The PowerShell Scripting & Toolmaking Book

Forever Edition

by Don Jones and Jeffery Hicks

The PowerShell Scripting and Toolmaking Book

Don Jones and Jeff Hicks

This book is for sale at http://leanpub.com/powershell-scripting-toolmaking

This version was published on 2024-05-23

Tweet This Book!

Please help Don Jones and Jeff Hicks by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I got The #PowerShell #Toolmaking book http://leanpub.com/powershell-scripting-toolmaking @JeffHicks + @concentrateddon

The suggested hashtag for this book is #PowerShellToolmaking.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#PowerShellToolmaking

Contents

About This Book **1**

Dedication **2**
Acknowledgments 2

About the Authors **3**
Additional Credits 3

Foreword **4**

Introduction **6**
Pre-Requisites 6
Versioning 6
The Journey 6
Following Along 7
Providing Feedback 7

Part 1: Review PowerShell Toolmaking **1**

Functions, the Right Way **2**
Tool Design 2
Start with a Command 2
Build a Basic Function and Module 3
Adding CmdletBinding and Parameterizing 3
Emitting Objects as Output 3
Using Verbose, Warning, and Informational Output 3
Comment-Based Help 4
Handling Errors 4
Are You Ready 4

Part 2: Professional-Grade Toolmaking **5**

Publishing Your Tools **6**
Begin with a Manifest 6
Publishing to PowerShell Gallery 9
Publishing to Private Repositories or Galleries 10
Your Turn 11
Let's Review 12

Part 3: Controller Scripts and Visual Scripting **13**

Proxy Functions **14**
For Example 14
Creating the Proxy Base 14
Modifying the Proxy 17
Adding or Removing Parameters 19
Your Turn 20
Let’s Review 22

Part 4: The Data Connection **23**

Working with SQL Server Data **24**
SQL Server Terminology and Facts 24
Connecting to the Server and Database 25
Writing a Query 25
Running a Query 28
Invoke-SqlCmd 30
Thinking About Tool Design Patterns 30
Let’s Review 31
Review Answers 31

Part 5: Seriously Advanced Toolmaking **32**

Measuring Tool Performance **33**
Is Performance Important 33
Measure What’s Important 33
Factors Affecting Performance 35
Key Take-Away 37

Part 6: Pester **38**

Why Pester Matters **39**

Part 7: PowerShell 7 Scripting **41**

Cross Platform Scripting **42**
Know Your OS 42
State Your Requirements 42
Testing Variables 43
Environment Variables 45
Paths 45
Watch Your Aliases 46
Using Culture 46
Leverage Remoting 47
Custom Module Manifests 54

About This Book

The ‘Forever Edition’ of this book is published on Leanpub[1], a digital publishing platform. That means the book is published as we write it, and *that* means we’ll be able to revise it as needed in the future. We also appreciate your patience with any typographical errors, and we appreciate you pointing them out to us - to keep the book as “agile” as possible, we’re forgoing a traditional copyedit. We hope that you’ll appreciate getting the technical content quickly, and won’t mind helping us catch any errors we may have made. You paid a bit more for the book than a traditional one, but that up-front price means you can come back whenever you like and download the latest version. We plan to expand and improve the book pretty much forever, so it’s hopefully the last one you’ll need to buy on this topic!

You may also find this book offered by traditional booksellers like Amazon. In those cases, the book is sold as a specific edition, such as “Second Edition.” These represent a point-in-time snapshot of the book and are offered at a lower price than the Leanpub-published version. **These traditionally published editions do not include future updates**.

If you purchased this book, thank you. Know that writing a book like this takes hundreds of hours of effort, during which we’re not making any other income. Your purchase price is important to keeping a roof over our families’ heads and food on our tables. Please treat your copy of the book as **your** copy - it isn’t to be uploaded anywhere, and you aren’t meant to give copies to other people. We’ve made sure to provide a DRM-free file (excepting any DRM added by a bookseller other than Leanpub) so that you can use your copy any way that’s convenient for you. We appreciate your respecting our rights and not making unauthorized copies of this work.

