decided to exclude the cmdlet from an activity implementation. Although it is not possible to use the cmdlet as an activity, it is still possible to call the cmdlet in an `inlineScript` block within a workflow.

Some of the cmdlets that have been excluded:

- `New-Alias`: This is used to only modify the current session. Activities do not share state, so this would not make sense to ever call.
- `Show-Command`: It is not possible to interact directly with a workflow.
- `Set-PSBreakpoint`: It is not possible to debug a workflow.
- `Format-List`: It is not possible to format within a workflow.
- `Enter-PSSession`: Remote capabilities are native to a workflow, so there is no need to utilize remoting.

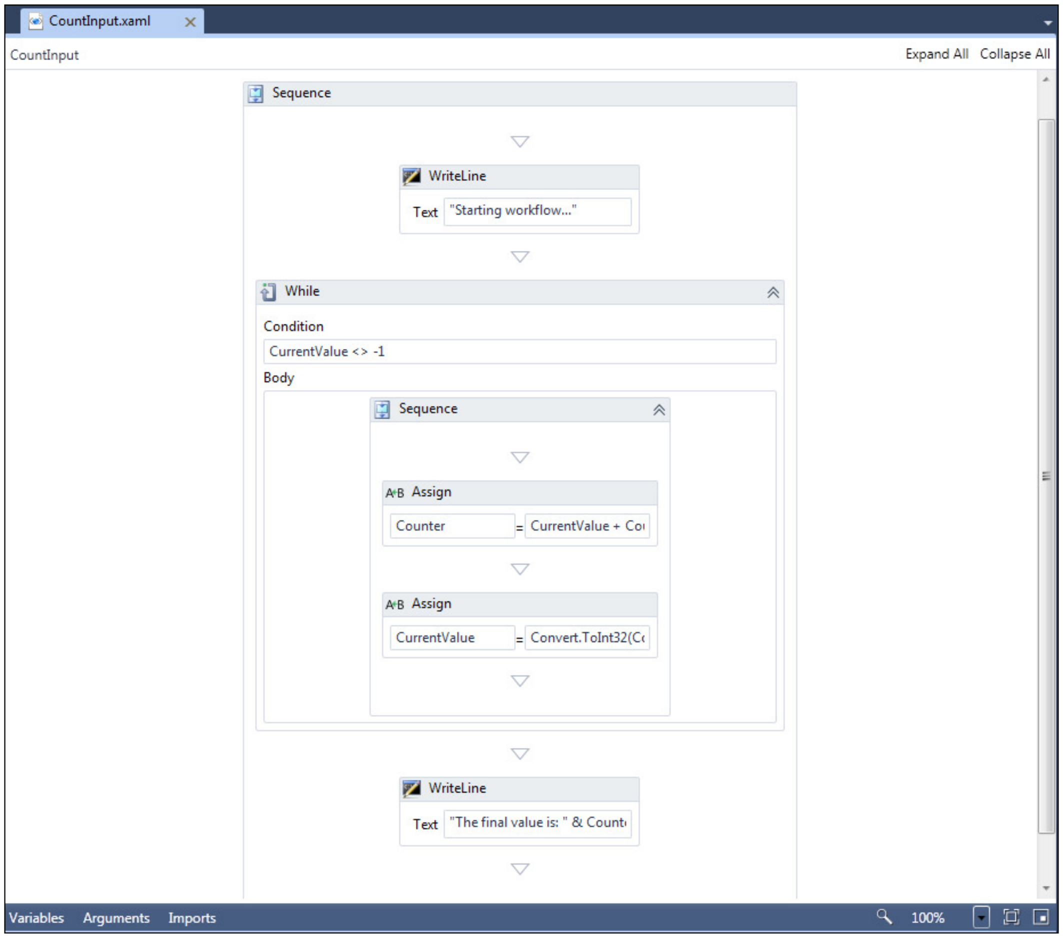
Importing activities into PowerShell

There are many activities available in Visual Studio. In addition to the default workflow activities, there are other product based workflows, such as Team Foundation Server. For example, **Team Foundation Server** (**TFS**) uses workflow to guide the automated build process. Whenever code is committed to the source control system, TFS can trigger the workflow to build, test, and even deploy the source. Using Windows Workflow to guide this makes it very easy for many build engineers to create complex workflows that are generally easy to follow.

With the proliferation of workflow into other Microsoft products, it becomes very necessary to harness this type of functionality directly in PowerShell. Although the basic, built-in PowerShell activities or cmdlets offer a lot of functionality, the ability to trigger activities just as TFS would, makes it even better.

In this section, we will investigate how to import and work with activities that are not based on PowerShell. We will look at the limitations of certain workflow activities as well. In the following example, we will create a workflow in Visual Studio and import it into PowerShell. We will then execute and interact with the workflow, just as we would with a regular PowerShell workflow.

Our example workflow, `CountInput`, is responsible for gathering integers from the user on the command line. It will count those integers until it encounters a `-1`. Once a `-1` is encountered it will display the final count of all the integers entered. This looks as shown in the following screenshot, in the Visual Studio workflow designer:



Text "Starting workflow..."

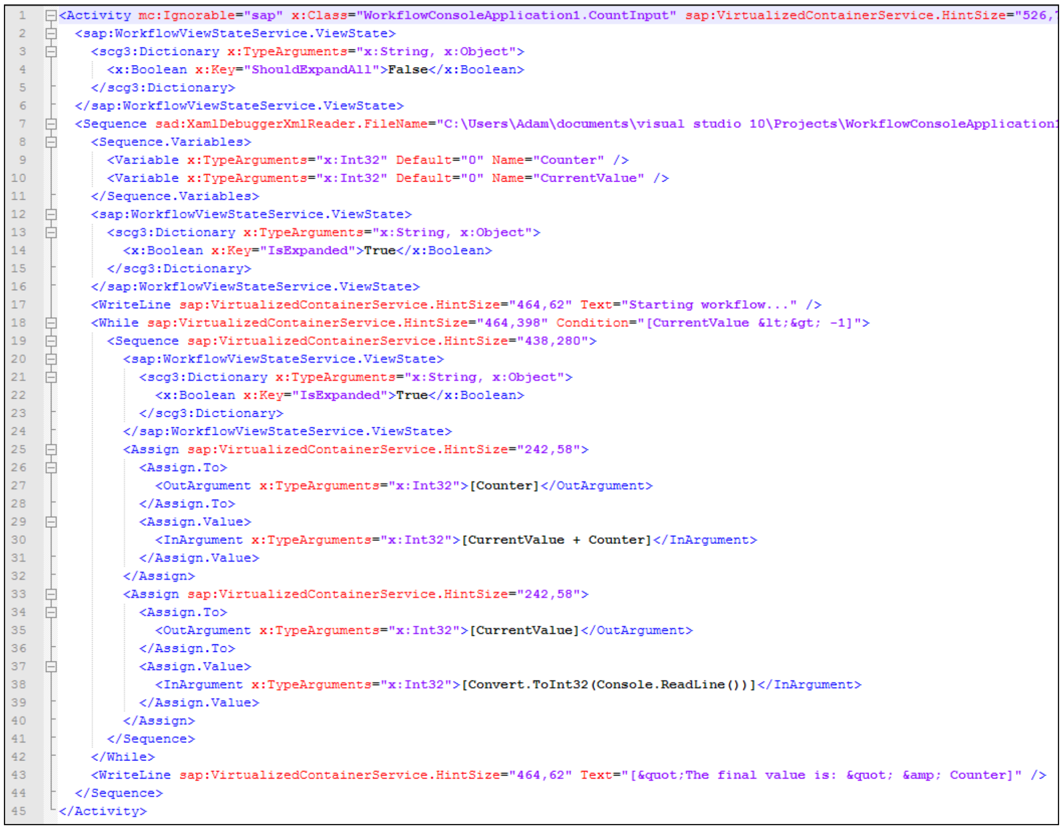
Condition
CurrentValue <> -1
Body

Counter = CurrentValue + Coı

CurrentValue = Convert.ToInt32(Cı

Text "The final value is: " & Count

This workflow utilizes several built-in workflow activities including `Sequence`, `WriteLine`, `While`, and `Assign`. The `Counter` and `CurrentValue` variables are defined on the `Sequence` level. This means, they are in scope for the entire outer sequence: virtually the entire workflow. The workflow is actually composed of an XAML, as mentioned earilier. The `CountInput.xaml` file is an XML-like definition of a workflow. This is the native type of definition for a workflow. The XAML defines all activity assemblies that are required to be loaded, any variables defined in the workflow, and the actual steps taken within the workflow, as shown in the following screenshot:



```
<Activity mc:Ignorable="sap" x:Class="WorkflowConsoleApplication1.CountInput" sap:VirtualizedContainerService.HintSize="526,
  <sap:WorkflowViewStateService.ViewState>
    <scg3:Dictionary x:TypeArguments="x:String, x:Object">
      <x:Boolean x:Key="ShouldExpandAll">False</x:Boolean>
    </scg3:Dictionary>
  </sap:WorkflowViewStateService.ViewState>
  <Sequence sad:XamlDebuggerXmlReader.FileName="C:\Users\Adam\documents\visual studio 10\Projects\WorkflowConsoleApplication
    <Sequence.Variables>
      <Variable x:TypeArguments="x:Int32" Default="0" Name="Counter" />
      <Variable x:TypeArguments="x:Int32" Default="0" Name="CurrentValue" />
    </Sequence.Variables>
    <sap:WorkflowViewStateService.ViewState>
      <scg3:Dictionary x:TypeArguments="x:String, x:Object">
        <x:Boolean x:Key="IsExpanded">True</x:Boolean>
      </scg3:Dictionary>
    </sap:WorkflowViewStateService.ViewState>
    <WriteLine sap:VirtualizedContainerService.HintSize="464,62" Text="Starting workflow..." />
    <While sap:VirtualizedContainerService.HintSize="464,398" Condition="[CurrentValue &lt;&gt; -1]">
      <Sequence sap:VirtualizedContainerService.HintSize="438,280">
        <sap:WorkflowViewStateService.ViewState>
          <scg3:Dictionary x:TypeArguments="x:String, x:Object">
            <x:Boolean x:Key="IsExpanded">True</x:Boolean>
          </scg3:Dictionary>
        </sap:WorkflowViewStateService.ViewState>
        <Assign sap:VirtualizedContainerService.HintSize="242,58">
          <Assign.To>
            <OutArgument x:TypeArguments="x:Int32">[Counter]</OutArgument>
          </Assign.To>
          <Assign.Value>
            <InArgument x:TypeArguments="x:Int32">[CurrentValue + Counter]</InArgument>
          </Assign.Value>
        </Assign>
        <Assign sap:VirtualizedContainerService.HintSize="242,58">
          <Assign.To>
            <OutArgument x:TypeArguments="x:Int32">[CurrentValue]</OutArgument>
          </Assign.To>
          <Assign.Value>
            <InArgument x:TypeArguments="x:Int32">[Convert.ToInt32(Console.ReadLine())]</InArgument>
          </Assign.Value>
        </Assign>
      </Sequence>
    </While>
    <WriteLine sap:VirtualizedContainerService.HintSize="464,62" Text="[&quot;The final value is: &quot; &amp; Counter]" />
  </Sequence>
</Activity>
```

This XAML is what PowerShell inteprets and is capable of running directly. To see this in action we need to import the XAML using the `Import-Module` cmdlet with the path to the XAML file, as shown in the following commands:

```
PS C:\> Import-Module C:\CountInput.xaml -Verbose
VERBOSE: Loading module from path 'C:\CountInput.xaml'.
VERBOSE: Importing cmdlet 'Import-PSWorkflow'.
```

```
VERBOSE: Importing cmdlet 'New-PSWorkflowExecutionOption'.
VERBOSE: Loading workflow C:\CountInput.xaml
VERBOSE: Exporting workflow 'CountInput'.
VERBOSE: Importing command as workflow 'CountInput'.
```

You'll notice the `Import-Module` does a couple things while importing the new XAML workflow. Among them you will see that it imported the `CountInput` command. Now that the count input has been imported, we can execute it just like we would execute any other workflow in PowerShell as follows:

```
PS C:\> CountInput
Starting workflow...
100
26
42
200
-100
-1
The final value is: 268
```

The workflow successfully ran and executed as we were expecting. It gathered each one of the inputs and added them to the counter variable. Once we entered a `-1` the while loop finished and the final value was displayed in a formatted string. Albeit this is a simple example, but it shows how seamless integrating a custom XAML based workflow into PowerShell is. There is no need to have to worry about assemblies or prerequisites as the XAML is responsible for this. Instead, we only need to import the XAML file and execute its exposed activities.

Another interesting piece of information is that as the `Import-Module` cmdlet is used to import XAML files into PowerShell, XAML files can now be packaged into modules just like scripts and binary assemblies. They can be specified in the module manifest as any one of the values for `RootModule`, `NestedModules`, `RequiredModules` or `RequiredAssemblies`.

Finally, it is important to know that not all built-in activities are supported by PowerShell. There is a small list of the activities that will not work correctly as follows:

- `Pick`
- `PickBranch`
- `Send`
- `SendReply`

- `Receive`
- `ReceiveReply`
- `CancellationScope`
- `CompensableActivity`
- `Compensate`
- `Confirm`
- `TransactionScope`
- `TransactedReceiveScope`
- `Interop`

In terms of how many activities there are available, this list is very small. Now that we have examined using a Visual Studio or XAML workflow in PowerShell, we will look at how to work in the opposite direction. This is to use PowerShell activities within Visual Studio.

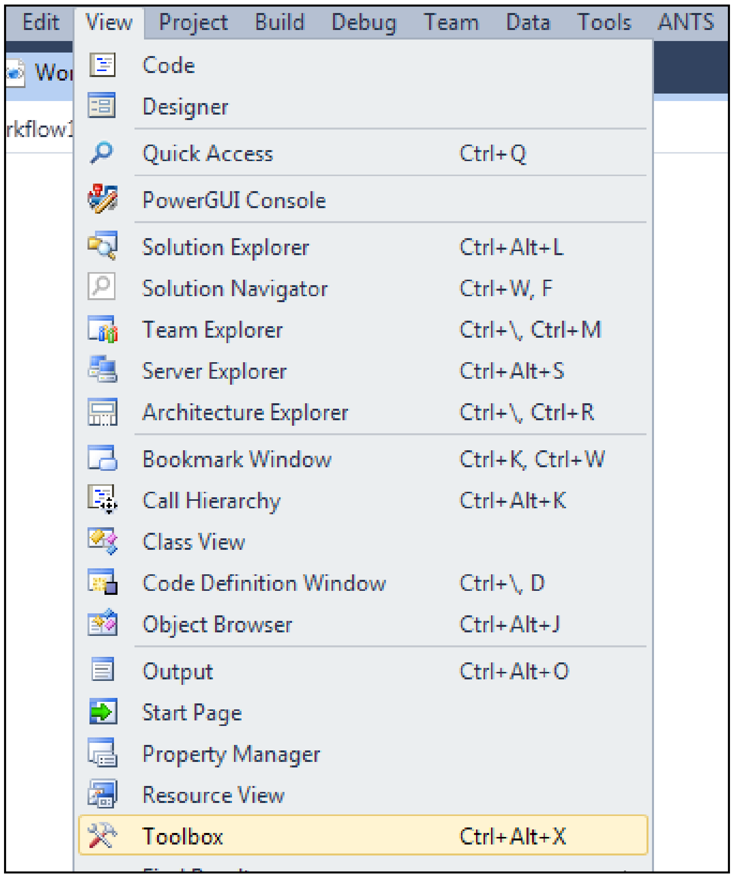
Using PowerShell workflows in Visual Studio

There are two different ways of utilizing PowerShell activities within Visual Studio. We can use the predefined activities by adding the Microsoft Activity assemblies for PowerShell or we can save our custom PowerShell Workflows to an XAML file which can be added to a Visual Studio Workflow project. When adding a custom XAML workflow, all the assemblies will be added automatically.

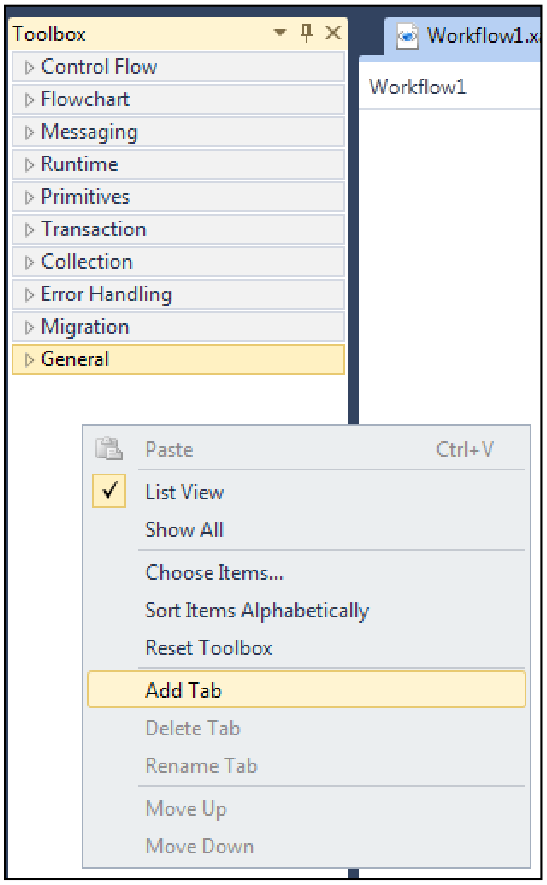
Adding a Microsoft workflow activity assembly

By adding an import to the correct PowerShell assemblies, we can drag the activities directly into the workflow like we would any other workflow activity.

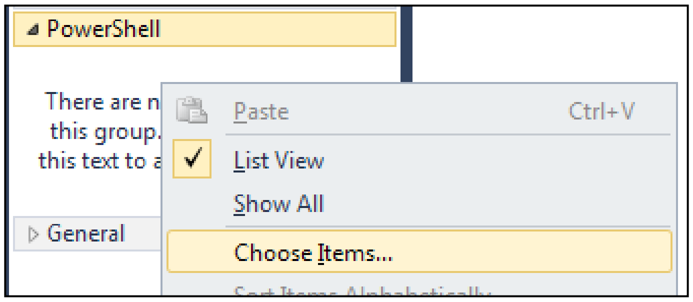
The first step to getting access to our PowerShell workflow activity is to add the assembly import to the workflow designer. When using the workflow designer, the **Toolbox** window is where we can access all of our activities. If it isn't open, it can be opened by clicking on the **View** menu and the clicking on **Toolbox**. This is shown in the following screenshot:


Edit View Project Build Debug Team Data Tools ANTS
Code
Designer
Quick Access Ctrl+Q
PowerGUI Console
Solution Explorer Ctrl+Alt+L
Solution Navigator Ctrl+W, F
Team Explorer Ctrl+\, Ctrl+M
Server Explorer Ctrl+Alt+S
Architecture Explorer Ctrl+\, Ctrl+R
Bookmark Window Ctrl+K, Ctrl+W
Call Hierarchy Ctrl+Alt+K
Class View
Code Definition Window Ctrl+\, D
Object Browser Ctrl+Alt+J
Output Ctrl+Alt+O
Start Page
Property Manager
Resource View
Toolbox Ctrl+Alt+X

To better organize the **Toolbox** window we want to create a new tab to store the PowerShell activities. Right-click within the **Toolbox** and then select **Add Tab**. This is explained in the following screenshot:



Name the tab PowerShell and then right-click within the tab and select **Choose Items**. We will see the following screenshot:

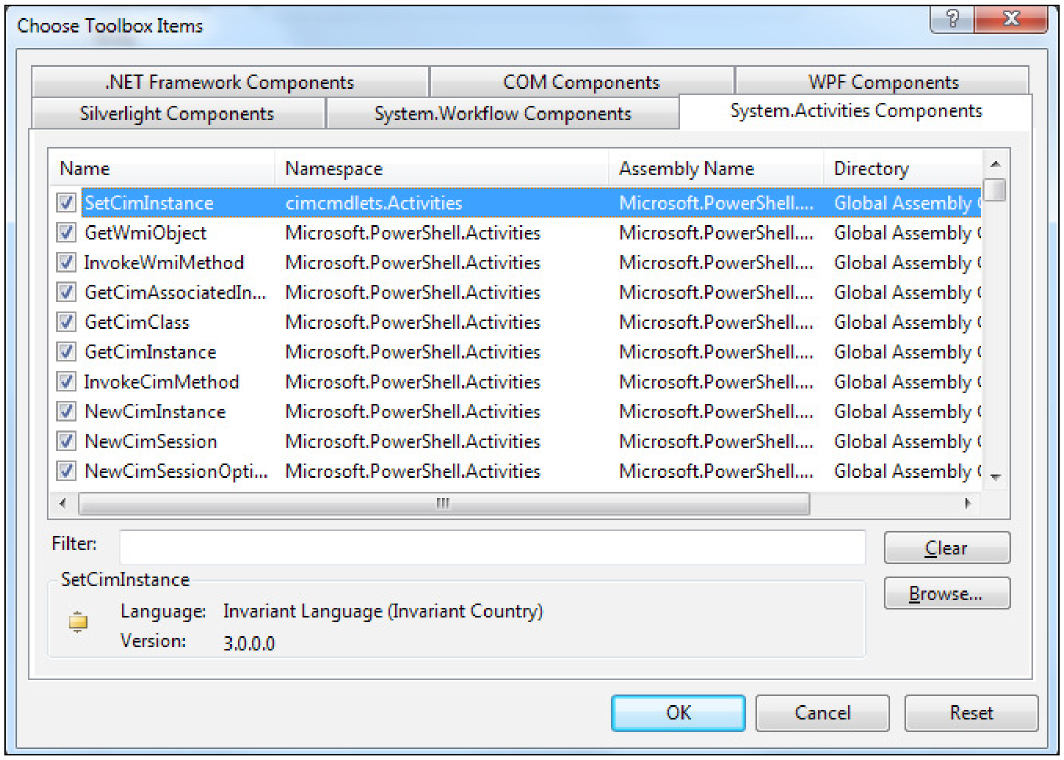


This will bring up the activity selection dialog. To get access to the `PowerShell` activities, we need to navigate to the **Global Assembly Cache** (**GAC**). Within the GAC we can select any one of the following `PowerShell` activity assemblies:

- `Microsoft.PowerShell.Activities`
- `Microsoft.PowerShell.Core.Activities`
- `Microsoft.PowerShell.Diagnostics.Activities`
- `Microsoft.PowerShell.Management.Activities`
- `Microsoft.PowerShell.Security.Activities`
- `Microsoft.PowerShell.Utility.Activities`
- `Microsoft.WSMan.Management.Activities`

The Global Assembly Cache is located in `C:\Windows\Assembly\GAC_MSIL`.