If you got this book for free from someplace, know that you are making it difficult for us to write books. When we can’t make even a small amount of money from our books, we’re encouraged to stop writing them. If you find this book useful, we would greatly appreciate you purchasing a copy from LeapPub.com or another bookseller. When you do, you’ll be letting us know that books like this are useful to you and that you want people like us to continue creating them.

Please note that this book is not authorized for classroom use unless a unique copy has been purchased for each student. No one is authorized or licensed to manually reproduce the PDF version of this book for use in any kind of class or training environment.

---

[1]https://leanpub.com

Dedication

This book is fondly dedicated to the many hardworking PowerShell users who have, for more than a decade, invited us into their lives through our books, conference appearances, instructional videos, live classes, and more. We're always humbled and honored by your support and kindness, and you inspire us to always try harder and to do more. Thank you.

Acknowledgments

Thanks to Michael Bender, who has selflessly provided a technical review of the book. Any remaining errors are, of course, still the authors' fault, but Michael has been tireless in helping us catch many of them.

About the Authors

Don Jones was an on-stage presenter during the launch of Windows PowerShell at TechEd Europe 2006 and was the co-author of the first three published books on PowerShell. He was a 16-year recipient of Microsoftâ€™s Most Valuable Professional (MVP) Award, co-founder of PowerShell Summit and The DevOps Collective, and co-author of the bestselling Learn Windows PowerShell in a Month of Lunches through its first three editions. Don also wrote more than sixty books on a variety of technology and business topics. Today, most of Donâ€™s writing consists of fantasy and science fiction novels, which you can find at DonJones.com[2].

Jeff Hicks is an IT veteran with over 30 years of experience, much of it spent as an IT infrastructure consultant specializing in Microsoft server technologies with an emphasis on automation and efficiency. He is a multi-year recipient of the Microsoft MVP Award for his PowerShell-related contributions. He works today as an independent author, teacher, and consultant. Jeff has taught and presented about PowerShell and the benefits of automation to IT Pros worldwide for over 25 years. He has authored and co-authored several books, writes for numerous online sites, is a Pluralsight[3] author, and is a frequent speaker at technology conferences and user groups.

You can learn about all of Jeff's online identities and activities at jdhitsolutions.github.io[4].

Additional Credits

Technical editing has been helpfully provided not only by our readers but also by Michael Bender. We're grateful to Michael for not only catching a lot of big and little problems but for fixing most of them for us. Michael rocks, and you should watch his Pluralsight videos. However, anything Michael didn't catch is still firmly the authors' responsibility.

[2]https://donjones.com
[3]https://app.pluralsight.com/profile/author/jeff-hicks
[4]https://jdhitsolutions.github.io/

Foreword

After the success of *Learn PowerShell in a Month of Lunches*, Jeff and I wanted to write a book that took people down the next step, into actual scripting. The result, of course, was *Learn PowerShell Toolmaking in a Month of Lunches*. In the intervening years, as PowerShell gained more traction and greater adoption, we realized that there was a lot more of the story that we wanted to tell. We wanted to get into help authoring, unit testing, and more. We wanted to cover working with different data sources, coding in Visual Studio, and so on. These were really out of scope for the *Month of Lunches* series' format. And even in the "main" narrative of building a proper tool, we wanted to go into more depth. So while the *Month of Lunches* book was still a valuable tutorial in our minds, we wanted something with more teeth.

At the same time, this stuff is changing *very* fast these days. Fast enough that a traditional publishing process - which can add as much as four months to a book's publication - just can't keep up. Not only are we kind of constantly tweaking our narrative approach to explaining these topics, but the topics themselves are constantly evolving, thanks in part to many incredibly robust community-building add-ons like Pester, Platyps, and more.

So after some long, hard thinking, we decided to launch this effort. As an Agile-published book on Leanpub, we can continuously add new content, update old content, fix the mistakes you point out to us, and so on. We can then take major milestones and publish them as "snapshots" on places like Amazon, increasing the availability of this material. We hope you find the project as exciting and dynamic as we do, and we hope you're generous with your suggestions - which may be sent to us via the author contact form from this book's page on Leanpub.com. We'll continue to use traditional paper publishing, but through a self-publishing outlet that doesn't impose as much process overhead on getting the book in print. These hard copy editions will be a "snapshot" or "milestone edition" of the electronic version.

It's important to know that we still think traditional books have their place. *PowerShell Scripting in a Month of Lunches*, the successor to *Learn PowerShell Toolmaking in a Month of Lunches*, covers the kind of long-shelf-life narrative that is great for traditionally published books. It's an entry-level story about the right way to create PowerShell tools, and it's very much the predecessor to this book. If *Month of Lunches* is about getting your feet under you and putting them on the right path, this book is about refining your approach and going a good bit further on your journey.

Toolmaking, for us, is where PowerShell has always been headed. It's the foundation of a well-designed automation infrastructure, of a properly built DSC model, and of pretty much everything else you might do with PowerShell. Toolmaking is understanding what PowerShell is, how PowerShell *wants* to work, and how the world engages with PowerShell. Toolmaking is a big responsibility.

My first job out of high school was as an apprentice for the US Navy. In our first six weeks, we rotated through various shops - electronics, mechanical, and so on - to find a trade that we thought we'd want to apprentice for. For a couple of weeks, I was in a machine shop. Imagine a big, not-climate-controlled warehouse full of giant machines, each carving away at a piece of metal. Lubrication and metal chips are flying everywhere, and you wash shavings out of yourself every evening when you go home. It was disgusting, and I hated it. It was also boring - you set a block of metal into the machine, which might take hours to get precisely set up, and then you just sat back and kind of watched it happen. Ugh. Needless to say, I went into the aircraft mechanic trade instead. Anyway, in the machine shop, all the drill bits and stuff in the machine were called *tools* and *dies*. Back in the corner of the shop, in an enclosed, climate-controlled room sat a small number of nicely-dressed guys in front of computers. They were using CAD software to *design new tools and dies* for specific machining purposes. These were the *tool makers*, and I vowed that if I was ever going to be in this hell of a workplace, I wanted to be a *toolmaker* and not a *tool user*. And that's the genesis of this book's title. All of us - including the organizations we work for - will have happier, healthier, more comfortable lives as high-end, air-conditioned *toolmakers* rather than the sweaty, soaked, shavings-filled tool users out on the shop floor.

Enjoy!

Don Jones

Introduction

Pre-Requisites

We're assuming that you've already finished reading an entry-level tutorial like, *Learn Windows PowerShell in a Month of Lunches*, or that you've got some solid PowerShell experience already under your belt. Specifically, nothing on this list should scare you:

- Find commands and learn to use them by reading help
- Write very basic "batch file" style scripts
- Use multiple commands together in the pipeline
- Query WMI/CIM classes
- Connect to remote computers by using Remoting
- Manipulate command output to format it, export it, or convert it, using PowerShell commands to perform those tasks

If you've already done things like written functions in PowerShell, that's marvelous - but, you may need to be open to un-learning some things. Some of PowerShell's best practices and patterns aren't immediately obvious, and especially if you know how to code in another language, it's easy to go down a bad path in PowerShell. We're going to teach you the right way to do things, but you need to be willing to re-do some of your past work if you've been following the Wrong Ways.

We also assume that you've read *PowerShell Scripting in a Month of Lunches*, a book we wrote for Manning. It provides the core narrative of "the right way to write PowerShell functions and tools," this book essentially picks up where that one leaves off. Look for that book in late 2017 from Manning or your favorite bookseller. Part 1 of this book *briefly* slams through this "the right way" narrative just to make sure you've got it in your mind, but the *Month of Lunches* title digs into those ideas in detail.