Once the assembly has been loaded, we can choose from the activities found within the assembly. Selecting an activity will put it into the **PowerShell** tab within the **Toolbox** window:



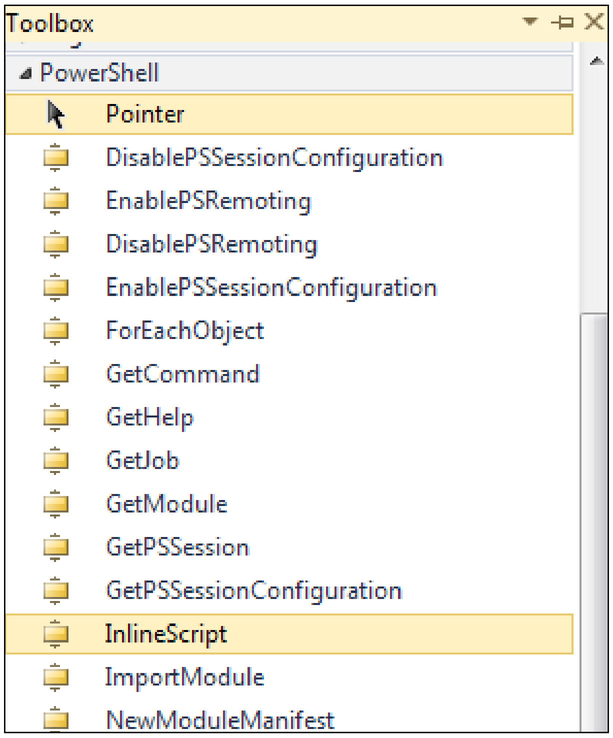
Name	Namespace	Assembly Name	Directory
✓ SetCimInstance	cimcmdlets.Activities	Microsoft.PowerShell....	Global Assembly
✓ GetWmiObject	Microsoft.PowerShell.Activities	Microsoft.PowerShell....	Global Assembly
✓ InvokeWmiMethod	Microsoft.PowerShell.Activities	Microsoft.PowerShell....	Global Assembly
✓ GetCimAssociatedIn...	Microsoft.PowerShell.Activities	Microsoft.PowerShell....	Global Assembly
✓ GetCimClass	Microsoft.PowerShell.Activities	Microsoft.PowerShell....	Global Assembly
✓ GetCimInstance	Microsoft.PowerShell.Activities	Microsoft.PowerShell....	Global Assembly
✓ InvokeCimMethod	Microsoft.PowerShell.Activities	Microsoft.PowerShell....	Global Assembly
✓ NewCimInstance	Microsoft.PowerShell.Activities	Microsoft.PowerShell....	Global Assembly
✓ NewCimSession	Microsoft.PowerShell.Activities	Microsoft.PowerShell....	Global Assembly
✓ NewCimSessionOpti...	Microsoft.PowerShell.Activities	Microsoft.PowerShell....	Global Assembly

SetCimInstance

Language: Invariant Language (Invariant Country)

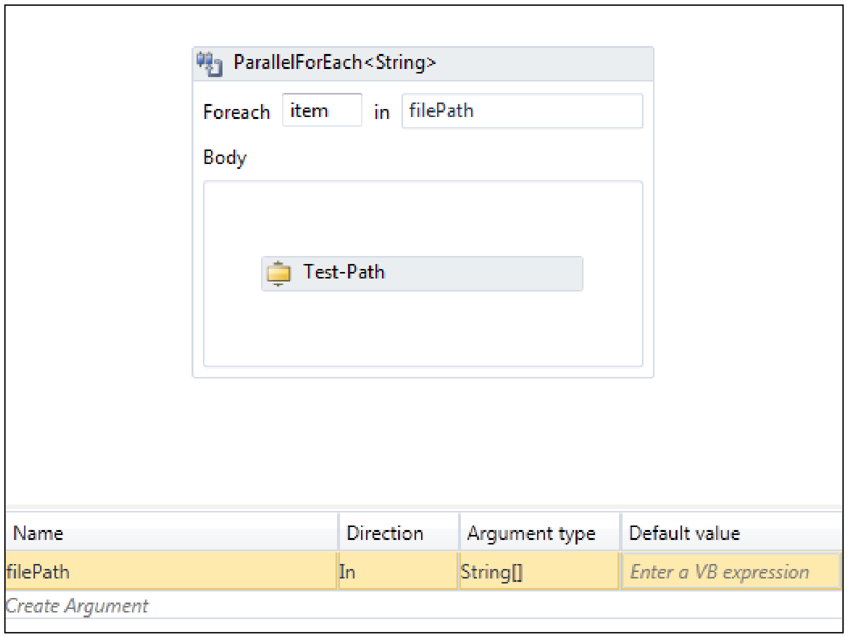
Version: 3.0.0.0

We have selected all activities in the `Microsoft.PowerShell.Core.Activities`, `Microsoft.PowerShell.Activities`, and `Microsoft.PowerShell.Management.Activities` assemblies. Some of these activities include **GetCommand**, **GetModule**, **StartJob**, and even **InlineScript**. Now that we have the activities added to the **Toolbox** window, we can start to utilize these activities within the workflow. Refer to the following screenshot:



In our previous Visual Studio Workflow, we were accepting integers and accumlating a final value. PowerShell workflows are intended not being interacted with, while they are running. Although we could run our previous workflow on the command line, we could interact with it, it was because we had no PowerShell-based activities within the workflow. If we were to add one to the existing workflow an error would be presented.

In this next example, let's take file names as input to the workflow, and then use that input to output whether the files exist using `Test-Path` in parallel. The `Test-Path` activity is found in the `Microsoft.PowerShell.Management.Activities.dll`. It is shown in the following screenshot:



Name	Direction	Argument type	Default value
filePath	In	String[]	Enter a VB expression
Create Argument			

In this workflow you can see that we added a new argument, called `filePath`, which accepts an array of strings. These strings are then sent into the `ParallelForEach` activity which then passes them to the PowerShell `Test-Path` activity. Once we save and import the activity into PowerShell, we can execute it and specify the `filePath` parameter using the following commands:

```
PS C:\> TestPathParallel -filePath C:\inetpub,C:\Users
True
True
```

Utilizing the built-in PowerShell activities is very helpful as many of the commands offer a better experience than using simple workflow commands. To take PowerShell integration into Visual Studio a step further, let's investigate how to expose our custom PowerShell workflow into Visual Studio.

Adding a custom PowerShell workflow to Visual Studio

To add a custom PowerShell workflow to Visual Studio, we need to save the XAML generated by PowerShell to an XAML file. Although there is no built-in function or cmdlet to do this, it is very easy. Running the following command will output the XAML that has been generated for a workflow:

```
PS C:\Users\Adam> (Get-Command MyWorkflow).XamlDefinition["MyWorkflow"]
<Activity
  x:Class="Microsoft.PowerShell.DynamicActivities.Activity_1844550451"
...
```

As the XAML definition is available right on the command line, we can easily pipe this into the `Out-File` cmdlet to write out the XAML definition to a file, which can be consumed by Visual Studio using the following command:

```
PS C:\> (Get-Command MyWorkflow).XamlDefinition["MyWorkflow"] |
OutFile –FilePath C:\MyWorkflow.xaml
```

Now that we have the XAML file saved, we can easily add it to the Visual Studio Solution Explorer and include it in the project.

Summary

In this chapter we examined the new Windows Workflow integration into PowerShell. The ability to use Workflow seamlessly within PowerShell offers a host of new functionality and possible solutions that make an administrator's job easier. The Workflow engine allows for a simplistic approach to problems which did not sacrifice functionality. Being able to harness custom workflows within Visual Studio enables users that are not knowledgeable in PowerShell to access and work with many of the key cmdlets, or even import custom PowerShell workflows written by others.

By integrating with Workflow, PowerShell has taken a large step forward in terms of scalability and re-usability. Enabling a PowerShell user to manage infrastructures well beyond the ones they currently are capable of. Care should be taken in understanding the differences between a PowerShell Workflow and a PowerShell script as they are very different from one another.

In the next chapter, we will learn how PowerShell controls WMI interfaces.

5
Using the Common Informational Model

Windows Management Instrumentation (**WMI**) is an infrastructure built into Windows which allows for management and automation of the system. WMI is based on the **Common Informational Model** (**CIM)**, which helps define objects within a computer system using a common model. Thus, WMI is Microsoft's implementation of CIM.

PowerShell exposes ways of querying and controlling the WMI interfaces to make working with the system easier. Since the first version of PowerShell, cmdlets and type extensions have existed that have made working with WMI seamless. In the newest version of PowerShell and the Windows Management Framework, WMI and PowerShell become even more intertwined. The Windows Management Framework is an overarching category of systems that both PowerShell and WMI fall under. We do not simply install PowerShell; we install the Windows Management Framework. WMI is packaged into this installation as well.

In PowerShell 3.0 it is possible to harness WMI and to utilize the new WMI constructs, to produce PowerShell cmdlets using native C code. Microsoft has released a new integrated development environment which extends Visual Studio to allow for simple creation of WMI providers. The simplified model enables developers to quickly create providers that can provide both data and invoke actions.

In this chapter we will cover the following:

- The CIM IDE that is used to easily create new WMI providers
- Create a simple provider and then use PowerShell to create cmdlets based on metadata about the provider
- The differences among WMI cmdlets that have been in previous versions and experiment with the new cmdlets in this version
- The new cmdlets for interacting with CIM classes and instances

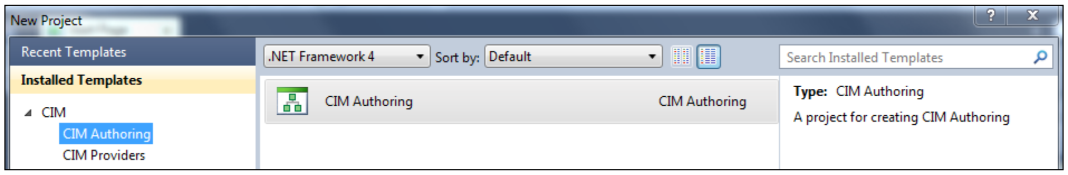
Finally, we will take a quick look at the NanoWBEM service which will be available around the time when Windows 8 is shipped. NanoWBEM is a Linux based CIM service, which allows for management of machines or devices that are not running Windows.

Introduction to the CIM IDE

Visual Studio is a very extensible development environment. The core system is built upon an idea of services that are discoverable and can easily be plugged in and out. As more services are added to the framework more functionality is exposed. In Visual Studio 2010 Microsoft made it even easier to extend the IDE. The Visual Studio 2010 software development kit (SDK), simplified the way in which developers work with Visual Studio. There are several thousand extensions found on the Visual Studio Gallery that provide all kinds of different functionality.

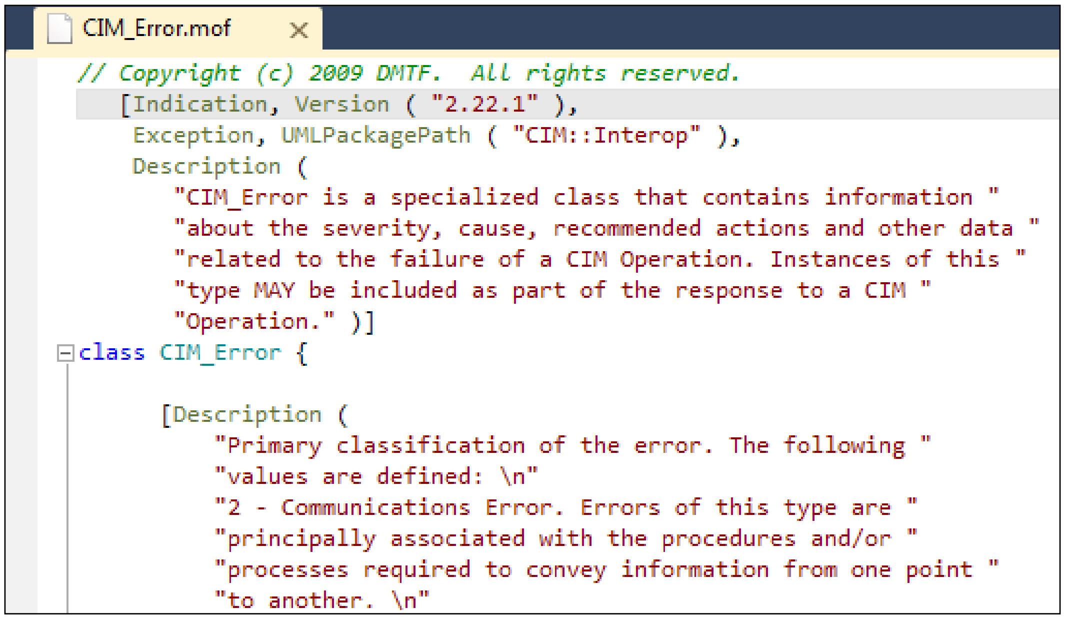
To download a trial version of Visual Studio you can visit `http://www.microsoft.com/visualstudio`

The CIM IDE is based on this extension model and plugs right into Visual Studio. Currently, the CIM IDE is available for download from the Visual Studio Code Gallery. Once the extension is installed we will have access to two new project types within Visual Studio. Once the new project dialog is open, we see that there is a new category of project; **CIM**. The first project type, **CIM Authoring**, is responsible for defining the WMI model interface which will be exposed to PowerShell. The second project type **CIM Providers**, is responsible for implementing the interface generated by the first. The following screenshot explains this:



Type: CIM Authoring
A project for creating CIM Authoring

In addition to the different project types available in the CIM IDE, there are new designers, syntax highlighting, and IntelliSense support. In a **CIM Authoring** project, the WMI interface is defined using a standard format refered to as the **Managed Object Format** (**MOF**). MOF files are simple attributed C-style source files, that provide the metadata about the properties, methods, and events exposed by a WMI provider's objects. The CIM IDE provides syntax highliting and IntelliSense support for MOF files. The following screenshot shows an `CIM_Error` MOF file:

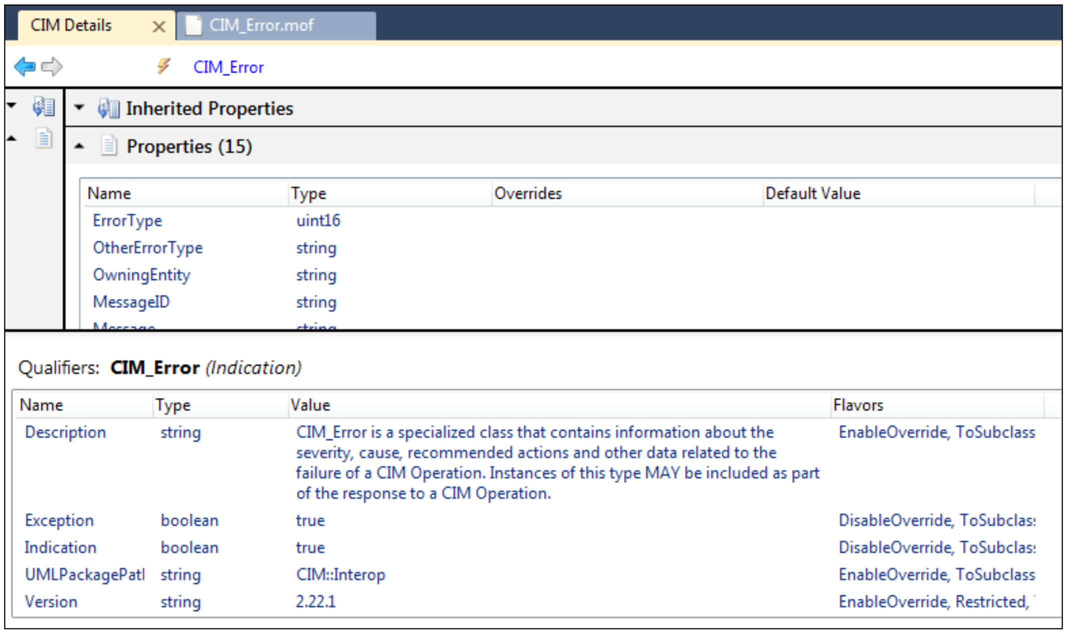


CIM_Error.mof

```
// Copyright (c) 2009 DMTF.  All rights reserved.
   [Indication, Version ( "2.22.1" ),
    Exception, UMLPackagePath ( "CIM::Interop" ),
    Description (
       "CIM_Error is a specialized class that contains information "
       "about the severity, cause, recommended actions and other data "
       "related to the failure of a CIM Operation. Instances of this "
       "type MAY be included as part of the response to a CIM "
       "Operation." )]
class CIM_Error {

      [Description (
          "Primary classification of the error. The following "
          "values are defined: \n"
          "2 - Communications Error. Errors of this type are "
          "principally associated with the procedures and/or "
          "processes required to convey information from one point "
          "to another. \n"
```

The MOF format is a specification designed by the **Distributed Management Task Force** (**DMTF**). The DMTF works to create standards for the distributed management of computer systems. By having standard ways of communicating within a distributed organization it is easier to ensure interoperability between systems.

The CIM IDE also offers a simplified, read-only view of the MOF file in a custom designer. It lists each of the methods, properties, and events defined within a MOF file. This makes reading MOF files much easier, as there is often much metadata found within the file that must be interpreted:



Name	Type	Overrides	Default Value
ErrorType	uint16		
OtherErrorType	string		
OwningEntity	string		
MessageID	string		

Qualifiers: **CIM_Error** *(Indication)*

Name	Type	Value	Flavors
Description	string	CIM_Error is a specialized class that contains information about the severity, cause, recommended actions and other data related to the failure of a CIM Operation. Instances of this type MAY be included as part of the response to a CIM Operation.	EnableOverride, ToSubclass
Exception	boolean	true	DisableOverride, ToSubclas
Indication	boolean	true	DisableOverride, ToSubclas
UMLPackagePatl	string	CIM::Interop	EnableOverride, ToSubclass
Version	string	2.22.1	EnableOverride, Restricted,

By viewing the MOF file in the **CIM Explorer** view, it is much easier to understand the purpose of the interface defined in the MOF file. Once we have defined a custom object type it is easy to have the CIM IDE generate the framework code for us. After the framework has been laid out, it is our responsibility to implement the different properties and methods within the **CIM Provider** project.

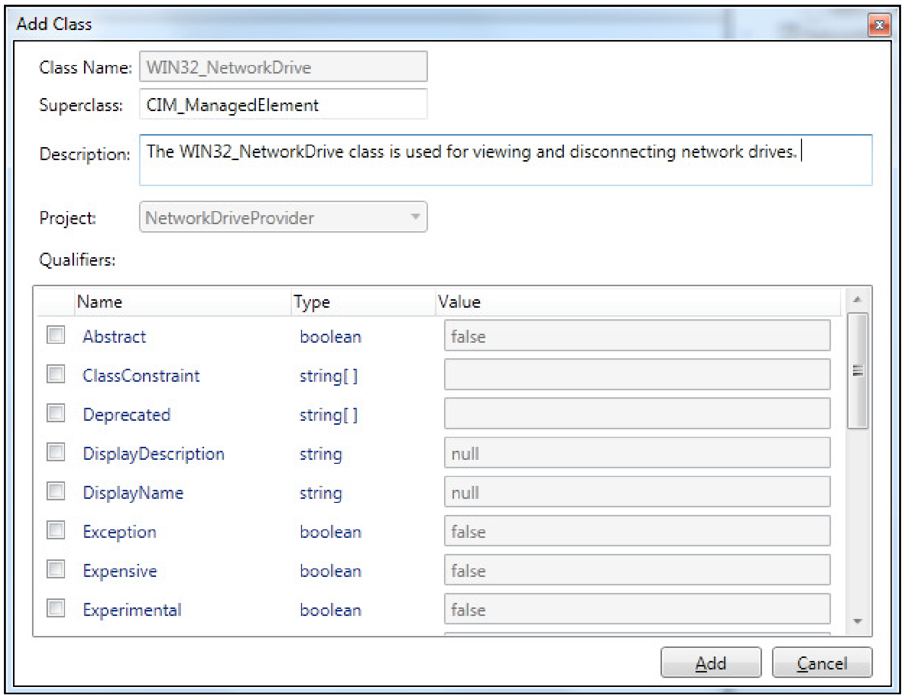
Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Implementing a simple WMI provider

In the following example, we will generate the MOF and build the skeleton for a provider from scratch. We won't actually implement the provider's internals but we will see how easy it is to get started creating a provider. Once this is done we will use the new **cmdletization** feature found in the WMF to create native PowerShell cmdlets without having to write any code or scripts to do so. The goal of this provider is to list the connected network drives and provide a way of disconnecting them. This section will not be about the code necessary in implementing the actual provider as this is beyond the scope of this book.

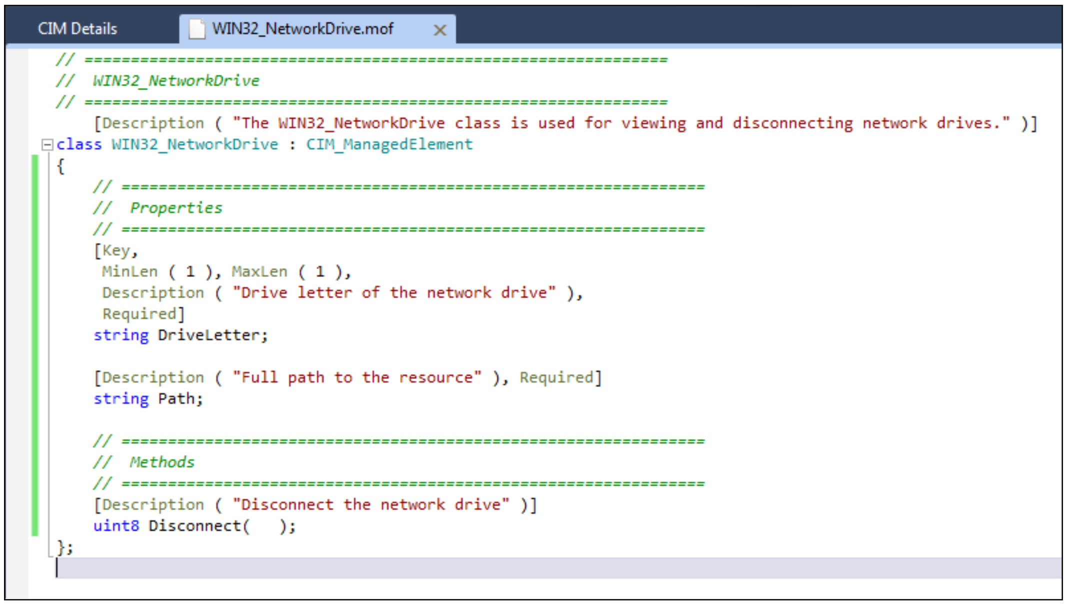
The first step in creating our new WMI provider is to author the MOF, as seen in the previous section. We need to create a new **CIM Authoring** project. This can be done by clicking on the **Menu** and selecting **File** and the **New Project**. Once this is done, we can add a new MOF file to the project to define our class. The class we are looking to implement will be called **WIN32_NetworkDrive**. It will expose a `DriveLetter` and `Path` properties and a method called `Disconnect`. To create our first MOF file, right-click on the **Project** and click on **Add** and then **CIM Class**. Name the file `WIN32_NetworkDrive.mof`. Another dialog will be shown which allows us to set several features of the class we are creating. This includes the base class and numerous attributes of the class, including `Experimental`, `Deprecated`, or `DisplayName`. This is shown in the following screenshot:



Class Name: WIN32_NetworkDrive
Superclass: CIM_ManagedElement
Description: The WIN32_NetworkDrive class is used for viewing and disconnecting network drives.
Project: NetworkDriveProvider
Qualifiers:

	Name	Type	Value
☐	Abstract	boolean	false
☐	ClassConstraint	string[ ]	
☐	Deprecated	string[ ]	
☐	DisplayDescription	string	null
☐	DisplayName	string	null
☐	Exception	boolean	false
☐	Expensive	boolean	false
☐	Experimental	boolean	false

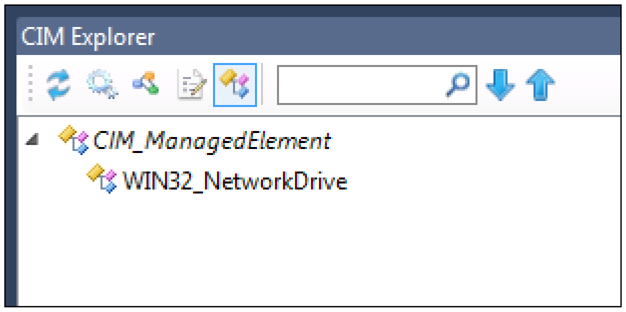
We want our class's superclass to be `CIM_ManagedElement`. It defines some of the basic properties that are found on a CIM class. If we use a standard base class found in the WMI repository, the CIM IDE will automatically generate the MOF file for that base class. Once our class is created we can begin editing the MOF. The syntax is very simple. Properties are defined by providing the type name and then naming them. This is exactly the same syntax would provide for a typical field found in a class. Methods are defined by a return type and parameters within parenthesis. The `Disconnect` method will not take any parameters:



```
// ==================================================================
//  WIN32_NetworkDrive
// ==================================================================
    [Description ( "The WIN32_NetworkDrive class is used for viewing and disconnecting network drives." )]
class WIN32_NetworkDrive : CIM_ManagedElement
{
    // ==================================================================
    //  Properties
    // ==================================================================
    [Key,
     MinLen ( 1 ), MaxLen ( 1 ),
     Description ( "Drive letter of the network drive" ),
     Required]
    string DriveLetter;

    [Description ( "Full path to the resource" ), Required]
    string Path;

    // ==================================================================
    //  Methods
    // ==================================================================
    [Description ( "Disconnect the network drive" )]
    uint8 Disconnect(   );
};
```

Adding a CIM Provider project

Now that we have the definition for our provider authored, we can generate the code that will be a framework for the provider implementation. To do this we first need to create a new project within our solution. Right-click on the **Solution** in the **Solution Explorer** window and click on **Add** and then **New Project** and select the **CIM Provider** project type. Once we have the project added we can generate the source code by using the **CIM Explorer** window. If this window is not currently shown, we can display it by clicking on the **View** menu item, navigating to **Other Windows** and clicking on **CIM Explorer**. The **CIM Explorer** window displays all the classes that have been defined in our current **CIM Authoring** project. In our case, we see that the **WIN32_NetworkDrive** class is shown in the following screenshot:

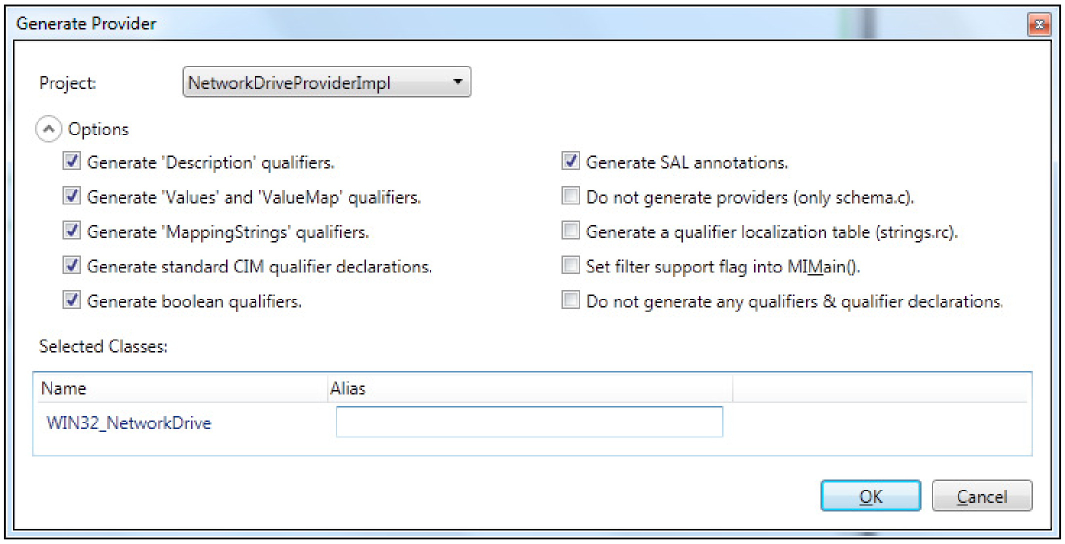


The implementation details

We can now generate the code for the implementation of the provider. To do this, we need to do the following:

1. Select the **WIN32_NetworkDrive** class**.**
2. Click on the **Generate Code** button on the toolbar of the **CIM Explorer** window. It appears as two gears.
3. Select the options we wish to use in the **Generate Provider** window which is now shown.

The **Generate Provider** window also lets us select the **Project** within which we will be writing the implementation of the provider. In our circumstance, as there is only one project in which this can be done, the drop-down list already has the project selected. There is no need to modify any of the default settings for this example. This is shown in the following screenshot:



Generate Provider

Project: NetworkDriveProviderImpl

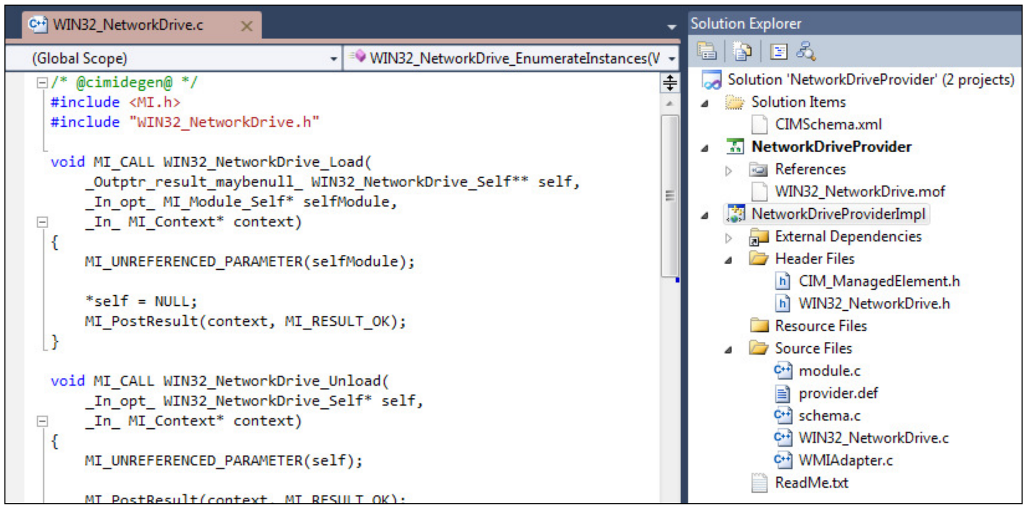
Options

- [✓] Generate 'Description' qualifiers.
- [✓] Generate 'Values' and 'ValueMap' qualifiers.
- [✓] Generate 'MappingStrings' qualifiers.
- [✓] Generate standard CIM qualifier declarations.
- [✓] Generate boolean qualifiers.
- [✓] Generate SAL annotations.
- [ ] Do not generate providers (only schema.c).
- [ ] Generate a qualifier localization table (strings.rc).
- [ ] Set filter support flag into MIMain().
- [ ] Do not generate any qualifiers & qualifier declarations.

Selected Classes:

Name	Alias
WIN32_NetworkDrive	

Once we are ready to continue we can press **OK**. The CIM IDE instantly produces several header and source files in the **NetworkDriveProviderImpl** project. Opening the content of one of these files we can see that the entire skeleton for our provider has been created:



```
/* @cimidegen@ */
#include <MI.h>
#include "WIN32_NetworkDrive.h"

void MI_CALL WIN32_NetworkDrive_Load(
    _Outptr_result_maybenull_ WIN32_NetworkDrive_Self** self,
    _In_opt_ MI_Module_Self* selfModule,
    _In_ MI_Context* context)
{
    MI_UNREFERENCED_PARAMETER(selfModule);

    *self = NULL;
    MI_PostResult(context, MI_RESULT_OK);
}

void MI_CALL WIN32_NetworkDrive_Unload(
    _In_opt_ WIN32_NetworkDrive_Self* self,
    _In_ MI_Context* context)
{
    MI_UNREFERENCED_PARAMETER(self);

    MI_PostResult(context, MI_RESULT_OK);
```

Solution Explorer

```
Solution 'NetworkDriveProvider' (2 projects)
  Solution Items
    CIMSchema.xml
  NetworkDriveProvider
    References
    WIN32_NetworkDrive.mof
  NetworkDriveProviderImpl
    External Dependencies
    Header Files
      CIM_ManagedElement.h
      WIN32_NetworkDrive.h
    Resource Files
    Source Files
      module.c
      provider.def
      schema.c
      WIN32_NetworkDrive.c
      WMIAdapter.c
    ReadMe.txt
```

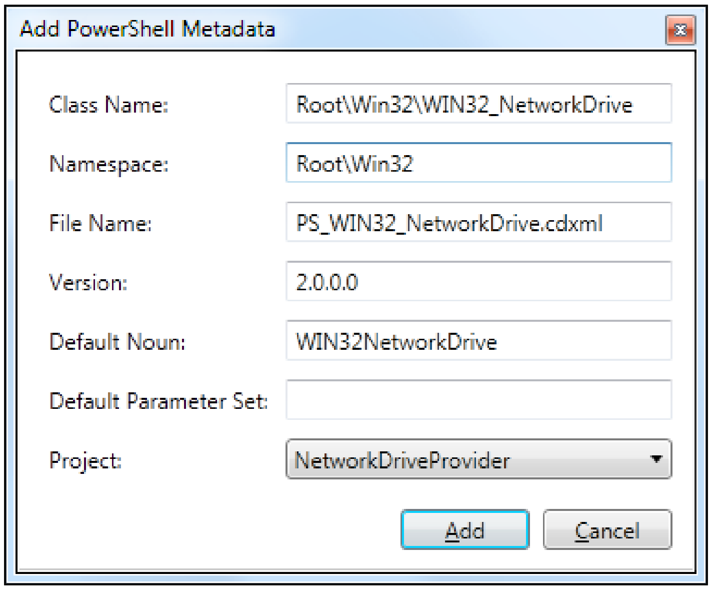
The implementation details of a provider are beyond the scope of this book. A developer could write the C or C++ code necessary to complete the provider and compile it. Once the provider was complete, we could then move into PowerShell to register the provider and automatically create cmdlets based on the methods exposed by the object model.

Generating PowerShell metadata

The last step we need to take is to generate metadata that PowerShell can consume to automatically create cmdlets based on the provider's object model. This can be done through the following steps:

1. Right-click on the **WIN32_NetworkDrive** class in the **CIM Explorer** window.
2. Click on the **Add PowerShell Metadata** option in menu.
3. The **Add PowerShell Metadata** window will be shown where we can set the settings for the metadata file.

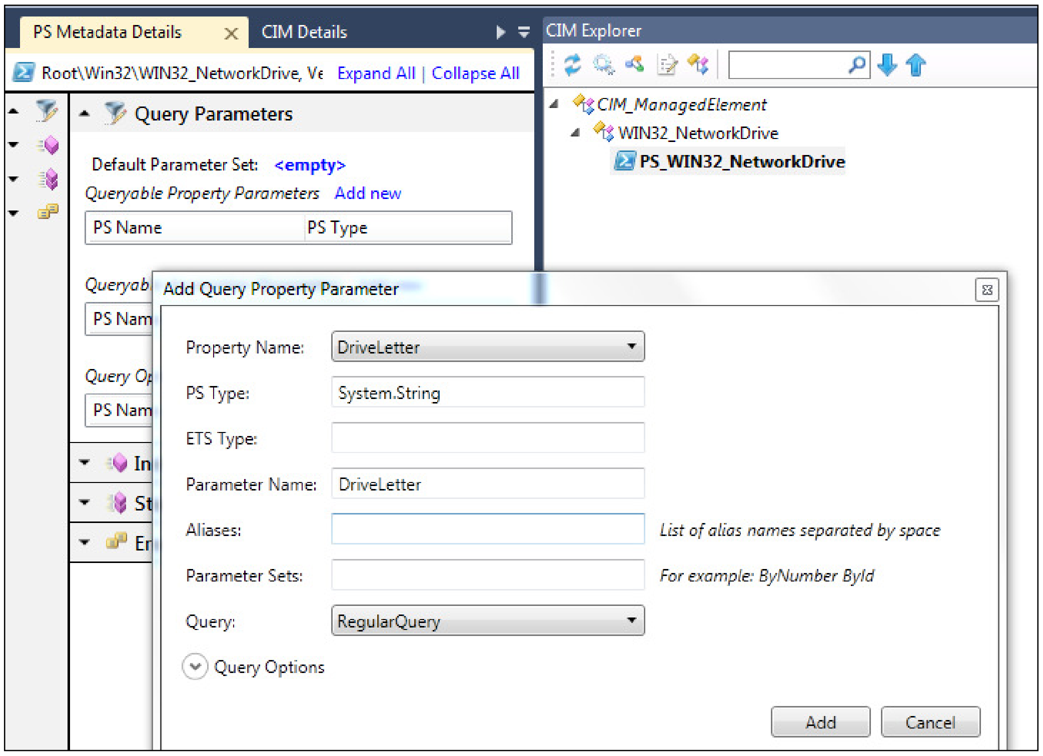
Notice in the following screenshot, that the file name generated is a CDXML file. A CDXML file is an XML structured file which contains all the metadata necessary for PowerShell to generate cmdlets for the provider:



Add PowerShell Metadata

Class Name:	Root\Win32\WIN32_NetworkDrive
Namespace:	Root\Win32
File Name:	PS_WIN32_NetworkDrive.cdxml
Version:	2.0.0.0
Default Noun:	WIN32NetworkDrive
Default Parameter Set:	
Project:	NetworkDriveProvider

Once we click on **Add** in the window, a new file will be added to the project. We will use this file later to cmdletize the provider. The file created by this dialog is empty and it requires us to define the mappings between cmdlets and the WMI objects. By clicking on the **PowerShell Metadata** file in the **CIM Explorer** window we can utilize the new editor to create this mapping. The window is shown in the following screenshot:


PS Metadata Details
CIM Details
Root\Win32\WIN32_NetworkDrive, Ve
Expand All | Collapse All
Query Parameters
Default Parameter Set: <empty>
Queryable Property Parameters Add new
PS Name
PS Type
CIM Explorer
CIM_ManagedElement
WIN32_NetworkDrive
PS_WIN32_NetworkDrive
Add Query Property Parameter

Property Name: DriveLetter
PS Type: System.String
ETS Type:
Parameter Name: DriveLetter
Aliases: *List of alias names separated by space*
Parameter Sets: *For example: ByNumber ById*
Query: RegularQuery
Query Options

The editor allows us to specify cmdlets for querying existing instances, for both instance based and static methods, and even enumeration values. This requires some knowledge of good cmdlet design but is very straightforward. Once we have mapped the properties of the CIM class to the parameters or the methods to the cmdlets, we will then have a CDXML file that contains the information that is necessary for creating cmdlets. We will see how to do this in the next section.

In order to create a cmdlet which queries the existing instances of the `Win32_NetworkDrive` class, we will want to add query parameters in the PowerShell metadata details. The property name drop-down allows us to select the properties of the CIM class and map them to a cmdlet property. To create a method we can right-click on the **Instance Method** section and map the `Disconnect` CIM method to a cmdlet `Disconnect-Win32_NetworkDrive`.

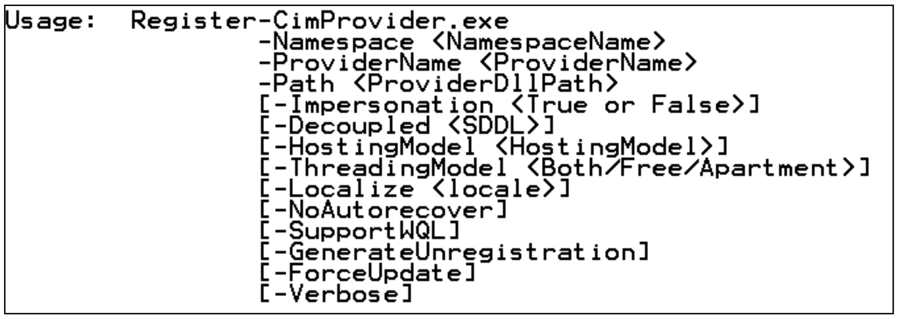
The product of a CIM Provider project is a DLL which will then be loaded into WMI as a provider and the CDXML file we just generated. The next section will focus on registering such a provider to offer the newly created functionality to WMI consumers, including PowerShell.

Registering a WMI provider

Registering a WMI provider is very simple in the new version of the Windows Management Framework. There is a new executable which has been added to aid in registration. It offers a PowerShell-like syntax. The executable, `Register-CimProvider`, is stored within the system directory, so it can easily be run just by typing its name. To learn more about how to use the program, use the `Help` parameter. All the parameters will be listed with a description of what each one does. This can be done using the following command:

```
PS C:\> Register-CimProvider -help
```

The output is as shown in the following screenshot:



```
Usage:  Register-CimProvider.exe
                -Namespace <NamespaceName>
                -ProviderName <ProviderName>
                -Path <ProviderDllPath>
                [-Impersonation <True or False>]
                [-Decoupled <SDDL>]
                [-HostingModel <HostingModel>]
                [-ThreadingModel <Both/Free/Apartment>]
                [-Localize <locale>]
                [-NoAutorecover]
                [-SupportWQL]
                [-GenerateUnregistration]
                [-ForceUpdate]
                [-Verbose]
```

When registering a CIM provider, there are three parameters that are required by `Register-CimProvider`. These include `Namespace`, `Path`, and `ProviderName`. The `Namespace` parameter defines within which namespace the provider will be registered. Namespaces are groupings of functionally similar providers that are referenced by a backslash separated path beginning with the `Root`. For example, there is a `Root\Virtualization` namespace which contains providers for managing Hyper-V machines. The `Path` parameter is used to specify the path to the DLL which was generated in the previous section. Finally, the `ProviderName` parameter is a unique name given to the provider. It should be easily recognizable what the provider is intended to expose by its name. To register our new CIM provider we will use the following command. It is required that we run the PowerShell console as an administrator due to security restrictions of the WMI registry. Attempting to run `Register-CimProvider` in a prompt which is not elevated will result in an `Access Denied` error:

```
PS C:\> Register-CimProvider -Namespace Root\WIN32 -ProviderName
WIN32NetworkDriveProvider -Path "NetworkDriveProviderImpl.dll"
Successfully registered the provider.
```

Once our provider is registered we can begin to access it via PowerShell. We can first verify that registration was a success by returning the definition of our class. This can be done using the `Get-CimClass` cmdlet as follows:

```
PS C:\ > Get-CimClass -ClassName Win32_NetworkDrive -Namespace Root\WIn32

   NameSpace: ROOT/WIN32

ClassName           Methods        Properties
---------           -------        ----------
WIN32_NetworkDrive  {Disconnect}   {Caption,Description...}
```

As we can see, the class has been successfully registered in the `Root/Win32` namespace. It has also returned the correct methods and properties, as defined in our MOF file in the previous section. During the registration process a fully defined MOF file is generated. If we examine the file system next to where the DLL for the provider is situated, we will see that a MOF file with the same name as the provider has been generated. If we wish to unregister the provider, we can use the `GenerateUnregistration` parameter on `Register-CimProvider`. This generates a MOF file which can be used to unregister the provider.

To unregister a provider using the unregistration MOF file we would utilize the `mofcomp` executable which comes with Windows. The `mofcomp` parses and compiles MOF files before adding them to the WMI repository. If we provide `mofcomp` with a path to the unregistration MOF file, it will remove the class from the `Root/Win32` namespace. This is done using the following commands:

```
PS C:\> mofcomp "WIN32NetworkDriveProviderUninstall.mof"

Microsoft (R) MOF Compiler Version 6.2.8158.0
Copyright (c) Microsoft Corp. 1997-2006. All rights reserved.
Parsing MOF file: C:\ WIN32NetworkDriveProviderUninstall.mof
MOF file has been successfully parsed
Storing data in the repository...
...
Done!

PS C:\ > Get-CimClass -ClassName Win32_NetworkDrive -Namespace Root\WIn32

Get-CimClass : NOT_FOUND
At line:1 char:1
+ Get-CimClass -ClassName Win32_NetworkDrive -Namespace Root\WIn32
```

Automatically creating provider cmdlets

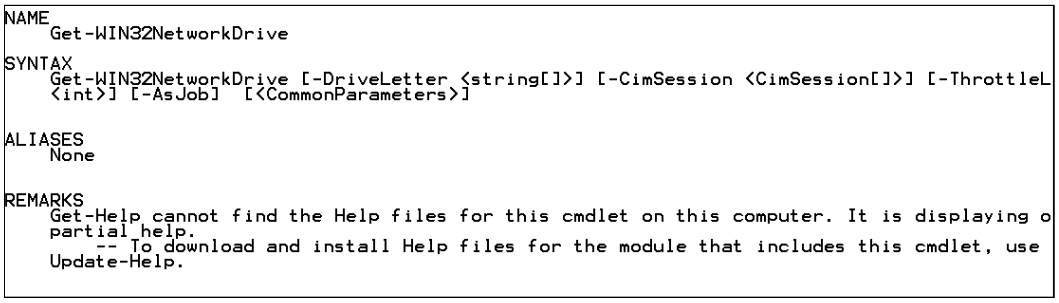
One of the major advantages to the new provider model and the CIM IDE is that the metadata created by Visual Studio can be directly consumed by PowerShell. Instead of having to write functions that expose the functionality, or by using the generic CIM or WMI functions, we can just create specific functions for our provider. In the previous section we created PowerShell metadata as a CDXML file. This file is responsible for generating the functions. In order to generate the functions, we just import the metadata file using `Import-Module` as follows:

```
PS C:\> Import-Module "C:\PS_WIN32_NetworkDrive.cdxml" -Verbose
VERBOSE: Importing function 'Disconnect-WIN32NetworkDrive'.
VERBOSE: Importing function 'Get-WIN32NetworkDrive'.
```

Now that they are imported we can use them just like we would any other function. If we investigate the syntax of the function we will also see that the parameters that we specified in the CIM IDE are exposed. This is done by using the following command:

```
PS C:\> Get-Help Get-WIN32NetworkDrive
```

This produces output as shown in the following screenshot:



```
NAME
    Get-WIN32NetworkDrive

SYNTAX
    Get-WIN32NetworkDrive [-DriveLetter <string[]>] [-CimSession <CimSession[]>] [-ThrottleL
    <int>] [-AsJob]  [<CommonParameters>]

ALIASES
    None

REMARKS
    Get-Help cannot find the Help files for this cmdlet on this computer. It is displaying o
    partial help.
        -- To download and install Help files for the module that includes this cmdlet, use
    Update-Help.
```

The auto-generated function even exposed the `CimSession` parameter, so that we can tie it in with a CIM session. We will investigate how this is useful in a later section. We can also see that the `DriveLetter` parameter is available to us. This enables us to filter the `WIN32_NetworkDrive` classes by the `DriveLetter` parameter. And we didn't have to write any PowerShell script to do this. As the CDXML file is processed with the `Import-Module` cmdlet, we can add it to modules as a `RequiredModule` to offer even more functionality to our modules.

Why new cmdlets to interact with WMI?

The WMI cmdlets in the previous version of PowerShell were very handy. Some of the most useful modules developed have been based on WMI. The biggest reason for developing new cmdlets is that the existing cmdlets are designed to only interact with WMI. Rather than simply interacting with Windows based systems, the CIM cmdlets are intended to communicate via the standard CIM protocol. This is why there are several cmdlets that are seemingly exactly the same as their WMI predecessor. The idea is to expose a standard cmdlet which will be able to interact with Windows systems, Linux devices, or other machines running some implementation of a CIM service. This enables Windows administrators to manage more than just Windows.

In addition to the ability to traverse different operating systems, the CIM cmdlets can also take advantage of the WSMan protocol when communicating with remote machines. This helps to alleviate firewall issues that are commonly encountered when using DCOM. Each of the cmdlets supports a `Protocol` parameter which allows us to specify either DCOM or WSMan as the communication protocol.

Differences between the CIM and WMI cmdlets

The CIM cmdlets are very similar to the WMI cmdlets that have been in PowerShell. For example, we can accomplish all the same things that we could by using the `Get-WmiObject` class with the `Get-CimInstance` class.

Both of the cmdlets even offer the same filtering mechanism found in WMI. As we investigate further, we will see that one of the biggest differences between the CIM and WMI cmdlets is that they handle remote connections differently. In Windows 8, the WSMan service will be on by default. Rather than having to configure remoting via PowerShell, this service will be the end point of choice. In the past WMI has communicated over the **Distributed Component Object Model** or **DCOM**. DCOM is notorious for being slow and synchronous. It is also considered a security concern by some. The WSMan protocol and service is a much better network stack. It offers superior performance and security. This allows PowerShell, and other services using WSMan, to scale much better to larger environments. It also allows for communication beyond the Windows stack.

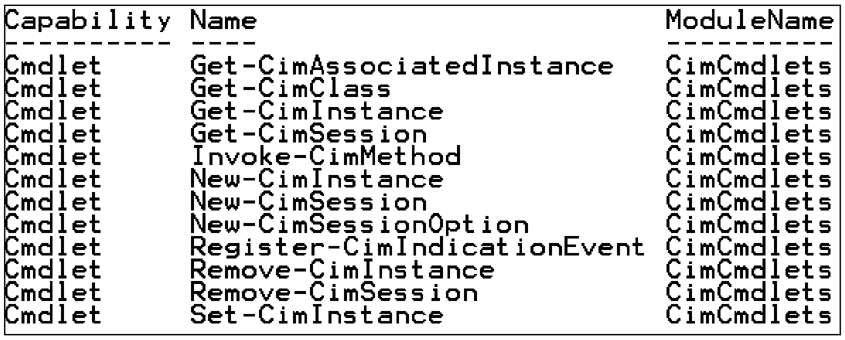
The difference between the CIM cmdlets and the WMI cmdlets is that the former communicate via both the WSMan and DCOM protocols and the latter communicate only via DCOM. In the next section we will investigate how to configure remote sessions with the new CIM cmdlets.

Cmdlet differences between CIM and WMI

In addition to the `Get-CimClass` cmdlet, there are several other new CIM based cmdlets. We can get a full list by typing the following command:

```
PS C:\> Get-Command -Module CimCmdlets
```

This gives the list as follows:

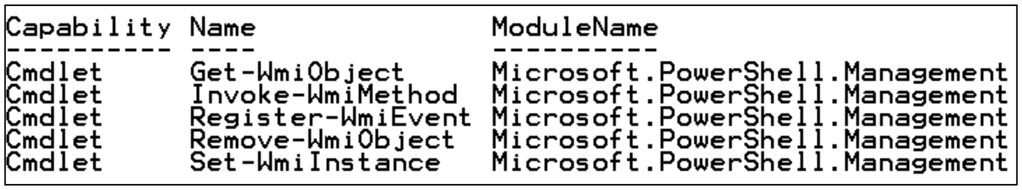


```
Capability Name                          ModuleName
---------- ----                          ----------
Cmdlet     Get-CimAssociatedInstance     CimCmdlets
Cmdlet     Get-CimClass                  CimCmdlets
Cmdlet     Get-CimInstance               CimCmdlets
Cmdlet     Get-CimSession                CimCmdlets
Cmdlet     Invoke-CimMethod              CimCmdlets
Cmdlet     New-CimInstance               CimCmdlets
Cmdlet     New-CimSession                CimCmdlets
Cmdlet     New-CimSessionOption          CimCmdlets
Cmdlet     Register-CimIndicationEvent   CimCmdlets
Cmdlet     Remove-CimInstance            CimCmdlets
Cmdlet     Remove-CimSession             CimCmdlets
Cmdlet     Set-CimInstance               CimCmdlets
```

Another obvious difference between the two sets of cmdlets is that there are many more CIM cmdlets than there are WMI cmdlets. We can get a list of WMI cmdlets by typing the following command:

```
PS C:\> Get-Command *WMI* -Type Cmdlet
```

The list of WMI cmdlets is as follows:



```
Capability Name                ModuleName
---------- ----                ----------
Cmdlet     Get-WmiObject       Microsoft.PowerShell.Management
Cmdlet     Invoke-WmiMethod    Microsoft.PowerShell.Management
Cmdlet     Register-WmiEvent   Microsoft.PowerShell.Management
Cmdlet     Remove-WmiObject    Microsoft.PowerShell.Management
Cmdlet     Set-WmiInstance     Microsoft.PowerShell.Management
```

Furthermore, the cmdlets found in the CIM module, are different in terms of parameters and sometimes intent. Even though several of the cmdlets seem to overlap in functionality, they are used in different ways. We can see this by comparing the `Invoke-WmiMethod` and `Invoke-CimMethod` cmdlets. They are intended to invoke one of the methods on a particular WMI object. For example, if we wanted to start a new process, such as the PowerShell command line, we could utilize the `Create` method of the `Win32_Process` class. To do this with the `Invoke-WmiMethod` cmdlet, we would do the following:

```
PS C:\> Invoke-WmiMethod -Class Win32_Process -Name Create -ArgumentList
"powershell.exe"
```

The `Create` method takes several parameters, the first being the command line for the process to start. In this example we are passing in the command line `powershell.exe`. As the PowerShell command line executable is on the default search path for Windows, the process will start.