Versioning

This book is primarily written against Windows PowerShell v5/v5.1 running on Microsoft Windows. In January 2018, Microsoft announced the General Availability of PowerShell Core 6.0, which is a distinct cross-platform "branch" of PowerShell. This branch has now become PowerShell 7, which was released in early 2020. As far as we can tell, everything we teach in this book applies to PowerShell 7, too - although some of our specific examples may still only work on Windows PowerShell, the concepts and techniques apply to PowerShell 7. However, PowerShell 7 includes some new scripting features which we'll cover in a dedicated chapter or two.

The Journey

This book is laid out into seven parts:

1. A quick overview of "the right way" to write functions.
2. Professional-grade toolmaking, where you amp up your skills, comes next in a second narrative. This part is less tightly coupled than the first, so you can just read what you think you need, but we still recommend reading the chapters in order.
3. Moving on from toolmaking for a moment, we'll cover different kinds of controller scripts that can put your tools to use. Read these in whatever order you like.
4. Data sources are often a frustrating point in PowerShell, and so this part is dedicated to those. Again, read whichever ones you think you need.
5. More advanced topics complete the book, and again you can just read these as you encounter a need for them.
6. A high-level introduction to using Pester in your toolmaking development.
7. Scripting for the PowerShell 7 and cross-platform world.

Following Along

We've taken some pains to provide review Q&A at the end of most chapters, and to provide lab exercises (and example answers) at the end of many chapters. We **strongly** encourage you to follow along and complete those exercises - *doing* is a lot more effective than just *reading*. We've tried to design the labs so that they only need a Windows client computer - so you won't need a complex, multi-machine lab setup. Of course, if you *have* more than one computer to play with, some of the labs can be more interesting since you can write tools that query multiple computers and so forth. But the code's the same even if you're just on a single Windows client, so you'll be learning what you need to learn.

Providing Feedback

Finally, we hope that you'll feel encouraged to give us feedback on this book. There's a "Contact the Authors" form on this book's page[5] on Leanpub.com.

[5]http://leanpub.com/powershell-scripting-toolmaking

Part 1: Review PowerShell Toolmaking

This first Part of the book is essentially a light-speed refresher of what *PowerShell Scripting in a Month of Lunches* covers. If you've read that book, or feel you have equivalent experience, then this short Part will help refresh you on some core terminology and techniques. If you haven't… well, we strongly recommend you get that fundamental information under your belt first.

Functions, the Right Way

This chapter is essentially meant to be a warp-speed review of the material we presented in the core narrative of *Learn PowerShell Toolmaking in a Month of Lunches* (and its successor, *PowerShell Scripting in a Month of Lunches*). This material is, for us, “fundamental” in nature, meaning it remains essentially unchanged from version to version of PowerShell. Consider this chapter a kind of “pre-req check;” if you can blast through this, nodding all the while and going, “yup,” then you’re good to skip to the next Part of this book. If you run across something where you’re like, “Wait, what?” then a review of those foundational, prerequisite books might be in order, along with a thorough reading of this Part of this book.

By the way, you’ll notice that our downloadable code samples for this book (the “PSToolmaking” module in PowerShell Gallery) contain the same code samples as the core “Part 2” narrative from *PowerShell Scripting in a Month of Lunches*. Those code samples also align with this book, and we use them in this chapter as illustrations.

Tool Design

We strongly advocate that you always begin building your tools by first *designing* them. What inputs will they require? What logical decisions will they have to make? What information will they output? What inputs might they consume from other tools, and how might their output be consumed? We try to answer all of these questions - often in writing! - upfront. Doing so helps us think through how our tool will be used, by different people at different times, and to make good decisions about how to build the tool when it comes time to code.