[For processes that are not on the default search path, it is required to provide the full path name.]

In PowerShell 3.0 the same can be accomplished using the `Invoke-CimMethod` cmdlet. The difference is in the name of the parameters and the type of the arguments list. Notice that the `Invoke-WmiMethod` accepts an array for a parameter list. This works fine unless we want to specify only a particular parameter and not all the parameters in order. For example, if we look at the definition of the `Win32_Process Create` method we can see it takes more than just a single parameter as follows:

```
uint32 Create(
  [in]   string CommandLine,
  [in]   string CurrentDirectory,
  [in]   Win32_ProcessStartup ProcessStartupInformation,
  [out]  uint32 ProcessId
);
```

If we wanted to specify more arguments for the `Create` method in using the `Invoke-WmiMethod` cmdlet, we would simply add those arguments to the array in order. One of the downfalls of this type of positional specification is that we have to specify each parameter, even if we don't intend on using it. If we wanted to specify the `ProcessStartupInformation` parameter, we would also have to specify the `CurrentDirectory` because it is positioned in front of the `ProcessStartupInformation` parameter. This is because `Invoke-WmiMethod` finds parameters by position and not by name. The `Invoke-CimMethod` cmdlet works differently, as shown in the following commands. It accepts a hash table as a parameter list, thus allowing us to just specify the parameters we want to use:

```
PS C:\> Invoke-CimMethod –ClassName Win32_Process –MethodName Create –
Arguments @{CommandLine="powershell.exe"}
```

Although the syntax is a bit more typing, it allows for much more control over which parameters are specified when the method is invoked. The cmdlets look the same on the surface but there is enough change to warrant studying the differences.

Registering CIM events

Another cmdlet which maps between the CIM cmdlets and WMI cmdlets is the `Register-WmiEvent` cmdlet and the `Register-CimIndicationEvent`. Both of these cmdlets aim to allow us to be signalled when a particular WMI event has taken place. Once signalled, we can execute a script block which takes a particular action.

As an example we can set up an event listener which will output the name of a process whenever one is started. For this example, we will have to run the PowerShell command line as an administrator as follows:

```
PS C:\ > Register-WmiEvent -Query "Select * FROM Win32_ProcessStartTrace"
-Action { Write-Host ($Event.SourceEventArgs.NewEvent.ProcessName) }
-SourceIdentifier ProcessListener
```

This registration command will output the name of the process whenever a process is started. The `SourceIdentifier` is used to find the registered event later. This makes it easier for us to remove the event later.

Note the object model found within the script block. We need to drill down through the set of properties just right to get to the `ProcessName` property.

To test it out we can start notepad using the `Start-Process` cmdlet. Once `Notepad` is started, the event will signal and we will see the process name written to the host as shown in the following commands:

```
PS C:\> Start-Process Notepad
PS C:\> notepad.exe
```

This is possible through the CIM cmdlets. In order to accurately test the functionality, we should unregister the previous event that we set into place. This can be done using the `Unregister-Event` cmdlet as follows:

```
PS C:\> Unregister-Event ProcessListener
```

Now that the previous event listener has been removed, we can add a new one using the `Register-CimIndicationEvent`. The syntax for this cmdlet is almost identical to that of the previous example with the WMI cmdlet. The following commands show this:

```
PS C:\ > Register-CimIndicationEvent -Query "Select * FROM Win32_
ProcessStartTrace" -Action { Write-Host ($Event.SourceEventArgs.NewEvent.
ProcessName) } -SourceIdentifier ProcessListener
```

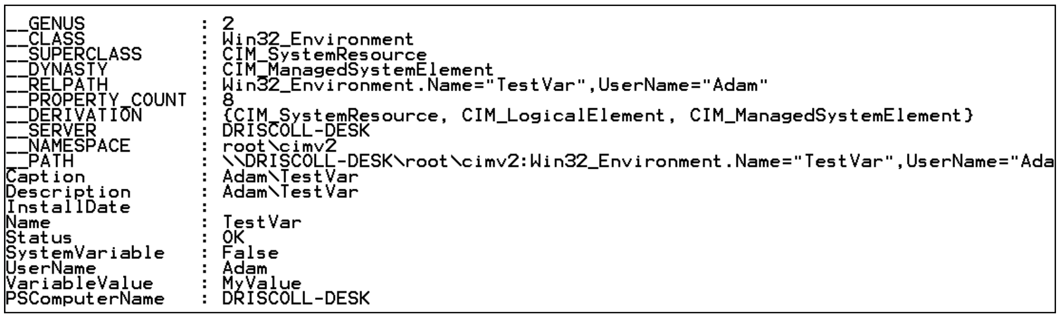
The event listener can even be unregistered the same way as the previous WMI event using the exact same syntax as used in the previous examples by using the `Unregister-Event` cmdlet.

Updating CIM instances

The final comparison to outline is that between the `Set-WmiInstance` and `Set-CimInstance` cmdlets. It would seem that the goal of both of these cmdlets is to offer a way to update instances of properties of WMI objects. We will see why this isn't necessarily the case. Both of the cmdlets accept a hash table of names and values to set particular properties on a WMI object. If, for instance, we wanted to set a new `Environment` variable, we could use the `Set-WmiInstance` cmdlet as follows:

```
PS C:\> Set-WmiInstance -Class Win32_Environment -Arguments @{Name="TestV
ar";VariableValue="MyValue";UserName="Adam" }
```

This produces the hash table as shown in the screenshot:



```
__GENUS            : 2
__CLASS            : Win32_Environment
__SUPERCLASS       : CIM_SystemResource
__DYNASTY          : CIM_ManagedSystemElement
__RELPATH          : Win32_Environment.Name="TestVar",UserName="Adam"
__PROPERTY_COUNT   : 8
__DERIVATION       : {CIM_SystemResource, CIM_LogicalElement, CIM_ManagedSystemElement}
__SERVER           : DRISCOLL-DESK
__NAMESPACE        : root\cimv2
__PATH             : \\DRISCOLL-DESK\root\cimv2:Win32_Environment.Name="TestVar",UserName="Ada
Caption            : Adam\TestVar
Description        : Adam\TestVar
InstallDate        :
Name               : TestVar
Status             : OK
SystemVariable     : False
UserName           : Adam
VariableValue      : MyValue
PSComputerName     : DRISCOLL-DESK
```

In order to actually use the `Environment` variable we would have to restart the current PowerShell command prompt. The variable will even be present in the `Environment` variables window found in the **Windows System Properties** dialog. This is a strange way to use a `Set` cmdlet. Rather than actually setting the properties on an existing instance of a class, it has created a new instance of the `Environment` variable class. The CIM cmdlets do not behave this way. The `Set-CimInstance` does not even expose a way of specifying the class name. It requires that we use the `Get-CimInstance` cmdlet to return the existing CIM instance that we wish to set.

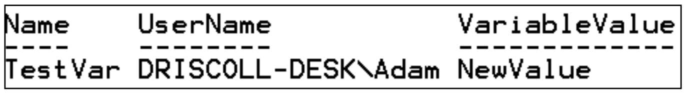
To retrieve the `Win32_Environment` instance that we just created we could use the `Get-CimInstance` cmdlet with a filter on the name as follows:

```
PS C:\> Get-CimInstance -ClassName Win32_Environment -Filter
'Name="TestVar"'

Name              UserName                VariableValue

----              --------                -------------

TestVar           DRISCOLL-DESK\Adam      MyValue
```

As we can see it returned the same value we just set using the WMI cmdlet. If we wanted to modify the value, we would then pipe the output of the `Get-CimInstance` to the `Set-CimInstance` cmdlet. The `Properties` parameter on `Set-CimInstance` accepts a hash table, just like the `Arguments` of the `Set-WmiInstance`:

```
PS C:\> Get-CimInstance -ClassName Win32_Environment -Filter
'Name="TestVar"' | Set-CimInstance -Property @{Value="NewValue"}

PS C:\> Get-CimInstance -ClassName Win32_Environment -Filter
'Name="TestVar"'
```

The output is as shown in the following screenshot:



```
Name    UserName           VariableValue
----    --------           -------------
TestVar DRISCOLL-DESK\Adam NewValue
```

In this circumstance, the `Set-CimInstance` behaves much more like other `Set` cmdlets found in PowerShell.

One final thing to note is the difference between the output of the CIM and WMI cmdlets. Rather than returning a large list of the properties of the CIM instances, a much more succinct, readable format is returned. This is because the objects returned by the CIM cmdlets are actually of a different type than that of the WMI cmdlets. The WMI objects are returned as `System.Management.ManagementObject` and the CIM cmdlets return `Microsoft.Management.Infrastructure.CimInstance`. Refer to the following commands:

```
PS C:\> Get-WmiObject -Class Win32_Environment | Get-Member
TypeName: System.Management.ManagementObject#root\cimv2\Win32_Environment
...


PS C:\> Get-CimInstance -ClassName Win32_environment | Get-Member


TypeName: Microsoft.Management.Infrastructure.CimInstance#root/cimv2/
Win32_Environment
```

In addition to modifying the existing functionality found in the WMI cmdlets for interacting with CIM systems, Microsoft has also extended the reach of the CIM capabilities. We will see in the next section what new functionality is available to us by way of WMI and CIM.

Exploring the new CIM cmdlets

In the previous section we examined what was different between the existing WMI cmdlets and the new CIM based cmdlets. In this section we will look at what additional functionality has been added to the CIM module which does not appear in the WMI cmdlets.

CIM sessions

Using the WMI cmdlets, a connection was made for each command. If we needed to specify credentials it would need to be done every time the WMI cmdlet was invoked. This leads to a lot of extra typing and work. This can also make it difficult to work with a large group of computers. Although possible with remoting, there is a new mechanism which the CIM cmdlets can harness. A CIM session is a lot like a remote session started with `New-PSSession`. The big difference is that it is not designed to be used exclusively by PowerShell. Instead it is intended to be used only upon the CIM framework. This enables the session type behavior to traverse more than just Windows machines.

To get a new CIM session started we can use a combination of the `New-CimSession` cmdlet and the `New-CimSessionOption` cmdlet. You will notice that this is very similar to a `New-PSSession` and `New-PSSessionOption` cmdlets. Maintaining the same type of structure makes learning the new cmdlets much easier. The `New-CimSession` cmdlet has parameters that set up the basics of the CIM session such as computer name, credentials, and authentication type. The `New-CimSessionOption` has parameters for more advanced configuration of the session including protocol type and certificate settings.

If our computer is not part of the domain we will see a problem by utilizing the `New-CimSession` without any additional configuration. By default, the authentication type is `Kerberos`, which only works in a domain environment. To be able to connect to a workgroup computer we need to configure WSMan on the local machine to either accept HTTPS connections or add the host we wish to connect to the trusted hosts list.

To add a computer to the list of the trusted hosts, we can run the following command. Please note that this list should be restricted to a short list of machines as they may not be authenticated.

```
PS C:\> winrm set winrm/config/client '@{TrustedHosts ="driscoll-
laptop"}'
```

Now it is possible for my `driscoll-desk` machine to connect to my laptop without any actual authentication. This obviously is not the ideal situation for any type of production environment. In this circumstance, we can just use Kerberos to authenticate against the domain which both machines are a part of. To get a simple new CIM session started we can use the `New-CimSession` cmdlet by itself as follows:

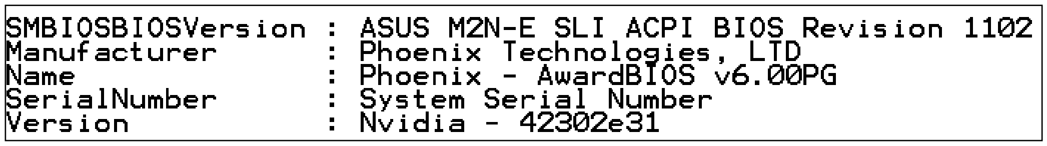
```
PS C:\> New-CimSession -ComputerName Driscoll-laptop

Id          : 1

Name        : CimSession1
```

```
InstanceId   : f415359f-d392-4a2e-80cc-5043ec128549
ComputerName : Driscoll-laptop
Protocol     : WSMAN
```

The new CIM session is now open and we can begin utilizing it to issue commands to the remote machine. Notice that by default, CIM sessions utilize the WSMan protocol instead of DCOM. The type of communication mechanism can be changed using the options provided by the `New-CimSessionOption` cmdlet. To utilize this session with another one of the CIM cmdlets we can specify the name or ID on the `CimSession` parameter of the other CIM cmdlet. For example, if we wanted to list to view the BIOS information on the remote machine, we could do the following:

```
PS C:\> Get-CimInstance –ClassName Win32_BIOS –CimSession 1
```

This gives the following output:



```
SMBIOSBIOSVersion : ASUS M2N-E SLI ACPI BIOS Revision 1102
Manufacturer      : Phoenix Technologies, LTD
Name              : Phoenix - AwardBIOS v6.00PG
SerialNumber      : System Serial Number
Version           : Nvidia - 42302e31
```

To view all open sessions in the current PowerShell command prompt, we can utilize the `Get-CimSession` cmdlet. It will return all open CIM sessions. It also lets us filter by computer name, ID, or instance ID as follows:

```
PS C:\> Get-CimSession
Id           : 1
Name         : CimSession1
InstanceId   : f415359f-d392-4a2e-80cc-5043ec128549
ComputerName : Driscoll-laptop
Protocol     : WSMAN
```

To close our CIM session we can utilize the `Remove-CimSession` cmdlet. It takes pipeline input. It should be used in conjunction with the `Get-CimSession` cmdlet. Removing a CIM session will close the communication between the two machines and this is done using the following command:

```
PS C:\> Get-CimSession –Id 1 | Remove-CimSession
```

Creating new CIM instances

When we looked at the differences between the WMI cmdlets and the CIM cmdlets we saw that the `Set-WmiInstance` cmdlet behaved almost like a `New-WmiInstance` cmdlet (which does not exist). The CIM cmdlets offer a cmdlet which is intended to do this, and is correctly named. The `New-CimInstance` cmdlet is used for creating new instances of CIM classes that support this type of creation. We can use the previous example of the `Environment` variable to illustrate this point. Instead of calling `Set-CimInstance`, which we saw previously is intended for setting properties on existing instances, we will use the `New-CimInstance` cmdlet.

The `New-CimInstance` cmdlet accepts a class name and a hash table of properties to enable the creation of these instances. To create a new `Environment` variable we can do the following:

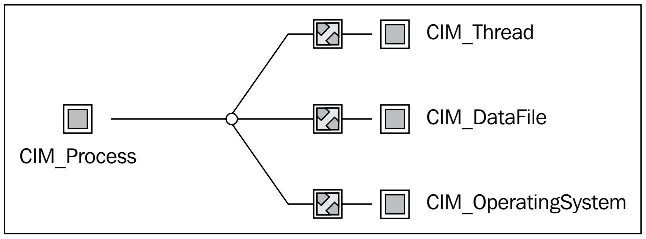
```
PS C:\> New-CimInstance -ClassName Win32_Environment -Property @{Name="Te
stVar";VariableValue="MyValue";UserName="Adam" }

Name              UserName              VariableValue

----              --------              -------------

TestVar           DRISCOLL-DESK\Adam    MyValue
```

The `New-CimInstance` cmdlet is just as easy to use as the `Set-WmiInstance` we used earlier. In addition to creating CIM instances with a simple cmdlet, we can remove them as well, with a similar, cmdlet-based approach.

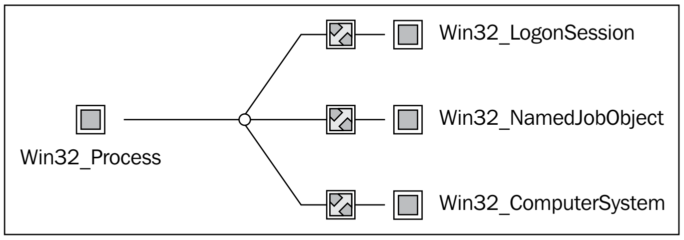
Working with associated classes

One concept in CIM is the idea of associated classes. Classes can be related to each other. Instead of merely having a simple property, for example a service name, a CIM class may have a complex object which is related to it. In the previous version of PowerShell it was possible to retrieve associated classes but only by utilizing the complex **WMI Query Language** (**WQL**), which the WMI cmdlets were capable of using. With the new `CIMCmdlets` module, comes a new cmdlet, the `Get-CimAssociatedInstance` cmdlet. This greatly simplifies how we have got associated references to the classes we are retrieving.

The easiest example is that of a process. A process is composed of several components. It runs on an operating system, under the context of a particular user and is often made up of one or more files. Each of these particular properties could not effectively be displayed in a single string on the `Win32_Process` object. Instead, they are referenced through different classes that are linked. We can see this in the following screenshot:



The `Win32_Process` class is said to inherit from the `CIM_Process` class. This means that all the properties, methods, and associations that the `CIM_Process` exposes, the `Win32_Process` class exposes as well. They are related like a fruit is to an apple. Just as the apple is a more specialized fruit, the `Win32_Process` is a more specialized type of `CIM_Process`. It offers most specialized properties that pertain to only a Windows based process, rather than a more general CIM process:

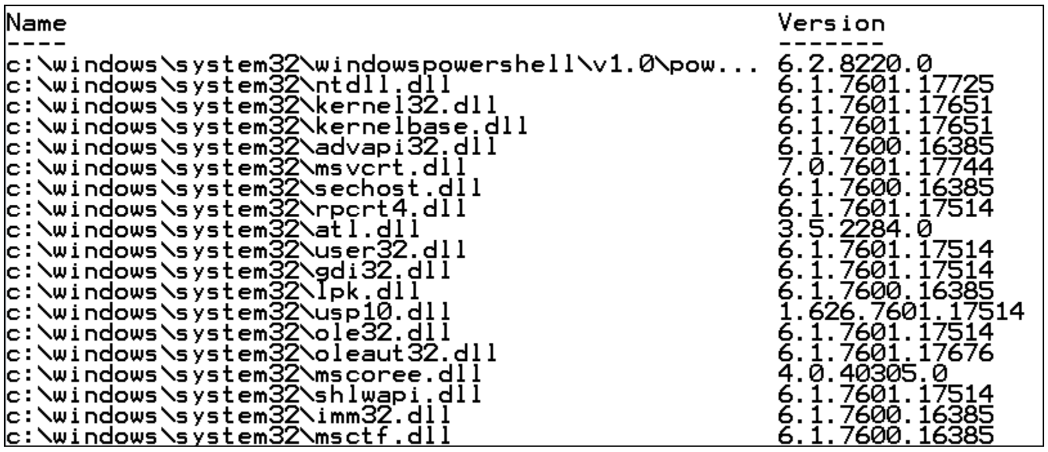


As we can see from the screenshot, the `Win32_Process` is linked to the `Win32_ComputerSystem` and `Win32_LogonSession`. The `CIM_Process` is also linked to the `CIM_Thread`, `CIM_DataFile`, and `CIM_OperatingSystem`. As the `Win32_Process` is really just a subset of the `CIM_Process`, which means it is also linked to the `CIM_Thread` and `CIM_DataFile` classes. The icon denoted by the two arrows between the two empty boxes is the **association class**. Association classes are the glue that holds together the two classes on either side of the screenshot. As both the `Win32_Process` and `Win32_LogonSession` are technically independent of each other, CIM requires that we add a linking class in between so that we can associate the classes together. The linking class is what we are particularly interested in, when using the new CIM associated instance cmdlet.

In order to query these particular items in .NET or previous versions of PowerShell, we would need to write a complex WQL query. In PowerShell 3.0 this is no longer the case. Instead we can utilize the `Get-CimAssociatedInstance` cmdlet. It accepts pipeline input from the `Get-CimInstance` cmdlet and allows us to retrieve the linked classes. If we want to get the associated `Cim_DataFiles` that the `PowerShell.exe` process is currently using, we can do the following:

```
PS C:\> Get-CimInstance -ClassName Win32_Process -Filter 'Name =
"PowerShell.exe"' | Get-CimAssociatedInstance -Association
Cim_ProcessExecutable | Select-Object Name,Version
```

This produces the following list:



```
Name                                                      Version
----                                                      -------
c:\windows\system32\windowspowershell\v1.0\pow...         6.2.8220.0
c:\windows\system32\ntdll.dll                             6.1.7601.17725
c:\windows\system32\kernel32.dll                          6.1.7601.17651
c:\windows\system32\kernelbase.dll                        6.1.7601.17651
c:\windows\system32\advapi32.dll                          6.1.7600.16385
c:\windows\system32\msvcrt.dll                            7.0.7601.17744
c:\windows\system32\sechost.dll                           6.1.7600.16385
c:\windows\system32\rpcrt4.dll                            6.1.7601.17514
c:\windows\system32\atl.dll                               3.5.2284.0
c:\windows\system32\user32.dll                            6.1.7601.17514
c:\windows\system32\gdi32.dll                             6.1.7601.17514
c:\windows\system32\lpk.dll                               6.1.7600.16385
c:\windows\system32\usp10.dll                             1.626.7601.17514
c:\windows\system32\ole32.dll                             6.1.7601.17514
c:\windows\system32\oleaut32.dll                          6.1.7601.17676
c:\windows\system32\mscoree.dll                           4.0.40305.0
c:\windows\system32\shlwapi.dll                           6.1.7601.17514
c:\windows\system32\imm32.dll                             6.1.7600.16385
c:\windows\system32\msctf.dll                             6.1.7600.16385
```

Notice that when calling the `Get-CimAssocaitedInstance` cmdlet, we did not use the `Cim_DataFile` class name. We instead use the class which links the two together; the glue. The `Cim_ProcessExecutable` class is an association class which links the two classes together. As the `Cim_DataFile` can be used by more than just processes, it is necessary to create a linking class.

There are several ways to query which associations a class may have. To do it in PowerShell we can use the `Get-WmiObject` cmdlet.

NanoWBEM

WMI has always been about managing Windows machines. Although the system was built on the foundation of the CIM standard, it was not possible to manage non-Windows CIM machines. Included with the Windows Management Framework, **NanoWBEM** will be a new service designed by Microsoft, to run on Linux computers or other devices to allow for management via the PowerShell CIM cmdlets. This is a huge leap forward for Windows administrators, because it allows us to use a central management tool to work in an infrastructure that is growing more complex and distributed.

Summary

The new CIM methodology which Microsoft has taken is a good route for the end user. Rather than being solely tied to Microsoft Windows, it is now possible to use Windows based management tools such as PowerShell, to step into the world of Linux. As we saw in this chapter, the new WMI v2 provider model and CIM IDE make developing WMI solutions much easier. The automatic cmdletization feature make working with our newly minted providers easier than ever. Time to market is greatly reduced because of the simplified model and tooling now available.

We also explored how the new CIM cmdlets go far beyond what the basic WMI cmdlets did previously. We examined the differences and similarities between the two sets of cmdlets. Finally, we took a quick look into NanoWBEM, the Linux based CIM implementation provided by Microsoft to aid in management of systems outside the Windows world.

In the next chapter, we will take a look at the new PowerShell Web Access and Integrated Scripting Environment that are available in Windows 8 and the Windows Management Framework.

6
New and Improved PowerShell Hosts

In PowerShell 2.0, there were only two ways to access it out of the box: through the command prompt, `PowerShell.exe`, or through the PowerShell **Integrated Scripting Environment** (**ISE**). On Windows Server the ISE was not even enabled by default and had to be enabled through a separate feature. The command prompt is typically the first stop for many users as it is simple and efficient. It allows us to quickly execute commands that we do not have to persist. Whenever users are looking to write something more than just a couple of lines and are looking to save their scripts, the ISE is the obvious choice.

In addition to the built-in PowerShell hosts, there are tons of third party PowerShell tools available to write scripts, create user interfaces, or execute command line operations. In PowerShell 3.0, Microsoft enhanced the built-in tooling. Not only did they enhance the ISE to be more robust and offer features found in most of the other script editors, but also in Windows 8 they have added a new feature to host PowerShell within the browser.

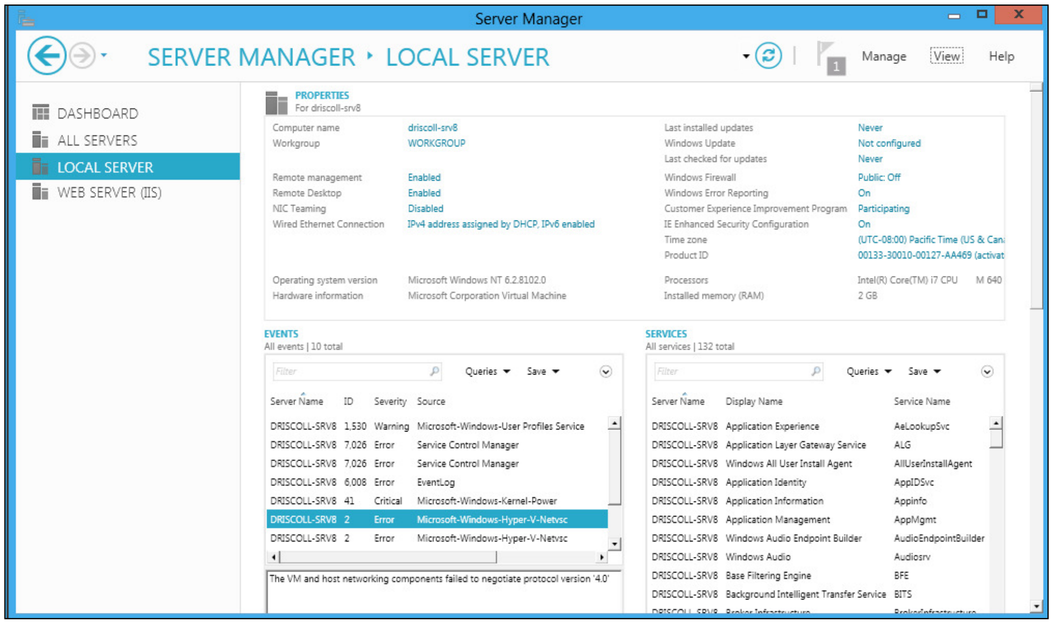
With the proliferation of mobile devices this comes as a welcome addition. Often administrators find themselves having to access resources, while having limited network access, or without hardware that is capable of running PowerShell. In this chapter we will take a look at the new PowerShell Web Access feature of the Windows Server 2012 to see what it offers. We will then dive into the new enhancements to the PowerShell ISE that makes authoring scripts even easier.

In this chapter we will cover:

- Installation, configuration, and usage of the new PowerShell Web Access feature of Windows Server 2012
- New features and enhancements to the PowerShell ISE

Installing the Windows PowerShell Web Access feature

The first step in using the Windows PowerShell Web Access is installing it. Windows Server 2012 does not have the Windows PowerShell Web Access feature enabled by default. We will have to enable the feature using the new **Server Manager**. The **Server Manager** has been redesigned to give a better dashboard view of the local server or distributed servers we want to manage. There is a new **All Servers** view which gives a better dashboard view of all the servers that are managed from the **Server Manager**. In order to simply configure the local server, there is a **Local Server** view. It provides a look into the properties, events, services, performance, and features of the server. It makes it much easier to discover information and manage the server without having to dig through a multitude of wizards and control panel applets. Each of the installed roles shares a similar view as the server dashboard as well, as seen in the following screenshot. The roles themselves report performance, events, and services:



PROPERTIES
For driscoll-srv8

Computer name	driscoll-srv8	Last installed updates	Never
Workgroup	WORKGROUP	Windows Update	Not configured
		Last checked for updates	Never
Remote management	Enabled	Windows Firewall	Public: Off
Remote Desktop	Enabled	Windows Error Reporting	On
NIC Teaming	Disabled	Customer Experience Improvement Program	Participating
Wired Ethernet Connection	IPv4 address assigned by DHCP, IPv6 enabled	IE Enhanced Security Configuration	On
		Time zone	(UTC-08:00) Pacific Time (US & Can
		Product ID	00133-30010-00127-AA469 (activat
Operating system version	Microsoft Windows NT 6.2.8102.0	Processors	Intel(R) Core(TM) i7 CPU M 640
Hardware information	Microsoft Corporation Virtual Machine	Installed memory (RAM)	2 GB

EVENTS
All events | 10 total

Server Name	ID	Severity	Source
DRISCOLL-SRV8	1,530	Warning	Microsoft-Windows-User Profiles Service
DRISCOLL-SRV8	7,026	Error	Service Control Manager
DRISCOLL-SRV8	7,026	Error	Service Control Manager
DRISCOLL-SRV8	6,008	Error	EventLog
DRISCOLL-SRV8	41	Critical	Microsoft-Windows-Kernel-Power
DRISCOLL-SRV8	2	Error	Microsoft-Windows-Hyper-V-Netvsc
DRISCOLL-SRV8	2	Error	Microsoft-Windows-Hyper-V-Netvsc

The VM and host networking components failed to negotiate protocol version '4.0'

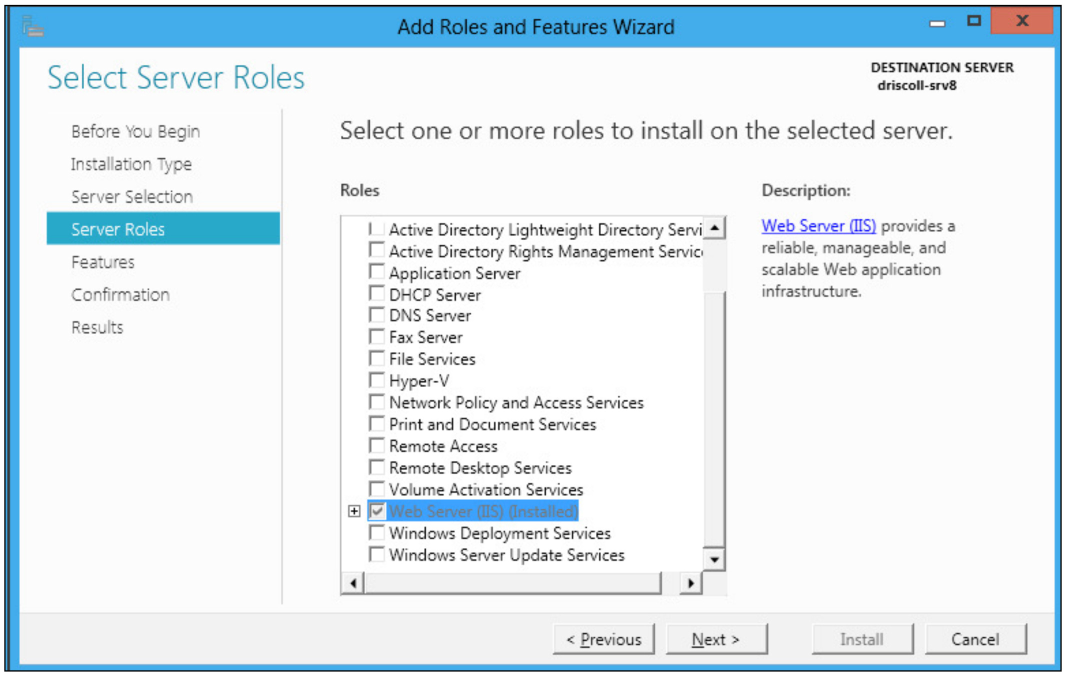
SERVICES
All services | 132 total

Server Name	Display Name	Service Name
DRISCOLL-SRV8	Application Experience	AeLookupSvc
DRISCOLL-SRV8	Application Layer Gateway Service	ALG
DRISCOLL-SRV8	Windows All User Install Agent	AllUserInstallAgent
DRISCOLL-SRV8	Application Identity	AppIDSvc
DRISCOLL-SRV8	Application Information	Appinfo
DRISCOLL-SRV8	Application Management	AppMgmt
DRISCOLL-SRV8	Windows Audio Endpoint Builder	AudioEndpointBuilder
DRISCOLL-SRV8	Windows Audio	Audiosrv
DRISCOLL-SRV8	Base Filtering Engine	BFE
DRISCOLL-SRV8	Background Intelligent Transfer Service	BITS
DRISCOLL-SRV8	Broker Infrastructure	BrokerInfrastructure

There are a few steps involved in installing and configuring the PowerShell Web Access. The first step is to install the **Web Server (IIS)** role. The **PowerShell Web Access** is a feature of the Web Server role. The **Internet Information Services** (**IIS**) will be responsible for hosting the web site. To install this role we need to step through the **Add Roles and Features Wizard**. This wizard can be accessed by navigating to the **Local Server** pane, clicking on the **Manage drop down menu** and selecting the menu item **Add Roles and Features**.

As we step through the menu, the first option we need to select is whether we want the feature- or role-based installation or the scenario-based installation. The scenario-based installation is only used for **Remote Desktop Services** so we need to select the **Role-Based** or **Feature-Based Installation** type. On the next page of the wizard that is **Server Selection**, we have the option to select one or more servers from our server pool. A **server pool** is a grouping of servers added to the **Server Manager** through the **Add Servers** option found in the **Manage drop down menu**. At this step we could configure the Web Server role and the **PowerShell Web Access** feature on any number of servers within the pool. As we are only looking to manage the current server we can simply click on next.

It is also possible to install the roles and features to a **Virtual Hard Disk** (**VHD**). We will choose not to do this now and instead merely install to the local server which we are managing. If nothing was changed during this page of the wizard we can simply move on. Now we need to select the **Web Server (IIS)** role found in the list of roles. This will allow the local server to host web applications. It is only required to install the **Management Tools** and the Web Server sub-roles found beneath the **Web Server (IIS)** role. This is shown in the following screenshot. Once checked we can advance to the **Features** page:



Select one or more roles to install on the selected server.

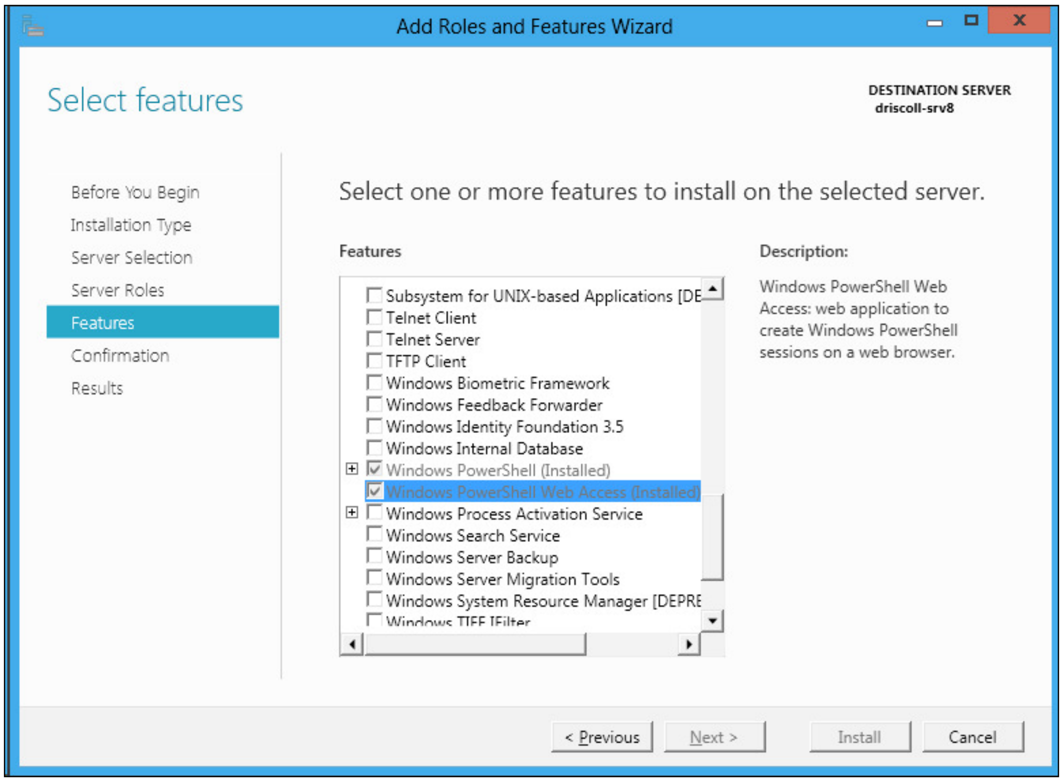
Roles

- [ ] Active Directory Lightweight Directory Servi
- [ ] Active Directory Rights Management Servic
- [ ] Application Server
- [ ] DHCP Server
- [ ] DNS Server
- [ ] Fax Server
- [ ] File Services
- [ ] Hyper-V
- [ ] Network Policy and Access Services
- [ ] Print and Document Services
- [ ] Remote Access
- [ ] Remote Desktop Services
- [ ] Volume Activation Services
- [✓] Web Server (IIS) (Installed)
- [ ] Windows Deployment Services
- [ ] Windows Server Update Services

Description:

Web Server (IIS) provides a reliable, manageable, and scalable Web application infrastructure.

On this page we need to select the **Windows PowerShell Web Access** feature. When this feature is installed, it will lay down the necessary pieces to install, configure, and host the web access. This is shown in the next screenshot:



Select one or more features to install on the selected server.

Features

- [ ] Subsystem for UNIX-based Applications [DE
- [ ] Telnet Client
- [ ] Telnet Server
- [ ] TFTP Client
- [ ] Windows Biometric Framework
- [ ] Windows Feedback Forwarder
- [ ] Windows Identity Foundation 3.5
- [ ] Windows Internal Database
- [x] Windows PowerShell (Installed)
- [x] Windows PowerShell Web Access (Installed)
- [ ] Windows Process Activation Service
- [ ] Windows Search Service
- [ ] Windows Server Backup
- [ ] Windows Server Migration Tools
- [ ] Windows System Resource Manager [DEPRE
- [ ] Windows TIFF IFilter

Description:

Windows PowerShell Web Access: web application to create Windows PowerShell sessions on a web browser.

We will be prompted to enable several other features if they are not already installed. These include features such as .NET. Finally, we can click on the **Install** button to install the pieces necessary for **Window PowerShell Web Access**. The pieces will not actually be installed into IIS. We will need to further configure the feature after it has been successfully installed.

In addition to enabling the feature through **Server Manager**, we can also utilize the `Server Manager PowerShell` module. It contains a cmdlet that can install features. The same steps taken previously could be replicated using the following cmdlet line:

```
Install-WindowsFeature -Name Web-Server, WindowsPowerShellWebAccess
```

PowerShell will need to be elevated by selecting the **Run As Administrator** option.

Configuring the Windows PowerShell Web Access

In the previous section we looked at what it would take to install the roles and features necessary to run the PowerShell Web Access. This alone does not fully configure the Web Access for use. We need to take some additional steps to install the correct pieces into the Web Server and configure it to use the **Secure Socket Layer** (**SSL**) to ensure that the web site traffic is fully encrypted.

Instructions on how to configure the PowerShell Web Access can be found in the feature's installation directory. This directory is `%systemroot%\Web\PowerShellWebAccess\wwwroot\`. Simply type the preceding path into the address bar of a **Windows Explorer** window to navigate to it.

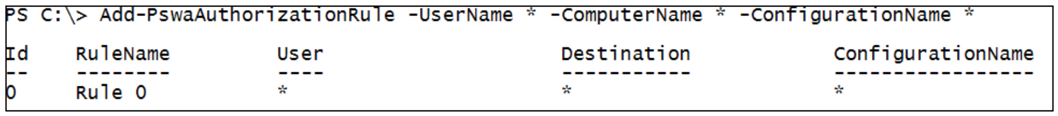
To install the PowerShell Web Access web application we can utilize the `PowerShellWebAccess` module. This module contains cmdlets for installing **PowerShell Web Access** and configuring user, computer, and configuration authorization. To quickly set up a **PowerShell Web Access** instance we can use the following command:

```
PS C:\> Install-PSWAWebApplication –UseTestCertificate
```

PowerShell Web Access only runs over HTTPS and thus requires a certificate. To set up a test certificate we can utilize the `UseTestCertificate` switch parameter as shown in this command. Once this command completes we will have a new application pool and website created for **PowerShell Web Access**.

Authorizing users

The next step is to authorize access for particular users, computers, and configurations. This can be accomplished using the `PSWAAuthorizationRule` cmdlets. To create a new rule we can utilize the command in the following screenshot:

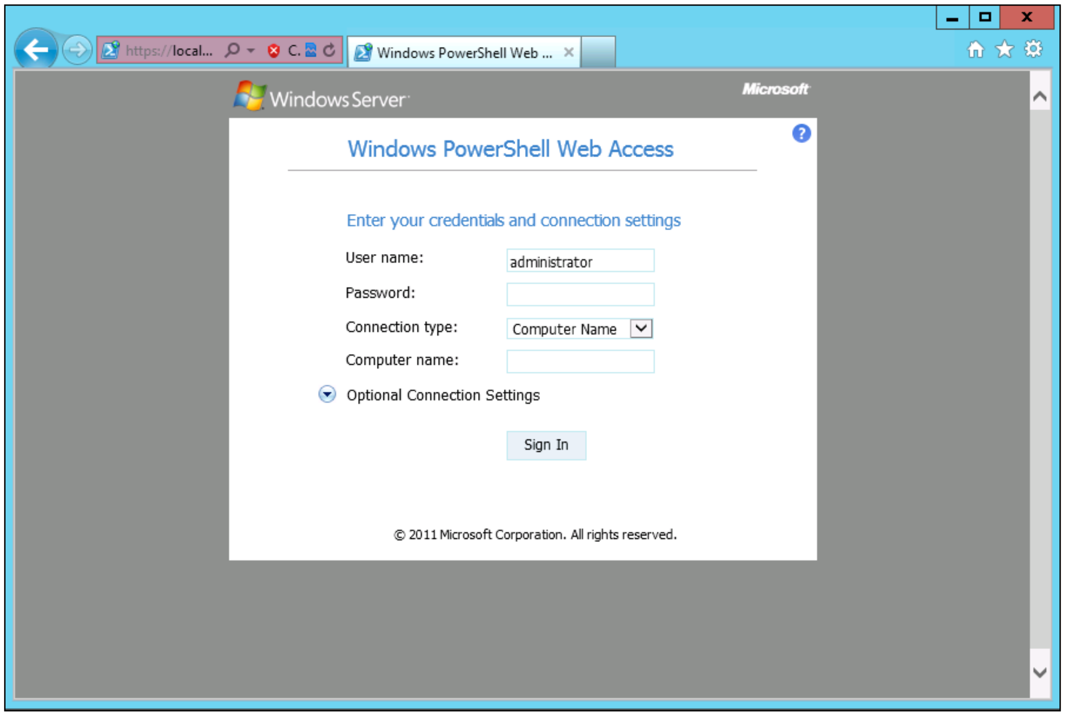


```
PS C:\> Add-PswaAuthorizationRule -UserName * -ComputerName * -ConfigurationName *

Id    RuleName        User                   Destination          ConfigurationName
--    --------        ----                   -----------          -----------------
0     Rule 0          *                      *                    *
```

Notice that wildcards are allowed. The authorization rule in this command will allow any user attempt to connect to any computer and/or configuration. Note that depending on the configuration of the target machine, the user may still not have access to it. Additional cmdlets are available for listing, removing, and testing authorization rules.

Accessing the PowerShell Web Access

Once we have finished installing and configuring the **PowerShell Web Access** we should be able to navigate to the URL in any web browser. To ensure everything is working correctly, launch Internet Explorer and navigate to `https://localhost/pswa` (assuming that you installed the web access role on the local system that you are running Internet Explorer from). We should see the following screenshot:



Windows PowerShell Web Access

Enter your credentials and connection settings

User name: administrator
Password:
Connection type: Computer Name
Computer name:
Optional Connection Settings

© 2011 Microsoft Corporation. All rights reserved.

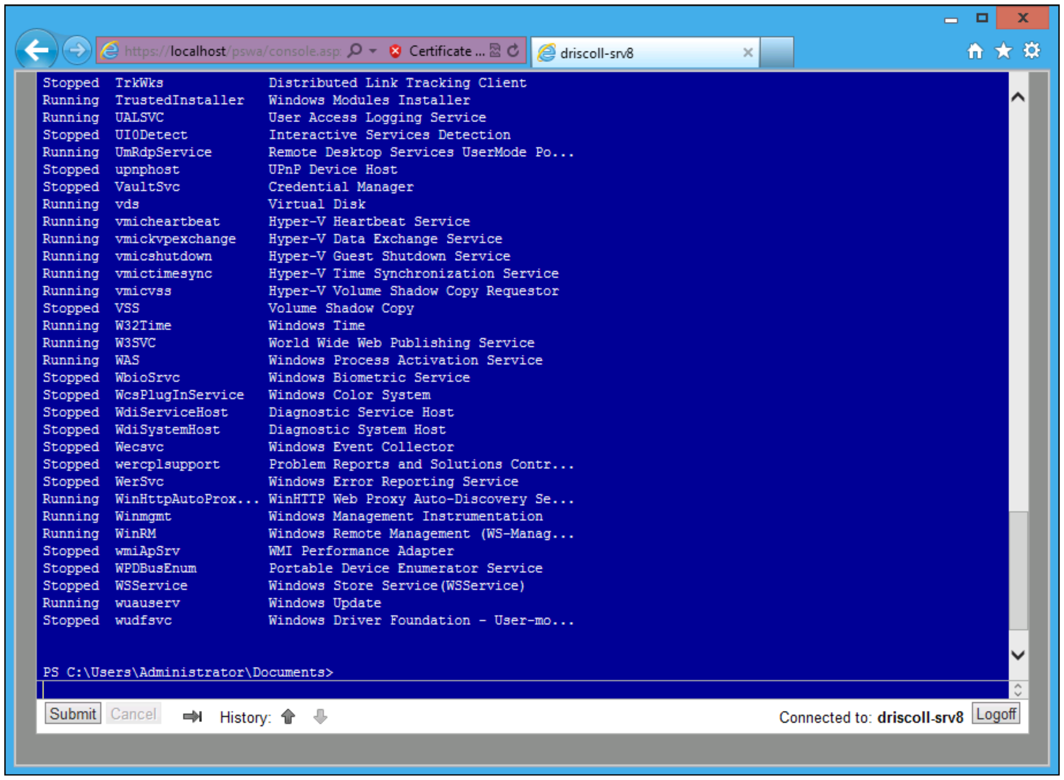
Internet Explorer will complain that there is a certificate error. This is because we used a self-signed certificate. To fix this we have to add the certificate to the personal certificate store.

The web access can be used to connect to any computer which has remoting enabled. It is necessary to specify the computer name when logging in. In addition to specifying a computer name there are a whole host of additional options that can be specified. These options are found in the **Optional Connection Settings** section. Some of these options include the session configuration name, the authentication type, whether to use SSL and the ability to specify alternate credentials. Any PowerShell-enabled computer with remoting enabled can benefit from the web access. The ability to specify a configuration name allows us to control the experience the user will have, when connecting to the remote machine.

It is even possible to connect to machines running PowerShell v2.0. This is very beneficial for organizations that may not be able to adopt Windows 8 or PowerShell 3.0 as quickly as others. It is possible to stand up a single PowerShell Web Access server to service the needs of a whole collection of computers within an organization.