Start with a Command

Once we know what the tool’s going to do, we begin a console-based (never in a script editor) process of discovery and prototyping. Or, in plain English, we figure out the commands we’re going to need to run, figure out how to run them correctly, and, figure out what they produce and how we’re going to consume it. This isn’t a lightweight step - it can often be time-consuming, and it’s where all of your experimentation can occur.

A user in PowerShell.org’s forums once posted a request for help with the following:

I need a PowerShell script that will check a complete DFS Root, and report all targets and access-based enumeration for each. I then need the script to check all NTFS permissions on all the targets and list the security groups assigned. I then need this script to search 4 domains and report on the users in these groups.

And yup - that’s what “Start with a Command” means. We’d probably start by planning that out - inputs are clearly some kind of DFS root name or server name and an output path for the reports to be written.

Then the discovery process would begin: how can PowerShell connect to a DFS root? How can it enumerate targets? How can it resolve the target physical location and query NTFS permissions? Good ol' Google, and experience, would be our main tool here, and we wouldn't go an inch further until we had a text file full of answers, sample commands, and notes.

Build a Basic Function and Module

With all the functional bits in hand, we begin building tools. We almost always start with a basic function (no `[CmdletBinding()]` attribute) located in a script module. Why a script module? It's the end goal for us, and it's easier to test. We'd fill in our parameters, and start adding the functional bits to the function itself. We tend to add things in stages. So, taking that DFS example, we'd first write a function that simply connected to a DFS root and spewed out its targets. Once that was working, we'd add the bit for enumerating the targets' physical locations. Then we'd add permission querying… and so on, and so on, until we were done. None of that along-the-way output would be pretty - it'd just be verifying that our code was working.

Adding CmdletBinding and Parameterizing

We'd then professional-ize the function, adding `[CmdletBinding()]` to enable the common parameters. If we'd hard-coded any changeable values (we do that sometimes, during development), we'd move those into the `Param()` block. We'd also dress up our parameters, specifying data [types], mandatory-ness, pipeline input, validation attributes, and so on. We'd obviously re-test.

Emitting Objects as Output

Next, we work on cleaning up our output. We remove any "development" output created by `Write-Output` or `Write-Host` (yeah, it happens when you're hacking away). Our function's only output would be an object, and in the DFS example, it'd probably include stuff like the DFS root name, target, physical location, and a "child object" with permissions.

If you're reading that DFS example, we'd probably stop our function at the point where it gets the permissions on the DFS targets. The results of that operation could be used to unwind the users who were in the resulting groups - a procedure we'd write as a separate tool, in all likelihood.

Using Verbose, Warning, and Informational Output

If we hadn't already done so, we'd take the time to add `Write-Verbose` calls to our function so that we could track its progress. *We* tend to do that habitually as we write, almost instead of comments a lot of the time, but *we* have built that up as a habit. We'd add warning output as needed, and potentially add `Write-Information` calls if we wanted to create structured, queryable "sidebar" output.

Comment-Based Help

We'd definitely "dress up" our code using comment-based help if not full help (we cover that later in the book). We'd make sure to provide usage examples, documentation for each parameter, and a pretty detailed description of what the tool did.

Handling Errors

Finally, and again if we hadn't habitually done so already, we'd anticipate errors and try to handle them gracefully. "Permission Denied" querying permissions on a file? Handled - perhaps outputting an object, for that file, indicating the error.

Are You Ready

That's our process. The entire way through, we make sure we're conforming as much as possible to PowerShell standards. Input via parameters only; output only to the pipeline, and only as objects. Standardized naming, including Verb-Noun naming for the function, and parameter names that reflect existing patterns in native PowerShell commands. We try to get our command to look and feel as much like a "real" PowerShell command as possible, and we do that by carefully observing what "real" PowerShell commands do.

Ok, if you've gotten this far and you're still thinking, "Yup, got all that and good to go," then you're… well, you're good to go. Proceed.