Working with the PowerShell Web Access console

Once we are logged into the web access console we will see a familiar blue console. The console looks and behaves very much like the standard Windows PowerShell console. Refer to the following screenshot:

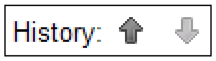

https://localhost/pswa/console.asp
Certificate ...
driscoll-srv8
Submit
Cancel
History:
Connected to: driscoll-srv8
Logoff

```
Stopped  TrkWks              Distributed Link Tracking Client
Running  TrustedInstaller    Windows Modules Installer
Running  UALSVC              User Access Logging Service
Stopped  UI0Detect           Interactive Services Detection
Running  UmRdpService        Remote Desktop Services UserMode Po...
Stopped  upnphost            UPnP Device Host
Stopped  VaultSvc            Credential Manager
Running  vds                 Virtual Disk
Running  vmicheartbeat       Hyper-V Heartbeat Service
Running  vmickvpexchange     Hyper-V Data Exchange Service
Running  vmicshutdown        Hyper-V Guest Shutdown Service
Running  vmictimesync        Hyper-V Time Synchronization Service
Running  vmicvss             Hyper-V Volume Shadow Copy Requestor
Stopped  VSS                 Volume Shadow Copy
Running  W32Time             Windows Time
Running  W3SVC               World Wide Web Publishing Service
Running  WAS                 Windows Process Activation Service
Stopped  WbioSrvc            Windows Biometric Service
Stopped  WcsPlugInService    Windows Color System
Stopped  WdiServiceHost      Diagnostic Service Host
Stopped  WdiSystemHost       Diagnostic System Host
Stopped  Wecsvc              Windows Event Collector
Stopped  wercplsupport       Problem Reports and Solutions Contr...
Stopped  WerSvc              Windows Error Reporting Service
Running  WinHttpAutoProx...  WinHTTP Web Proxy Auto-Discovery Se...
Running  Winmgmt             Windows Management Instrumentation
Running  WinRM               Windows Remote Management (WS-Manag...
Stopped  wmiApSrv            WMI Performance Adapter
Stopped  WPDBusEnum          Portable Device Enumerator Service
Stopped  WSService           Windows Store Service (WSService)
Running  wuauserv            Windows Update
Stopped  wudfsvc             Windows Driver Foundation - User-mo...


PS C:\Users\Administrator\Documents>
```

Tab completion can be accomplished with either a press of the *Tab* button or a click on the following icon:

Midline tabbing does not work in the web access console. Instead of expanding it simply does nothing. It will not delete the rest of the line.

Pressing the *Esc*, *Tab*, up and down arrow keys on a computer with a regular keyboard will work, but this probably will not work on a mobile device. The history buttons, as shown in the following image, provide a touch-sensitive way of doing this.



There is also a **Logoff** button in the lower right-hand corner of the web page. This will terminate the remote session and bring us back to the login panel where we first started.

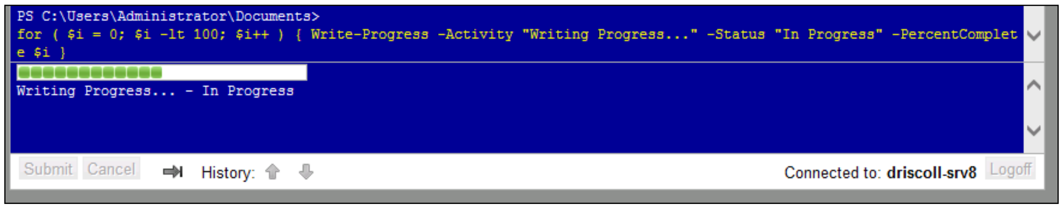
With a quick check of the host, we can see that the host is of the type `ServerRemoteHost`. This can be done using the following commands:

```
PS C:\> $Host

Name             : ServerRemoteHost
Version          : 1.0.0.0
InstanceId       : 93ba397b-114d-4cc4-ba09-54b01538a5ad
UI               : System.Management.Automation.Internal.Host.
InternalHostUserInterface
CurrentCulture   : en-US
CurrentUICulture : en-US
PrivateData      :
IsRunspacePushed :
Runspace         :
```

Additional input and output

The web access console supports all the standard input and output cmdlets that are available in the usual PowerShell host. These include `Write-Host`, `Write-Progress`, `Write-Error`, and so on. `Write-Host` even exposes the ability to change the background and foreground colors just like it does in the desktop PowerShell consoles.

A custom progress dialog is also integrated into the console which presents a nice indication of long running processes:



```
PS C:\Users\Administrator\Documents>
for ( $i = 0; $i -lt 100; $i++ ) { Write-Progress -Activity "Writing Progress..." -Status "In Progress" -PercentComplet
e $i }
Writing Progress... - In Progress
```

The web access even allows for entering credentials in a secure fashion. Rather than typing, plain text characters are masked by a dot.

It is recommended to run the PowerShell Web Access over HTTPS, because if someone or something was listening to the traffic between the page and the local machine, our password would be clear text.

Multiline statements

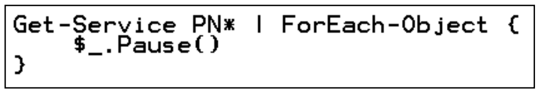
There is a difference between the regular PowerShell console and the web access console when it comes to multiline statements. When working with the desktop console, multiline statements are achieved by opening a statement block and pressing *Enter*. The console recognizes that the statement is intended to continue. Once we close the statement block and press *Enter* twice, the entire command will be executed. If, for instance, we were trying to loop through a list of services and pause them, we could do the following:

```
PS C:\> Get-Service PN* | ForEach-Object {
>> $_.Pause()
>> }
```

Notice that the PowerShell console allows us to continue the line, after we have opened the statement using the open curly brace ({). If we attempt to do the same thing in the web access console we will find that an error is shown as follows:

```
PS C:\> Get-Service PN* | ForEach-Object {
Missing closing '}' in statement block.
```

This is because the web access console treats multiline statements differently. Rather than asking for additional input after pressing *Enter*, the input box is simply multiline enabled. This allows us to enter multiple lines by pressing *Shift* + *Enter* to move to the next line. This means we can simply type the entire command right into the input box. It is also possible to copy and paste multiline commands into the textbox. The following screenshot shows an example of a multiline script that pauses services with names starting with `PN`:



```
Get-Service PN* | ForEach-Object {
    $_.Pause()
}
```

Enhancements to the PowerShell ISE

The PowerShell ISE has been in PowerShell since the second version. In the previous version it provided syntax highlighting and debugging capabilities. The ISE was never quite as full of features as some of the third party scripting applications available. In the newest version of PowerShell, Microsoft has included an ISE which is much more feature rich. In this section we will go over the enhancements to the ISE that make it a much more viable tool for working with PowerShell scripts.

Intellisense

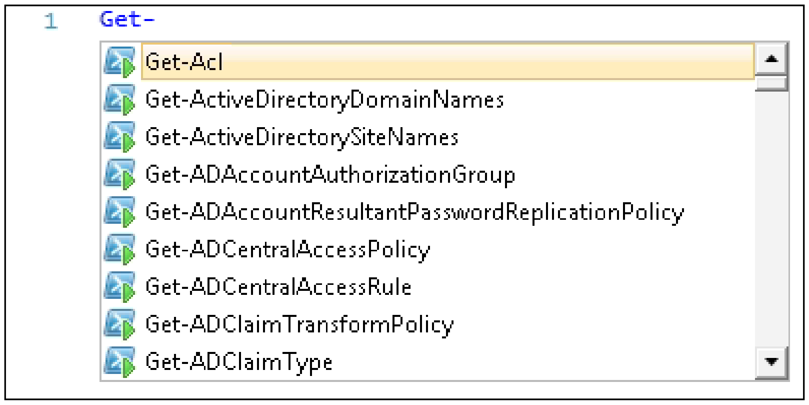
In the previous version of the PowerShell ISE the script editor supported tab completion. When we started to type a command, we could have the ISE behave just as it would within the PowerShell command prompt. Each subsequent tab would cycle through the available options and we could eventually narrow down what we were looking for. Although this is convenient, it is often nice to have a more rich experience when working with a more full featured user interface. In Visual Studio for example, when different portions of code such as variables are typed, the development environment will display a drop-down list of available options to the user. This allows us to see more than just a single item at a time.

The PowerShell 3.0 ISE now offers this type of support. If we were to start typing a cmdlet name, the ISE will drop down a list of available cmdlets that match the pattern that we have already typed. For example, if we started typing the following:

```
Invoke-
```

Once the hyphen was inserted, the ISE would then drop down a list of available `Invoke` cmdlets or functions that match the verb. They are in alphabetic order. If we continue to type, the **Intellisense** will narrow down the list even further, until we find what we are looking for. In addition to showing the names of the cmdlets or functions, the ISE will also give a tool-tip display of all of the parameters available for the cmdlet, which we are currently focused on. This is a great way to quickly discover cmdlets and their usages without having to read through the help.

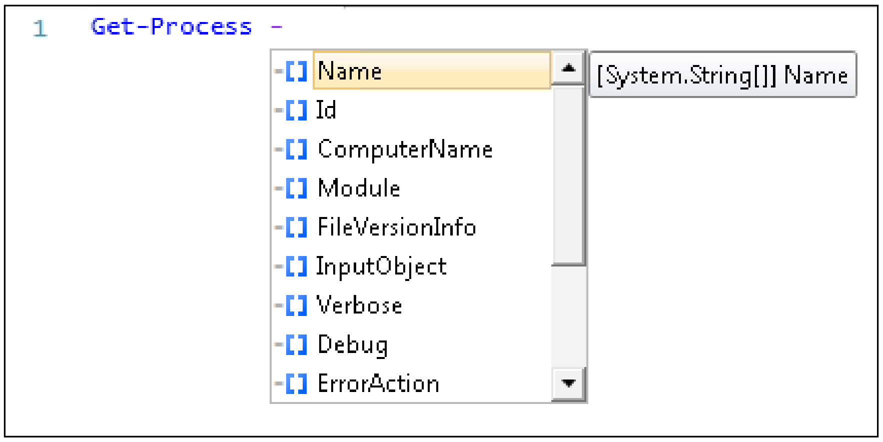
Intellisense is always context sensitive. Here are some examples of when Intellisense will be invoked. When typing the names of cmdlets or functions by typing a hyphen, it is invoked as follows:



```
Get-
```

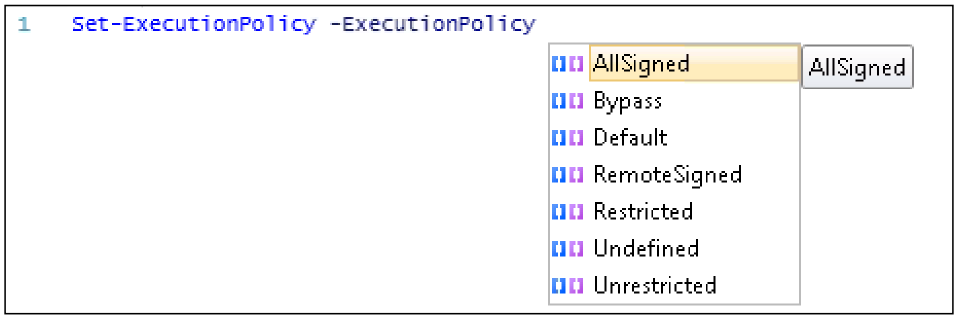
Get-Acl
Get-ActiveDirectoryDomainNames
Get-ActiveDirectorySiteNames
Get-ADAccountAuthorizationGroup
Get-ADAccountResultantPasswordReplicationPolicy
Get-ADCentralAccessPolicy
Get-ADCentralAccessRule
Get-ADClaimTransformPolicy
Get-ADClaimType

When adding parameters to a cmdlet by typing a hyphen, it is invoked as shown in the following screenshot:



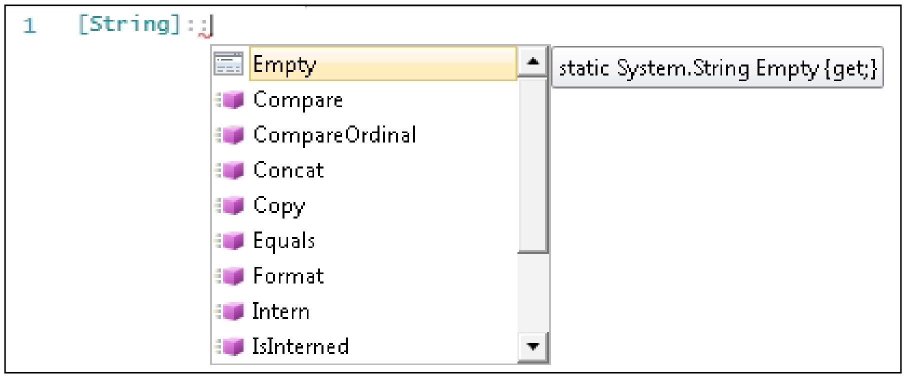
```
Get-Process -
```

When accessing parameters that accept enumeration or validation sets, it is invoked as follows:



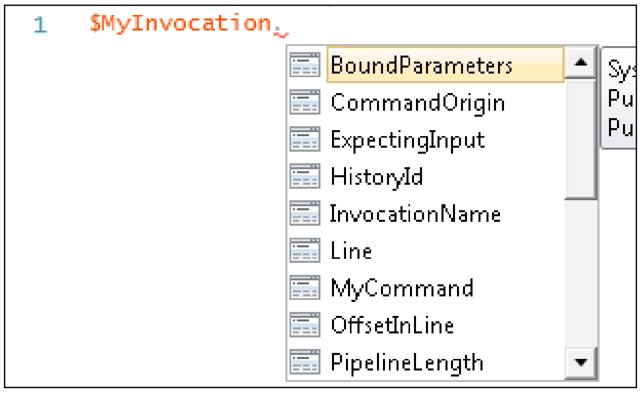
```
Set-ExecutionPolicy -ExecutionPolicy
```

When accessing the properties and methods of a type by typing the double colons, we are using the string type to invoke Intellisense as follows:



```
[String]::
```

When accessing the properties and methods of an object by typing a period it invokes, as shown in following screenshot:



```
$MyInvocation.
```

Arrow keys can be used to cycle through the available options. At any time we can press *Enter* or *Tab* to select the option that is currently selected. To force Intelllisense to open we can use the *Ctrl* + *Space bar* key combination.

There are several options that can change the behavior of Intellisense. One option is to disable the *Enter* key from selecting the currently suggested option. This can be a problem for some, as they wish to move to the next line but instead select an option presented by Intellisense. This option is broken down into two separate properties, one for the script pane and one for the command pane as follows:

```
$psise.Options.UseEnterToSelectInConsolePaneIntellisense
$psise.Options.UseEnterToSelectInScriptPaneIntellisense
```

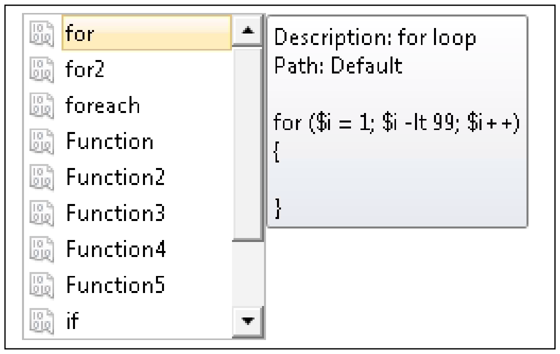
Intellisense can be turned off all together with the following options:

```
$psise.Options.ShowIntellisenseInConsolePane
$psise.Options.ShowIntellisenseInScriptPane
```

Snippets

Snippets are short scripts that can be re-used and quickly inserted into the scripts that we are working on. Sometimes it is nice to be able to quickly insert all the code necessary for a `for` loop or function. Rather than having to type all the code necessary, we can just induce the snippet selection menu and insert the script snippet. Previously the ISE did not have this feature while other third party script editors did. The new version of the ISE can now consume special snippet files and expose them through a key stroke and drop-down menu. Some of the built-in snippets include loops, functions, and try/catch blocks.

Snippets can both be defined by users or even included in modules. In order to show the snippet menu in the editor, we use the *Ctrl* + *J* short cut. This will display the following drop-down menu at the line where the cursor is. If we were to select one of the items from the list, the block would be inserted into that position.



Description: for loop
Path: Default

```
for ($i = 1; $i -lt 99; $i++)
{

}
```

Snippets define where the caret should be placed after inserting the snippet. This allows us to just continue on typing rather than having to situate the caret in the correct place within the script. For example, if we inserted the function snippet, the caret is placed right after the function keyword so we have an opportunity to enter the function name. The following commands explain this:

```
function <caret is here>
{
}
```

Defining our own snippets

The ISE exposes a couple of simple cmdlets that can be used to create snippets and retrieve snippets. Snippets are defined as `snippet.ps1xml` files and stored within the user's home directory. To access this directory from a PowerShell script or command line, we can use the following command:

```
PS C:\> $ENV:USERPROFILE\Documents\WindowsPowerShell\Snippets
```

Once a snippet has been defined the ISE will automatically load the snippet when it starts up. To define a new snippet, we can simply use the `New-IseSnippet` cmdlet as follows:

```
New-IseSnippet -Title "Set foreground color" -Description "This snippet
sets the foreground color of the PowerShell host" -Text "`$host.
UI.ForeGroundColor = [Color]::" -CaretOffset 36
```

This snippet would insert the text that is provided by the `Text` parameter and then move the user's caret into the 36 character position. This would place the cursor right after the double colons that specify the color. We could then just type in the name of the color and could then change the host's foreground color in our script. Executing the `New-IseSnippet` cmdlet would also store the snippet into the path defined by the following command:

```
Join-Path (Split-Path $profile.CurrentUserCurrentHost) "Snippets"
```

To get a list of the defined snippets we can also use the `Get-IseSnippet` cmdlet. It will return a list of the available user-defined snippets. This is done by using the following commands:

```
PS C:\Users\Adam> Get-IseSnippet


    Directory:
C:\Users\Adam\Documents\WindowsPowerShell\Snippets
```

```
Mode      LastWriteTime      Length Name

----      -------------      ------ ----

-a---     2/12/2012 8:39 AM 754 Set foreground color.snippets.ps1xml
```

Notice that it displays the path to where the snippet is stored and the file name of the snippet. The object type returned by the `Get-IseSnippet` cmdlet is a `FileInfo` object. Because of this we can use the file system provider to remove the snippet using `Remove-Item`. `Get-IseSnippet` does not offer a filtering mechanism so we will need to use `Where-Object` to get the correct snippet file as follows:

```
Get-IseSnippet | Where-Object Name -like "Set foreground color*" |
Remove-Item
```

The snippet will not be automatically removed from the snippet drop-down list. It is required to restart the ISE after removing the snippet file. It is also possible to store snippets in locations other than the home directory snippets folder. In order to load these snippets we need to call a special method found under the `$PSISE` variable as follows:

```
$psISE.CurrentPowerShellTab.Snippets.LoadFromFolder($MyFolderPath, $true)
```

The first parameter is the path to search for additional snippets. The second parameter is whether to recursively look for snippet files. The `LoadFromFolder` method finds any files matching the pattern `*.snippets.ps1xml`.

It is also now possible to ship modules with snippets included. Within the module, simply include a snippets folder underneath the module folder which contains all the snippet files. Once the module has been loaded into the ISE using `Import-Module`, we can use another method found on the `$PSISE` variable to load those snippets as follows:

```
$psISE.CurrentPowerShellTab.Snippets.LoadFromImportedModules()
```

Once we have executed the previous line all the snippets found in any of the loaded modules will be loaded into the ISE. As we can see the extensive use of snippets will help to alleviate some of the typing necessary for large blocks of PowerShell script. Being able to ship snippets in modules makes packaging functionality into a module even more beneficial.

XML file syntax highlighting

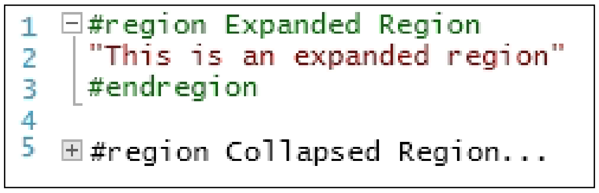
Although most PowerShell components are defined as either PowerShell scripts or binary modules, there are several pieces of the system that utilize XML. For example, the PS1XML file extension is used for defining type formatters and the XML file extension is used to define contextual help. PowerShell ships with a lot of these files to make working with certain types of objects much easier on the command line. Third party modules also have the option to ship with these types of modules if they wish. The new version of the ISE now supports simple XML syntax highlighting and code folding.

It also does some error checking on the fly so squiggly lines are placed under elements of sections that do not match the syntactical or structural guidelines for XML. The editor will work in XML-mode when a PS1XML or XML file is opened. There is no Intellisense support for XML documents.

Other editor enhancements

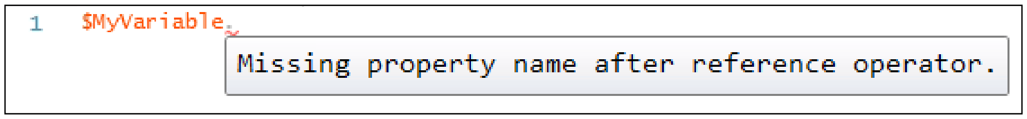
In addition to Intellisense and snippets there have been a whole host of other enhancements to make working in the ISE better. One enhancement has to do with braces. When braces are opened, it can sometimes be hard to find which closing brace matches the open brace. In order to make this easier to locate in a glace, there is now **brace matching** available. When positioning the caret after the opening brace, both matching opening and closing braces will be highlighted to show the match. In addition to the visual cue, it is also possible to jump to a matching brace by positioning the cursor before an opening brace, or after a closing brace and selecting **Edit\Go To Match** or by pressing *Ctrl +]*.

Not only can we match the braces but also fold the code within the braces. **Code folding** is the ability to hide entire blocks of code within a section. In this case, on the left-hand side of an open brace there will now be a minus icon. Clicking on this icon will fold or hide the code within the matching braces. This can help with large scripts that have many functions or nested code blocks. In addition to folding matched braces, we can also fold commented sections based on the new `region` code word. We can start a region with a comment that starts with the region. To signify the end of a region, we insert another comment with the word text `endregion`. The ISE will then allow us to fold the code between the two regions as shown in the following screenshot:



```
#region Expanded Region
"This is an expanded region"
#endregion

#region Collapsed Region...
```

Errors are now highlighted in the script prior to running it. The ISE parses the script and will signify errors, by drawing a squiggly line underneath the problematic code. Moving the mouse cursor over the squiggly line will cause the ISE to display a tool-tip which describes why it has highlighted to the code. This provides us with quicker feedback than we would typically get by re-running the script over and over again as follows:


```
$MyVariable.
```
Missing property name after reference operator.

Another usability improvement is the ability to block copy and replace text. With either *Alt* + Mouse or *Alt* + *Shift* + arrow keys we can select a block of text rather than lines of text. It behaves the same way the `Mark` option does within common Windows command prompts. In addition to selecting text though, we can replace text in the block. This is referred to as **filling** as follows:

```
Stop-Process PowerShell
Stop-Process Notepad
Stop-Service TermService
```

After block selecting the text like this, if we typed `Start` the script would result in the following text:

```
Start-Process PowerShell
Start-Process Notepad
Start-Service TermService
```

Often times we may need to copy text from the ISE and place it into another document. If it's a blog post or something similar, it may be nice to maintain the formatting and the syntax highlighting. This was not the case in the previous version of the ISE. In the new version, both the command pane and script pane will provide formatting data into the text copied from them. When pasted into a rich text editor like Microsoft Word, the syntax highlighting will be included as well.

Another hidden gem in this version of the ISE is the **auto-save** feature. In the previous version of the ISE, if the editor was shut down unexpectedly, any unsaved work would have been lost. In the new version of the ISE this is alleviated by an auto-save feature. There is a configurable interval that can be set which tells the ISE how often to auto-save documents. If the ISE is then closed forcibly, the next time the ISE opens it will attempt to recover the documents that were not saved during the last session.

Another subtle but beneficial enhancement is the **zoom** level. In the previous version of the ISE the text was zoomed based on a font size setting rather than a percentage. This could cause problems on machines that had different DPI settings and made it harder to work with add-ons. In the new ISE the zoom factor is based on a percentage.

ISE options

The ISE has a whole host of options for adjusting how the tool behaves. These options affect Intellisense, snippets, add-ons, and other properties and behaviors of the editor. This section will outline some of the important options, their default values, and accepted ranges.

Options are accessible through the `$PSISE` variable that is defined in the ISE host. Using the new Intellisense, typing the following will display a full list of options that the ISE exposes:

```
#Use for Intellisense
$PSISE.Options.

#Use for a full list in the output pane
$PSISE.Options | Get-Member
```

At any point that we want to reset our options back to default values we can use the `RestoreDefaults` method on the `Option` property as follows:

```
$PSISE.Options.RestoreDefaults()
```

List of ISE options

Although the list is not comprehensive, it contains many of the options that pertain to functionality that we have looked at elsewhere in this chapter. All of the option names are properties of the `ISEOptions` type. An instance of this type is available through the `$psISE.Options` property. Changing any of the properties will have an instant effect on the ISE. For example, to access the `FontSize` property we could do the following:

```
$PSISE.Options.FontSize = 20
```

Option Name	**Description**	**Default Value**
`AutoSaveMinuteInterval`	This is the interval at which documents within the ISE will be auto-saved.	2
`MruCount`	It configures how many most recently used files are displayed in the **File** menu.	10
`ShowDefaultSnippets`	Whether or not to display default snippets. If false, only user defined snippets will be shown.	True
`ShowIntellisenseInConsolePane`	Whether or not to display **Intellisense** in the command pane.	True
`ShowIntellisenseInScriptPane`	Whether or not to display **Intellisense** in the script pane.	True
`ShowLineNumbers`	Whether to show line numbers	True
`ShowOutlining`	Whether to show bracket outlining. When outlining is off, code folding is also disabled.	True

Option Name	Description	Default Value
`ShowWarningBeforeSavingOnRun`	Whether to show a warning that a file must be saved before running it.	True
`UseEnterToSelectInConsolePaneIntellisense`	Whether enter selects the current item in the command pane **Intellisense**. If this option is false, tab will still select the item.	True
`UseEnterToSelectInScriptPaneIntellisense`	Whether enter selects the current item in the script pane **Intellisense**. If this option is false, tab will still select the item.	True
`UseLocalHelp`	Whether the ISE help dialog is displayed or the online version is displayed. This does not affect PowerShell contextual help.	True

Colors

In addition to the options defined in this table, there are numerous color settings available through the `Options` property. Some of the simple colors include error, verbose, or debug colors, or even output and script pane colors. These colors can be set by setting the properties of the `ISEOptions` as follows:

```
$PSISE.Options.ErrorBackgroundColor = [System.Windows.Media.
Colors]::Orange
```

We could also use the string `Orange` rather than typing the full type name, as done in this example. Colors can also be created using RGB values using the `System.Windows.Media.Color` class.

In addition to the basic colors that we can set, there are a wide range of finely grained syntactical color settings that we can adjust. To access these color groups we can use the `TokenColors` or `XmlColors` properties of the `ISEOptions` class. Both of these items are hash tables of colors. The key is the name of the token or element, that the syntax highlighting affects and the value is the color. Some of the token colors include `Command`, `CommandParameter`, `Keyword`, `Number`, and `Operator`. Some of the XML token colors include `ElementName`, `Text`, and `Attribute`.

To adjust any of these colors we would use the following syntax. Note that the setting will take affect instantly.

```
$PSISE.Options.TokenColors["Comment"] = "Red"
```

Note that in this example, we used a string for the color name. PowerShell automatically converted Red into the object representation of the color.

ISE add-ons

The ISE has always been extensible through the `$PSISE` variable. This has allowed the PowerShell community to create a large set of add-ons that enhance the ISE beyond what was originally released. The properties exposed by the `$PSISE` available allow us to access the tabs, documents, and even text. Almost anything within the ISE can be modified. The PowerShell 3.0 version of the ISE comes with a built-in add-on which is enabled by default. When we open up the environment we have an extra tool window on the right-hand side of the editor called the **Command Window**. It has a list of the currently defined functions and cmdlets. Double-clicking on any one of the items will execute the command in the ISE.

The add-on menu is now shown by default and directs us to the Internet repository of downloadable add-ons. Creating add-ons is nothing more than writing scripts that interact with the ISE and expose new functionality. For example, if we wanted to add a new menu item to the **Add-ons** menu, we could do something like the following:

```
$psISE.CurrentPowerShellTab.AddonsMenu.SubMenus.Add("Say Hello", { Write-
Host "Hello!" },$null)
```

After executing this method we will now have a new menu item in the **Add-ons** menu with the display text of Say Hello. Clicking on the menu item will execute `Write-Host` and print the text into the output pane. Executing a command like this will not cause the ISE to recall the menu the next time it is started. In order to persist ISE add-ons, it is general practice to package the scripts into a module. As modules can define which hosts that they support, we can create a module that works only in the ISE by specifying the host name Windows PowerShell ISE Host. There are all kinds of things that the PSISE variable exposes to us. We can go as far as to modify the text within the tabbed documents by using the `Editor` property.

The latest version of the ISE has reached the level of some of the third party script editors that are available. The inclusion of Intellisense, snippets, and basic text editing features, such as block copy and code folding, make it much easier to work with than the previous version. Thanks to the easy to use extensibility and customizability of the ISE, it ensures a much better user experience to a wide range of PowerShell users.

Summary

As we saw in this chapter the ways to access and run PowerShell have extended beyond a desktop computer. The PowerShell Web Access now allows us to access our management from any device that supports a web browser. We also saw how the changes to the PowerShell ISE have made it a much more viable alternative to the third party script editors. Features such as snippets and Intellisense make it a much more valuable tool for working with scripts. Additionally, the ability to extend and modify the ISE makes it an even more valuable tool as the community begins to develop add-ons.

In the final chapter, we will take a look at a some of the new cmdlets and modules found within the Windows Server 2012 and Windows 8.

7
Windows 8 and Windows Server 2012 Modules and Cmdlets

The Windows 7 and Windows Server 2008 R2 were the first operating systems to have PowerShell packaged directly within the operating system. The language was even enabled by default. Out of the box PowerShell offered around 250 cmdlets and just a few modules. The list has grown dramatically in the next version of the client and server of the Microsoft operating system. Windows Server 2012 has over 2300 cmdlets and around 60 modules that come with the system. This wide coverage is in part, due to the native CIM cmdlets that make it easier for Windows feature teams to produce cmdlets, and in part due to the dedication of Microsoft to provide a PowerShell interface for all their products.

This chapter will be dedicated to looking at a select list of the modules and cmdlets in detail. Each section will be broken down by module and then will delve into some of the cmdlets within that module. The end of the chapter will have a description of some of the modules we didn't investigate in detail.

In this chapter we will cover the following topics:

- A selection of new cmdlets found in the core PowerShell modules
- A selection of new modules and cmdlets found in Windows 8 and Windows Server 2012

Core modules

These modules are installed by default in both Windows 8 and Windows Server 2012. These cmdlets will be available on machines that are running the latest version of the Windows Management Framework as well.

Invoke-WebRequest

In the previous version of PowerShell, a .NET class was required to access web-based content. In PowerShell 3.0, the same can be accomplished by utilizing the `Invoke-WebRequest` cmdlet. The cmdlet returns an object `HttpWebResponseObject` that contains all kinds of information about the HTTP request such as the response code, the response content, and even parsed elements within the page, such as links and forms.

Usage examples

The following example retrieves a pages links, selects the first three, and outputs their `innerHTML` and destination:

```
PS C:\> Invoke-WebRequest www.microsoft.com | Select-Object
-ExpandProperty links | Select-Object -First 3 –Property innerHtml,href


innerHTML                href
---------                ----
<DIV id=ctl00_...        /en-us/default.aspx?bldi=0-0
<DIV class=hp_...        http://windows.microsoft...
Windows                  http://windows.microsoft...
```

The following example retrieves the first image found on `www.microsoft.com` and displays it in the default web browser:

```
PS C:\ > Start-Process (invoke-webrequest -Uri http://www.microsoft.
com | Select-Object -ExpandProperty images | Select-Object -First 1
-ExpandProperty Src)
```

Invoke-RestMethod

This cmdlet is used to invoke **Representational State Transfer** (**REST**) methods.

> REST is a client/server architecture based on resource types and state. **Universal Resource Identifiers** (**URI**), are used to access different resources within a RESTful service. Typically, RESTful services utilize the existing HTTP protocol to communicate between the client and server.

Usage examples

The following example invokes a REST method using the HTTP `GET` method to retrieve a particular resource on the `mydomain.com` server. It is expected that the result will be returned as JSON:

```
Invoke-RestMethod –Uri http://www.mydomain.com/group/1/user/123 -Method
GET -ReturnType Json
```

ConvertTo-Json

This cmdlet is used to convert objects in PowerShell into **JavaScript Object Notation** (**JSON**). JSON is used to serialize object structures to a string so that it can be sent in between a client and server. JSON is frequently used on wide area networks because its syntax is much more succinct than schemes such as XML.

Usage examples

The following example converts an object to JSON:

```
PS C:\> Get-Variable PSUICulture | ConvertTo-Json
{
 "Value":  "en-US",
 "Name":  "PSUICulture",
 "Description":  "UI Culture of the current Windows PowerShell Session.",
 "Visibility":  0,
 "Module":  null,
 "ModuleName":  "",
 "Options":  9,
 "Attributes":  [

 ]
}
```

The following example converts an object to JSON that has members added at runtime:

```
PS C:\> Get-Random | Add-Member -MemberType NoteProperty -Value "MyValue"
-Name "MyProperty" -PassThru  | ConvertTo-Json
{
 "value":  1705462562,
 "MyProperty":  "MyValue"
}
```

The following example shows that the `ConvertTo-Json` treat data types like strings and integers differently than the objects shown previously. Notice that there is no length property serialized in this example, even though the string class exposes it.

```
PS C:\> "MyString" | ConvertTo-Json
"MyString"


PS C:\> Get-Random | ConvertTo-Json
1231
```

ConvertFrom-Json

This cmldet is used to convert a JSON string into an object that can be used in PowerShell. Unlike the `ConvertTo-CliXml`, the `ConvertFrom-Json` cmdlet does not preserve type data and will simply be of the type `PSCustomObject`. Particular data types can be converted into their object representation if put in the correct format.

Usage examples

The following example shows that the JSON string in the example is converted into a `PSCustomObject` with a property `DateTime` that is a `String` type:

```
PS C:\ > '{ "DateTime":  "Thursday, February 23, 2012 7:56:18 PM" }' |
ConvertFrom-Json | Get-Member -Name Date
Time


 TypeName: System.Management.Automation.PSCustomObject


Name     MemberType   Definition
----     ----------   ----------
DateTime NoteProperty System.String DateTime=Thursday, February 23, 2012
7:56:18 PM
```

The following example shows that particular data type formats, in this case a date, are passed to the `ConvertFrom-Json` cmdlet, it will be able to convert it to a .NET class:

```
PS C:\ > '{  "DateTime":  "\/Date(1330048578834)\/" }' | ConvertFrom-Json
| Get-Member -Name DateTime


 TypeName: System.Management.Automation.PSCustomObject


Name     MemberType   Definition
----     ----------   ----------
DateTime NoteProperty System.DateTime DateTime=2/24/2012 1:56:18 AM
```

ControlPanelItem

The `Get-ControlPanelItem` and `Show-ControlPanelItem` cmdlets are used to list and show control panel items that are currently installed on the system. Using the `Show-ControlPanelItem` will cause the control panel window to be shown immediately.

Usage examples

The following example gets the `Mouse` control panel item and shows the `Name`, `Category` and `Description` properties:

```
PS C:\> Get-ControlPanelItem -Name Mouse | Select-Object
Name,Category,Description


Name       Category                  Description
----       --------                  -----------
Mouse      {All Control Panel Items} Customize your...
```

The following example shows the `Fonts` control panel item using the canonical name:

```
PS C:\> Show-ControlPanelItem -CanonicalName Microsoft.Fonts
```

Rename-Computer

This cmdlet is used to rename local or remote computers. The cmdlet can use both local or domain credentials and can trigger a restart.

Usage examples

The following example renames the computer `driscoll-hv1` to `driscoll-hv2` using the domain credentials. The command also bypasses confirmation and causes a restart of the machine:

```
PS C:\> Rename-Computer -ComputerName driscoll-hv1 -NewName driscoll-hv2
-DomainCredential mdn\administrator -Force -Restart
```

TypeData

The `TypeData` cmdlets are used for managing the **Extended Type System** (**ETS**), type data that has been loaded into PowerShell. This type data is added through `*.types.ps1xml` files and are typically found in modules.

Usage examples

The following example returns the ETS type data for the `System.Management.ManagementObject` class and displays the added ETS members:

```
PS C:\ > Get-TypeData System.Management.ManagementObject | Select-Object
-ExpandProperty Members


Key                 Value
---                 -----
ConvertToDateTime   ...ScriptMethodData
ConvertFromDateTime ...ScriptMethodData
```

The following example removes the ETS type data for the `System.Guid` class:

```
PS C:\ > [System.Guid]::NewGuid() | Get-Member -Name Guid


 TypeName: System.Guid


Name        MemberType Definition
----        ---------- ----------
Guid        ScriptProperty System.Object Guid {get=$this.ToString();}


PS C:\ > Remove-TypeData -TypeName System.Guid
PS C:\ > [System.Guid]::NewGuid() | Get-Member -Name Guid
PS C:\ >
```

Unblock-File

This cmdlet is used to remove the alternate data stream from the file, known as the **zone identifier**. The zone identifier signifies that the file has come from the Internet and PowerShell will warn or prevent you from executing scripts with this zone identifier. This can become very hard to deal with when a lot of files are involved, such as with a module. The `Unblock-File` cmdlet makes this much easier.

Usage examples

The following example removes the zone identifier from the `Test.ps1` file:

```
PS C:\> Unblock-File –Path C:\Users\Adam\Documents\Test.ps1
```

The following example unblocks all the files in the `HyperV module` directory:

```
PS C:\> Get-ChildItem -Path C:\Users\Adam\Documents\WindowsPowerShell\
Modules\HyperV | Unblock-File
```

Standard modules

These modules are part of the standard set of modules available on Windows 8 and Windows Server 2012 machines. They are not available to users who install the Windows Management Framework on machines such as Windows 7 or Windows Server 2008 R2. No features or roles are required to access any of these modules and most of them are based on CIM providers so they support CIM sessions.

NetAdapter module

This module is available for the users running Windows 8 and Windows Server 2012 and it requires no feature. It contains a total of 64 cmdlets. The following command shows how to import this module:

```
PS C:\ > Import-Module NetAdapter
```

The `NetAdapter` module exposes cmdlets that manage the network adapters, protocol bindings, and numerous other network related operations. The module is enabled by default and there are no features that need to be installed to enable it. The cmdlets in this module are based on CIM and support CIM sessions.

NetAdapter cmdlets

These cmdlets allow us to manage the network adapters that are currently installed on the system. The `Get-NetAdapter` allows us to retrieve them. The `Set-NetAdapter` allows us to change the MAC and VLAN ID of the adapters. The `Enable-NetAdapater`, `Disable-NetAdapter`, and `Restart-NetAdapter` allow us to change the state of the adapter.

Usage examples

The following example returns all the network adapters on the local machine:

```
PS C:\> Get-NetAdapter


Name     Interface MacAddress Operational LinkSpeed
                        Index      Status
----     --------- ---------- ----------- ---------
Wired... 12 00-15-5D-91-3A-0C          Up   10 Gbps
```

The following example resets the network adapters that match the name Wired*:

```
PS C:\> Get-NetAdapter Wired* | Restart-NetAdapter
```

The following example sets the VLAN ID of all the network adapters that match the name Wired*:

```
PS C:\> Get-NetAdapter Wired* | Set-NetAdapter –VLANID 3
```

NetAdapterBinding

Theses cmdlets let us manage the network bindings for the various interfaces defined in the local or remote system. In addition to seeing the current defined bindings, such as TCP/IP v4, we can enable and disable the bindings for particular interfaces.

Usage examples

The following example returns the network bindings that match the display name *TCP/IPv4*:

```
PS C:\ > Get-NetAdapterBinding -DisplayName *TCP/IPv4*


Name    DisplayName    ComponentID         Enabled
----    -----------    -----------         -------
Wired...Internet...    ms_tcpip            True
```

The following example disables all network adapter bindings that match the display name *TCP/IPv6*:

```
PS C:\ > Get-NetAdapterBinding -DisplayName *TCP/IPv6* | Disable-
NetAdapterBinding
```

NetAdapaterAdvancedProperty

These cmdlets are used for managing some of the advanced properties for network adapters. Some of these properties include receive buffer size and TCP checksum offload for IPv4 and IPv6. We can retrieve the properties using the `Get-NetAdapterAdvancedProperty` cmdlet and set them using the `Set-NetAdapterAdvancedProperty` cmdlet. We can even create new properties using the `New-NetAdapterAdvancedProperty` cmdlet.

Usage examples

The following example gets the `Receive Buffers` advanced `NetAdapter` property:

```
PS C:\> Get-NetAdapterAdvancedProperty –DisplayName "Receive Buffers"|
Select-Object -ExpandProperty RegistryValue
512
```

The following example enables the `Jumbo Packet` advanced `NetAdapter` property:

```
PS C:\> Set-NetAdapterAdvancedProperty –Name "Wired Ethernet Connection"
–DisplayName "Jumbo Packet" –Value "Enabled"
```

SmbShare module

This module is available in Windows 8 and Windows Server 2012 and it does not require any feature as well. There are a total of 23 cmdlets in this module. The following command is used to import it:

```
PS C:\> Import-Module SmbShare
```

The `SmbShare` module is used to manage **Server Message Block** (**SMB**) shares, connections, and other share related information on a Windows machine. The cmdlets found within the module can create new shares, configure permissions, and enumerate connections to the shares. The cmdlet can also control SMB client configuration settings and network interfaces used for the SMB connections. The cmdlets within this module are native CIM cmdlets and support CIM sessions.

SmbShare

The `SmbShare` cmdlets allow for enumeration, creation, removal, and access control of the SMB shares on the current server.

Usage examples

The following example selects the first SMB share:

```
PS C:\ > Get-SmbShare | Select-Object -First 1


Name          ScopeName       Path         Description
----          ---------       ----         -----------
ADMIN$        *               C:\Windows   Remote Admin
```

The following example creates a new SMB share with the name Share and the path `C:\Share`:

```
PS C:\> New-SmbShare -Path C:\Share -Name Share
```

The following example removes the SMB share named Share:

```
PS C:\> Remove-SmbShare -Name Share
```

SmbSession

These cmdlets are used for managing sessions that currently are connected to the SMB shares on the local or remote machine.

Usage examples

The following example enumerates the current SMB session connections on the local machine:

```
PS C:\ > Get-SmbSession


SessionId      ClientComputerName   ClientUserName    NumOpens
---------      ------------------   --------------    --------
377957122073   \\[fe80::952f:e0...  MDN\Administrator 3
```

The following example closes all the SMB session connections for the mdn\administrator user on the local machine:

```
PS C:\> Get-SmbSession -ClientUserName mdn\administrator |
Close-SmbSession
```

SmbShareAccess

These cmdlets are used to retrieve, grant, and revoke SMB share access.

Usage examples

The following example retrieves a share's access control list:

```
PS C:\ > Get-SmbShareAccess -Name Share


Name    ScopeName   AccountName  AccessControlType AccessRight

----    ---------   -----------  ----------------- -----------

Share   *           Everyone     Allow             Read
```

The following example revokes the `Full` access right to a SMB share from the Everyone user group:

```
PS C:\> Get-SmbShareAccess –Name Share | Revoke-SmbShareAccess –
AccountName Everyone


Name   ScopeName    AccountName  AccessControlType AccessRight

----   ---------    -----------  ----------------- -----------

Share  *            Everyone     Deny              Full
```

The following example completely blocks a share access without specifying the access control type:

```
PS C:\> Block-SmbShareAccess –Name Share –AccountName Everyone


Name  ScopeName    AccountName   AccessControlType AccessRight

----  ---------    -----------   ----------------- -----------

Share    *           Everyone   Deny               Full

Share    *           Everyone   Allow              Full
```

SmbOpenFile

These cmdlets are intended to enumerate and close open files and directories that are being served through SMB shares on the local or remote machine.

Usage examples

The following example gets a list of open SMB files:

```
PS C:\> Get-SMBOpenFile | Select-Object ClientComputerName,ClientUserName
,Path


ClientComputerName  ClientUserName     Path
------------------  --------------     ----
\\10.0.0.3          MDN\Administrator  C:\\Users\Administrator
\\10.0.0.3          MDN\Administrator  C:\\Users\Administ...
```

The following example closes all the open SMB files and directories for a particular user:

```
PS C:\ > Get-SmbOpenFile -ClientUserName mdn\administrator | Close-
SmbOpenFile


Confirm
Are you sure you want to perform this action?
Performing operation 'Close-File' on Target '395136991761'.
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help
(default is "Y"): a
```

PrintManagement module

This module is available in Windows 8 and Windows Server 2012 and no feature is required. There are a total of 18 commands. The following command is used to import this module:

```
PS C:\> Import-Module PrintManagement
```

This module is used for managing printers, print drivers, printer configuration, and print jobs. These cmdlets are based on CIM classes and support CIM sessions.

Printer

These cmdlets are used for retrieving, adding, removing, and modifying printers.

Usage examples

The following example gets the OneNote printer and displays its name, type, driver name, and port:

```
PS C:\ > Get-Printer -Name *OneNote* | Select-Object
Name,Type,DriverName,PortName


Name        Type     DriverName                    PortName

----        ----     ----------                    --------

OneNote...  Local    Remote Desktop Easy Print     TS014
```

The following example sets the comment on the XPS Document Writer printer:

```
PS C:\> Get-Printer -Name "Microsoft XPS Document Writer" | Set-Printer
-Comment "Local Printer"
```

PrintJob

These cmdlets manage running and historic printer jobs. Jobs can be stopped, paused, restarted, and resumed.

Usage examples

The following example stops all printer jobs that are currently running on the Microsoft XPS Document Writer:

```
PS C:\> Get-Printer -Name "Microsoft XPS Document Writer"  |
Get-PrintJob | Remove-PrintJob
```

PrintConfiguration

These cmdlets get and set the print configuration for a printer.

Usage examples

The following example returns the print configuration for the Microsoft XPS Document Writer printer:

```
PS C:\ > Get-Printer -Name "Microsoft XPS Document Writer" | Get-
PrintConfiguration


PrinterName     ComputerName    Collate    Color DuplexingMode

-----------     ------------    -------    ----- -------------

Microsoft XP...                 False      True       OneSided
```

Windows data access control module

This module is available in Windows 8 and Windows Server 2012 and it requires no feature. There are a total of 12 cmdlets in this module. The following command is used to import this module:

```
PS C:\> Import-Module wdac
```

The `WDAC` module is used for managing ODBC connections. This module is based on CIM classes and supports CIM sessions.

OdbcDsn

These cmdlets are used for managing ODBC **Data Source Names** (**DSN**) defined on the local or remote system. It's possible to enumerate, create, update, and remove a DSN.

Usage examples

The following example creates a new ODBC DSN on the local machine named InternalDsn, connecting the SQL Server ODBC driver to the database LocalDatabase stored on the server sql2008. Note that depending on the driver, the property values may vary:

```
PS C:\ > Add-OdbcDsn -Name InternalDsn -DsnType User -DriverName "SQL
Server" -SetPropertyValue @("Database=LocalDatabase","Server=sql2008")
```

The following example retrieves the defined ODBC DSNs on the local machine:

```
PS C:\ > Get-OdbcDsn


Name       : InternalDsn
DsnType    : User
Platform   : 32/64-bit
DriverName : SQL Server
Attribute  : {Server, Database}
```

OdbcDriver

These cmdlets are used for managing the ODBC drivers that are installed on the system. ODBC drivers are used to connect to different types of databases.

Usage examples

The following example gets the 64-bit ODBC drivers on the local machine and formats the output as a list:

```
PS C:\ > Get-OdbcDriver –Platform 64-bit | Format-List


Name       Platform       Attribute
----       --------       ---------
SQL Server 64-bit         {Driver, APILevel, FileUsage,...}
```

DnsClient module

This module is available in Windows 8 and Windows Server 2012 and it requires no feature. There are a total of 15 cmdlets.

```
PS C:\> Import-Module DnsClient
```

The `DnsClient` module offers cmdlets for interacting with the local DNS client. This includes cmdlets for resolving DNS names, viewing the cached DNS records, and enumerating DNS servers. This cmdlet is based on a CIM provider and can utilize CIM sessions.

Resolve-DnsName

This cmdlet resolves the specified name to an IP address. This cmdlet offers many different kinds of filtering options. It also allows for queries based on DNS record type, such as A and AAAA.

Usage examples

The following example resolves the computer name driscoll-laptop:

```
PS C:\ > Resolve-DnsName –Name driscoll-laptop | Format-List


Name       : driscoll-desk
Type       : AAAA
TTL        : 30
DataLength : 16
Section    : Answer
IPAddress  : fe80::48d1:aec:7a7a:1a8
```

```
Name       : driscoll-desk
Type       : A
TTL        : 30
DataLength : 4
Section    : Answer
IPAddress  : 10.0.0.8
```

DnsClientCache

These cmdlets are used for viewing and clearing the DNS client cache records.

Usage examples

The following example returns a list of the DNS records currently in the DNS cache and selects the name of the device:

```
PS C:\ > Get-DnsClientCache| Select-Object -Property Name


Name
----
driscoll-laptop
sqm.telemetry.microsoft.com
driscoll-desk
```

The following example clears the DNS cache:

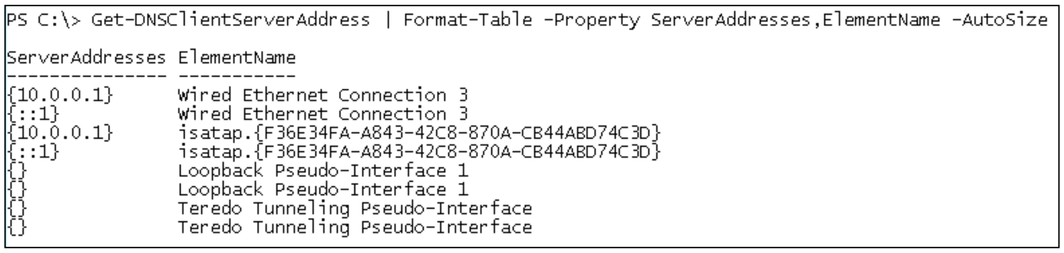
```
#Clear the DNS cache.
PS C:\ > Clear-DNSClientCache


#Get a list of the host names currently in the cache
PS C:\> Get-DNSClientCache | Select-Object -Property Name
```

Get-DnsServerAddress

This cmdlet returns the DNS server addresses that the current machine is using to resolve host names. This cmdlet will return the loopback, IPv4, and IPv6 addresses that are being used. The name of the adapter is also included.

Usage examples

The following example returns the names and addresses of the DNS servers that are currently being utilized by the local machine:



```
PS C:\> Get-DNSClientServerAddress | Format-Table -Property ServerAddresses,ElementName -AutoSize

ServerAddresses ElementName
--------------- -----------
{10.0.0.1}      Wired Ethernet Connection 3
{::1}           Wired Ethernet Connection 3
{10.0.0.1}      isatap.{F36E34FA-A843-42C8-870A-CB44ABD74C3D}
{::1}           isatap.{F36E34FA-A843-42C8-870A-CB44ABD74C3D}
{}              Loopback Pseudo-Interface 1
{}              Loopback Pseudo-Interface 1
{}              Teredo Tunneling Pseudo-Interface
{}              Teredo Tunneling Pseudo-Interface
```

DNSGlobalSettings

These cmdlets are used to enumerate and modify the global DNS settings for the local or remote DNS client. Some of the settings include root suffixes and whether to append parent suffixes.

Usage examples

The following example returns a list of the current DNS global settings for the local machine:

```
PS C:\> Get-DNSClientGlobalSetting


UseSuffixSearchList : True
SuffixSearchList    : {mdn.ard}
UseDevolution       : True
DevolutionLevel     : 0
```

The following example replaces the DNS suffix search list with mdnvdi:

```
PS C:\> Set-DNSClientGlobalSetting -SuffixSearchList {"mdnvdi" }
```

Storage module

This is available in Windows 8 and Windows Server 2012 and it requires no features. There are a total of 64 cmdlets in this module:

```
PS C:\> Import-Module Storage
```

The storage module exposes all kinds of cmdlets for managing storage devices on a machine. Some of the resources that can be managed include physical and virtual disks, volumes, and partitions. In addition to managing physical or virtual storage resources, the module can manage the storage subsystem, pools, and providers. The cmdlets in this module are based on a CIM provider and can utilize CIM sessions.

Disk

These cmdlets manage both physical and virtual disks. In addition to enumerating and setting properties of existing disks, there are cmdlets for initializing disks with boot sectors.

Usage examples

The following example returns the disks on the local machine:

```
PS C:\ > Get-Disk
Number Friendly Name OperationalStatus  Total Size Partition
------ ------------- -----------------  ---------- -----------
0      Virtual ...   Online             127 GB     MBR
1      Msft    ...   Online             200 MB     MBR
```

The following example sets disk number 1 to offline and returns the result using `Get-Disk`:

```
PS C:\Users\Administrator> set-disk -Number 1 -IsOffline $true
PS C:\Users\Administrator> Get-Disk -Number 1

Number Friendly Name    OperationalStatus Total Size Partition
------ -------------    ----------------- ---------- ---------
1      Msft       ...   Offline           200 MB     MBR
```

The following example initializes disk number 1 with the Master Boot Record partition style:

```
PS C:\> Initialize-Disk -Number 1 -PartitionStyle MBR
```

Partition

These cmdlets manage disk partitions. It is possible to enumerate, modify, create, remove, and even resize partitions.

Usage examples

The following example creates a new partition on virtual disk number 1. This will create a new partition but will not format it. It will assign the driver letter D to the disk:

```
PS C:\> New-Partition –DiskNumber 1 –DriveLetter D
   Disk Number: 1

Number DriveLetter Offset   TotalSize Type
------ ----------- ------   --------- ----
2      D           33619968 67.88 MB  Basic
```

The following example resizes an existing partition on disk number 1 to 50 MB:

```
PS C:\ > Resize-Partition -DiskNumber 1 -Number 2 -Size 50MB
PS C:\ > Get-Partition -DiskNumber 1
   Disk Number: 1

Number DriveLetter Offset   TotalSize Type
------ ----------- ------   --------- ----
1                  17408    32 MB     Reserved
2      D          33619968 50 MB     Basic
```

Volume

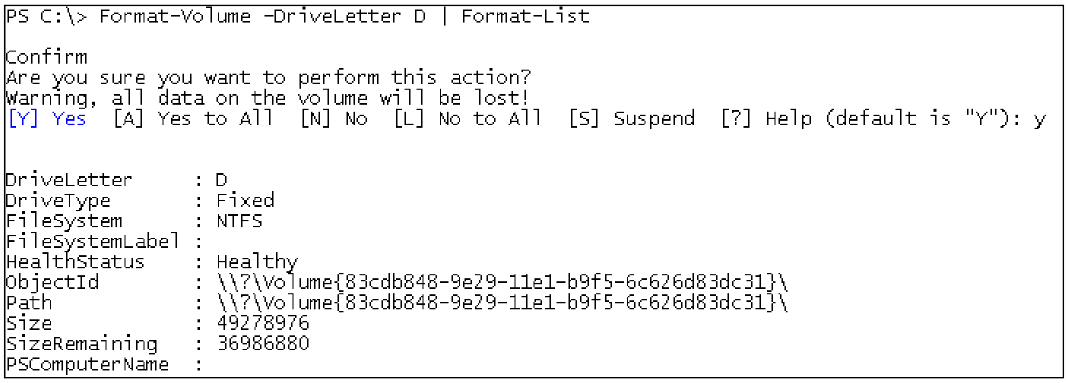
These cmdlets deal with disk volumes. This set of cmdlets can enumerate volumes, format and repair volumes and even optimize volumes.

Usage examples

The following example gets a list of the volumes and displays their drive letter, description, and size:

```
PS C:\ > Get-Volume | Select-Object -Property DriveLetter,FileSystemLabel
,Size

DriveLetter FileSystemLabel   Size
----------- ---------------   ----
            System Reserved   366997504
C                             135994011648
A                             0
D                             0
```

The following example formats an unallocated partition on a virtual disk. The default file system format is NTFS:



```
PS C:\> Format-Volume -DriveLetter D | Format-List

Confirm
Are you sure you want to perform this action?
Warning, all data on the volume will be lost!
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "Y"): y


DriveLetter     : D
DriveType       : Fixed
FileSystem      : NTFS
FileSystemLabel :
HealthStatus    : Healthy
ObjectId        : \\?\Volume{83cdb848-9e29-11e1-b9f5-6c626d83dc31}\
Path            : \\?\Volume{83cdb848-9e29-11e1-b9f5-6c626d83dc31}\
Size            : 49278976
SizeRemaining   : 36986880
PSComputerName  :
```

Roles and feature based modules

These modules are available for specified roles and features of the Windows 8 and Windows Server 2012 operating systems. They are not available to users who install the Windows Management Framework outside of these operating systems. Many of these modules require that a role or feature be installed. Several will not even be available until the role or feature is installed.

Hyper-V module

This module requires Hyper-V role, and it is imported using the following command:

```
PS C:\> Import-Module Hyper-V
```

Hyper-V is Microsoft's virtualization platform. It is a type 1 hypervisor and comes packaged as a configurable role in both Windows 8 and Windows Server 2012. Although this role was available in previous versions of the Windows Server family of operating systems, this is the first version to offer native PowerShell support, as well as run on the client version of the operating system. This module will not be available until the role is enabled.

VM

These cmdlets are used to directly work with virtual machines. The set includes cmdlets to start, stop, snapshot, and enumerate virtual machines on local and remote systems. These cmdlets are often paired with other virtual hardware cmdlets that allow for even finer grained management of virtual systems.

Usage examples

The following example shuts down all machines on the local server. The `Stop-VM` cmdlet can also power off machines;

```
PS C:\> Get-VM | Stop-VM -PassThru | Select-Object -Properties
Name,State,Status


Name      State Status
----      ----- ------
win7x6401 Off   Operating normally
windows7  Off   Operating normally
```

The following example shows how to create a new virtual machine. The newly created virtual machine will have one gigabyte of virtual memory, a 50 GB virtual hard disk and be named win7x6401. The virtual machine will also have several other hardware components added by default, such as a DVD drive:.

```
PS C:\> New-vm -Name win7x6401 -MemoryStartupBytes 1GB -NewVHDSize 50GB
-NewVHDPath C:\vhd\win7x6401.
Vhdx | Select-Object -Properties Name,State,Status


Name          State      Status
----          -----      ------
win7x6401     Off        Operating normally
```

The following example takes a snapshot of the current state of the virtual machine. It then returns a list of the snapshots for the virtual machine:

```
PS C:\ > Get-VM -Name win7x6401 | Checkpoint-VM
PS C:\ > Get-VM -Name win7x6401| Get-VMSnapshot | Select-Object -Property
Name


Name
----
win7x6401 - (2/26/2012 - 9:54:25 AM)
```

VMDvdDrive, VMHardDiskDrive, and VMNetworkAdapter

These cmdlets manage virtual hardware for virtual machines. They are typically coupled with the Get-VM cmdlet. This list is not complete. There are several other cmdlets that can manage additional hardware, such as processors, COM ports, and even the BIOS. Most cmdlets that attempt to modify an existing virtual machine will require that the machine is not running.

Usage examples

The following example removes the virtual DVD drive from the virtual machine win7x6401:

```
PS C:\> Get-VM win7x6401 | Get-VMDvdDrive | Remove-VMDvdDrive –Verbose


VERBOSE: Remove-VMDvdDrive will remove DVD Drive on IDE controller number
0 at location 1 from the virtual machine
"win7x6401".
```

The following example adds a new network adapter to the virtual machine "win7x6401."

```
PS C:\> Get-VM win7x6401 | Add-VMNetworkAdapter –Verbose


VERBOSE: Current VMobject  = Microsoft.HyperV.PowerShell.VirtualMachine[]
VERBOSE: Add-VMNetworkAdapter will add a network adapter to virtual
machine "win7x6401".
```

Measure-VM

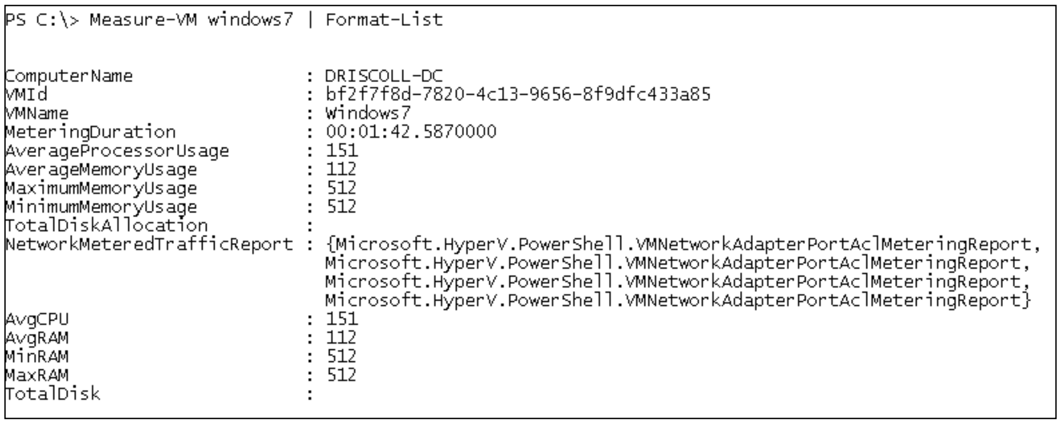
This cmdlet paired with `Enable-VMResourceMetering`, returns the historical resource usage for the virtual machine provided. This can be useful for tracking down troublesome, high-usage memory or processor time virtual machines.

Usage examples

The following example enables resource metering for the virtual machine windows7:

```
PS C:\ > Enable-VMResourceMetering -VMName windows7
```

The following example retrieves a snapshot of the resource usage for the virtual machine windows7:



```
PS C:\> Measure-VM windows7 | Format-List


ComputerName                 : DRISCOLL-DC
VMId                         : bf2f7f8d-7820-4c13-9656-8f9dfc433a85
VMName                       : Windows7
MeteringDuration             : 00:01:42.5870000
AverageProcessorUsage        : 151
AverageMemoryUsage           : 112
MaximumMemoryUsage           : 512
MinimumMemoryUsage           : 512
TotalDiskAllocation          :
NetworkMeteredTrafficReport  : {Microsoft.HyperV.PowerShell.VMNetworkAdapterPortAclMeteringReport,
                               Microsoft.HyperV.PowerShell.VMNetworkAdapterPortAclMeteringReport,
                               Microsoft.HyperV.PowerShell.VMNetworkAdapterPortAclMeteringReport,
                               Microsoft.HyperV.PowerShell.VMNetworkAdapterPortAclMeteringReport}
AvgCPU                       : 151
AvgRAM                       : 112
MinRAM                       : 512
MaxRAM                       : 512
TotalDisk                    :
```

VMHost

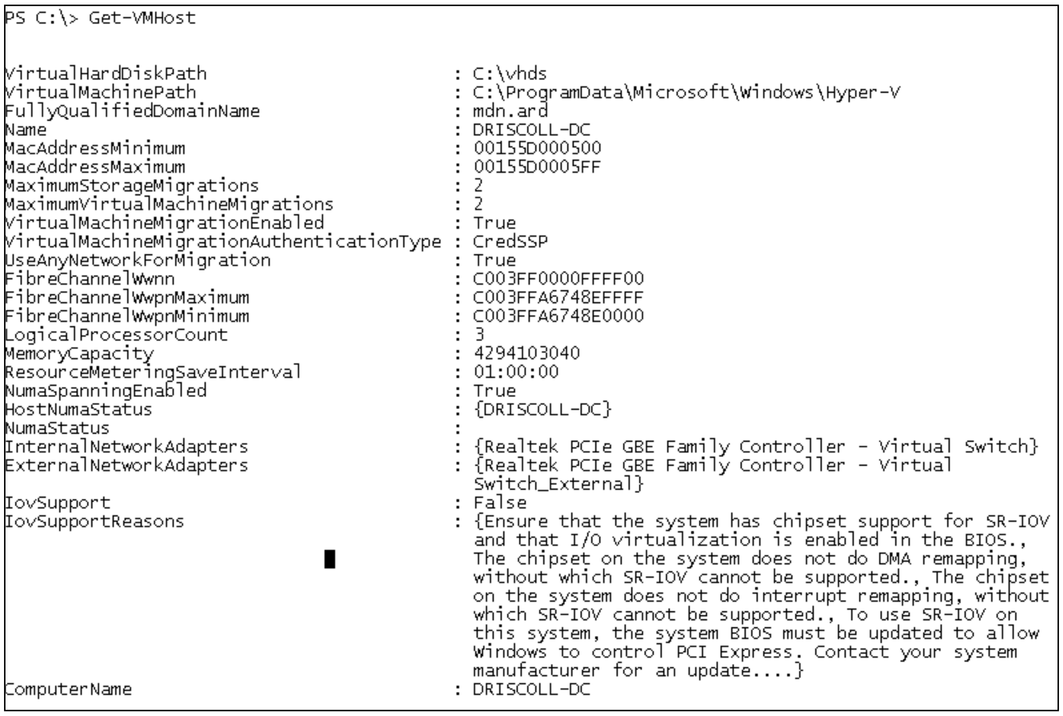
These cmdlets are used for managing Hyper-V host settings. Some of these settings include the MAC address range and the default path to store virtual machines and VHD files.

Usage examples

The following example sets the local host's resource metering save interval to two hours and the default virtual hard disk path to `C:\VHDS`:

```
PS C:\> Set-VMhost -ResourceMeteringSaveInterval (New-TimeSpan -Hours 2)
-VirtualHardDiskPath C:\VHDS
```

The following example returns the current settings for the local virtual machine host:



```
PS C:\> Get-VMHost


VirtualHardDiskPath                         : C:\vhds
VirtualMachinePath                          : C:\ProgramData\Microsoft\Windows\Hyper-V
FullyQualifiedDomainName                    : mdn.ard
Name                                        : DRISCOLL-DC
MacAddressMinimum                           : 00155D000500
MacAddressMaximum                           : 00155D0005FF
MaximumStorageMigrations                    : 2
MaximumVirtualMachineMigrations             : 2
VirtualMachineMigrationEnabled              : True
VirtualMachineMigrationAuthenticationType   : CredSSP
UseAnyNetworkForMigration                   : True
FibreChannelWwnn                            : C003FF0000FFFF00
FibreChannelWwpnMaximum                     : C003FFA6748EFFFF
FibreChannelWwpnMinimum                     : C003FFA6748E0000
LogicalProcessorCount                       : 3
MemoryCapacity                              : 4294103040
ResourceMeteringSaveInterval                : 01:00:00
NumaSpanningEnabled                         : True
HostNumaStatus                              : {DRISCOLL-DC}
NumaStatus                                  :
InternalNetworkAdapters                     : {Realtek PCIe GBE Family Controller - Virtual Switch}
ExternalNetworkAdapters                     : {Realtek PCIe GBE Family Controller - Virtual
                                              Switch_External}
IovSupport                                  : False
IovSupportReasons                           : {Ensure that the system has chipset support for SR-IOV
                                              and that I/O virtualization is enabled in the BIOS.,
                                              The chipset on the system does not do DMA remapping,
                                              without which SR-IOV cannot be supported., The chipset
                                              on the system does not do interrupt remapping, without
                                              which SR-IOV cannot be supported., To use SR-IOV on
                                              this system, the system BIOS must be updated to allow
                                              Windows to control PCI Express. Contact your system
                                              manufacturer for an update....}
ComputerName                                : DRISCOLL-DC
```

Active Directory deployment module

This module is available in Windows Server 2012 and it requires Active Directory Domain Services role. It has a total of 73 cmdlets. The following command shows how to import the module:

```
PS C:\> Import-Module ADDSDeployment
```

The Active Directory deployment module is used for installing and configuring Active Directory components such as domain controllers. In Windows Server 2012, Microsoft has deprecated the `DCPromo.exe` command line tool that has historically been used to perform the operations found in this module. In addition to installation, the module comes with numerous cmdlets for testing various aspects of an ADDS configuration.

Install-ADDSForest

These cmdlets are used to install a domain forest.

Usage examples

The following example installs a forest with the domain name MDN.ARD with a domain and forest mode of 2008 R2. Once the forest has been installed, the machine will be rebooted:

```
PS C:\> Install-ADDSForest –DomainName MDN.ARD –DomainMode
Win2008R2 –ForestMode Win2008R2 –RebootOnCompletion
```

ADDSDomainController

These cmdlets are used to promote and demote servers to and from domain controller status.

Usage examples

The following example promotes the current machine to a domain controller in the domain MDN.ARD:

```
PS C:\> Install-ADDSDomainController –DomainName MDN.ARD

cmdlet Install-ADDSDomainController at command pipeline position 1

Supply values for the following parameters:

SafeModeAdministratorPassword: ************
```

The following example demotes the last domain controller in a domain and removes the application partitions associated with it. The machine will restart once this is complete:

```
PS C:\ > Uninstall-ADDSDomainController -LastDomainControllerInDomain
-RemoveApplicationPartitions


cmdlet Uninstall-ADDSDomainController at command pipeline position 1

Supply values for the following parameters:

LocalAdministratorPassword: ****
```

Test ADDSDeployment

These cmdlets test all kinds of pieces of the ADDS deployment scenario. Some of the configuration aspects that can be tested include credential validity, NetBIOS names, and even available disk space. Many of these cmdlets are run as a part of the validation steps for some of the previously mentioned cmdlets.

Usage examples

The following example shows how we can use one of the test cmdlets to verify the status of a demote operation:

```
PS C:\ > Test-ADDSDomainControllerUninstallation | Select-Object
-ExpandProperty message


You indicated that this Active Directory domain controller is not the
last domain controller for the domain "mdn.ard".
However, no other domain controller for that domain can be contacted.
Proceeding will cause any Active Directory Domain Services changes that
have been made on this domain controller to be lost.  To proceed anyway,
set the 'IgnoreIsLast
DCInDomainMismatch' option to 'YES'.
```

AppX module

This module is available in Windows 8 and Windows Server 2012 and it requires no role or feature. It has a total of five cmdlets:

```
PS C:\> Import-Module AppX
```

AppX is a new type of application packaging and deployment model that is part of Windows 8 and Windows Server 2012. Applications are packed with an application manifest file that defines metadata about the application. This metadata includes properties such as the name and copyright information, the dependencies, and the OS extensions that the application includes. The cmdlets in this module are designed to enumerate and deploy AppX packages. The cmdlets are not based on a CIM provider and do not offer a CIM session. The cmdlets also do not offer a `ComputerName` parameter and are intended to be run locally. To run remotely, we will have to use PowerShell remoting.

AppXPackage

These cmdlets can add, enumerate, and remove existing AppX packages from the local system.

Usage examples

The following example returns a list of the AppX packages currently installed on the local system:

```
PS C:\> Get-AppXPackage

Name                   : windows.immersivecontrolpanel

Publisher              : CN=Microsoft Windows, O=Microsoft Corporation,
L=Redmond, S=Washington, C=US

Architecture           : Neutral

ResourceId             :

Version                : 6.2.0.0

PackageFullName        : windows.immersivecontrolpanel_6.2.0.0_neutral_
neutral_cw5n1h2txyewy

InstallLocation        : C:\Windows\System32\ImmersiveControlPanel

IsFramework            : False

PackageUserInformation : {}
```

The following example removes an AppX package using the `WhatIf` parameter:

```
PS C:\Users\Administrator> remove-appxpackage -Package "windows.
Immersivecontrolpanel" -whatif


What if: Performing operation "Remove package" on Target "windows.
Immersivecontrolpanel".
```

Get-AppXPackageManifest

This cmdlet returns the AppX package manifest as an XML document.

Usage examples

The following example returns the AppX package manifest for the `Windows.ImmersiveControlPanel` as an XML string:

```
PS C:\> Get-AppXPackageManifest -Package "windows.Immersivecontrolpanel"
| Select-Object -Property OuterXml


OuterXml
--------
<?xml version="1.0" encoding="utf-8"?><Package xmlns="http://schemas.
microsoft.com/appx/2010/...
```

Other modules

In the following section we will look over some of the new modules included with Windows 8 and Windows Server 2012. These modules are all role or feature based and some require that these roles be installed before they will even be available on the command line. We will not be looking at individual cmdlets in this section.

Remote Desktop management module

This module is available in Windows 8 and Windows Server 2012 and it requires no role or feature. There are a total of 103 cmdlets. The following command shows how to import this module:

```
PS C:\> Import-Module RemoteDesktop
```

The `Remote Desktop` management module exposes cmdlets for work with the remote desktop components found in Windows. The cmdlets include sets for working with pooled session hosts, virtual desktops, and working with brokered resources such as RemoteApps and remote desktops. This module can also configure roles, such as the Web Access and Gateway. Although the cmdlet does not require the `Remote Desktop Services` role to be installed, it requires access to an RDS-enabled server to manage.

BranchCache module

This module is available in Windows 8 and Windows Server 2012 and it requires `BranchCache` feature. It includes a total of 31 cmdlets. The following command is used to import this module:

```
PS C:\> Import-Module BranchCache
```

`BranchCache` is a Windows server and client feature that caches files locally that are stored on a remote, central location. As the BranchCache name implies, files are typically stored within a central headquarters while the outlying branches would consume them through network shares. Because of the limited bandwidth and high latency often found in WAN networks, `BranchCache` aims to limit this impact on shared file access. `BranchCache` works through both a hosted model, where a single server is the local cache and a peer-to-peer model, where clients themselves cache files and serve them to other clients. `BranchCache` has been available since Windows 7 and Server 2008 R2.

Once configured, the distribution server that hosts the BranchCache-enabled shares can be accessed by clients that have `BranchCache` enabled locally. Files within shares that have enabled `BranchCache` will have hashes calculated and published, so that client machines can easily identify files without downloading them. There are numerous new cmdlets found in this module to view the status of `BranchCache`, disable the system, or publish the file hashes for machines to consume.

Windows Update Services module

This module is available in Windows Server 2012 and it requires Windows Server Update Services feature. It includes a total of 13 cmdlets. The following command is used to import this module:

```
PS C:\> Import-Module UpdateServices
```

The **Windows Update Services** (**WSUS**) Module is used to manage a Windows update server. The cmdlets in this module can retrieve computers, updates, and products managed by WSUS. It is possible to enable product updating in WSUS through the use of these cmdlets as well.

Summary

In this chapter we looked at some of the new modules available in Windows 8 and Windows Server 2012. We saw through some simple examples, how to use some of the notable cmdlets in each one of the modules that are for more general use, and some of the more specific modules for particular roles that are enabled for the operating system.

The number of PowerShell cmdlets and modules in the new version of the Windows Management Framework and Windows operating systems is staggering. As we move forward with the adoption of the next version of these technologies, it will become necessary to have some knowledge of the capabilities that are built into the system and understand how to learn more about the functionality we have at our fingertips. Harnessing the strength that PowerShell offers will make us better administrators by reducing the need for manual interactions with an ever growing ecosystem of computer infrastructure.

Index