Part 2: Professional-Grade Toolmaking

In this Part, we're going to try and take your toolmaking skills a bit further. This is the stuff that sets the beginners apart from the real pros. We've constructed these chapters into a kind of storyline, so each one builds on what the previous ones taught. That said, the storyline here isn't tightly coupled, so feel free to dive into whatever chapter seems of most interest or use to you. Because we're moving into Toolmaking areas that are more optional and as-you-need, you won't see "Your Turn" lab elements in every chapter - but that doesn't mean you shouldn't try and play along! Just follow along with your *own* code. However, when we include a "Your Turn" section, we obviously strongly suggest you follow along with that "lab."

Publishing Your Tools

Inevitably, you'll come to a point where you're ready to share your tools. Hopefully, you've put those into a PowerShell module, as we've been advocating throughout this book because in most cases it's a module that you'll share.

Begin with a Manifest

You'll typically need to ensure that your module has a .psd1 manifest file since most repositories will use information from that to populate repository metadata. Here's the manifest from our downloadable sample code.

```
#
# Module manifest for module 'PowerShell-Toolmaking'
#
# Generated by: Don Jones & Jeffery Hicks
#

@{

# Script module or binary module file associated with this manifest.
RootModule = 'PowerShell-Toolmaking.psm1'

# Version number of this module.
ModuleVersion = '2.0.0.0'

# Supported PSEditions
CompatiblePSEditions = @("Desktop")

# ID used to uniquely identify this module
GUID = '3926b244-469c-4434-a4b1-70ce3b0bfb5d'

# Author of this module
Author = 'Don Jones & Jeffery Hicks'

# Company or vendor of this module
CompanyName = 'Unknown'

# Copyright statement for this module
Copyright = '(c) 2024 Don Jones & Jeffery Hicks. All rights reserved.'

# Description of the functionality provided by this module
Description = "Sample for for 'The PowerShell Scripting and Toolmaking Book' by Don \
Jones and Jeffery Hicks published on Leanpub.com."

# Minimum version of the Windows PowerShell engine required by this module
# PowerShellVersion = ''

# Name of the Windows PowerShell host required by this module
# PowerShellHostName = ''

# Minimum version of the Windows PowerShell host required by this module
# PowerShellHostVersion = ''

# Minimum version of Microsoft .NET Framework required by this module. This prerequi\
site is valid for the PowerShell Desktop edition only.
# DotNetFrameworkVersion = ''

# Minimum version of the common language runtime (CLR) required by this module. This\
 prerequisite is valid for the PowerShell Desktop edition only.
# CLRVersion = ''

# Processor architecture (None, X86, Amd64) required by this module
# ProcessorArchitecture = ''

# Modules that must be imported into the global environment prior to importing this \
module
# RequiredModules = @()

# Assemblies that must be loaded prior to importing this module
# RequiredAssemblies = @()

# Script files (.ps1) that are run in the caller's environment prior to importing th\
is module.
# ScriptsToProcess = @()

# Type files (.ps1xml) to be loaded when importing this module
# TypesToProcess = @()

# Format files (.ps1xml) to be loaded when importing this module
# FormatsToProcess = @()

# Modules to import as nested modules of the module specified in RootModule/ModuleTo\
Process
# NestedModules = @()

# Functions to export from this module, for best performance, do not use wildcards a\
nd do not delete the entry, use an empty array if there are no functions to export.
FunctionsToExport = '*'

# Cmdlets to export from this module, for best performance, do not use wildcards and\
 do not delete the entry, use an empty array if there are no cmdlets to export.
CmdletsToExport = '*'

# Variables to export from this module
VariablesToExport = '*'

# Aliases to export from this module, for best performance, do not use wildcards and\
 do not delete the entry, use an empty array if there are no aliases to export.
AliasesToExport = '*'

# DSC resources to export from this module
# DscResourcesToExport = @()

# List of all modules packaged with this module
# ModuleList = @()

# List of all files packaged with this module
# FileList = @()

# Private data to pass to the module specified in RootModule/ModuleToProcess. This m\
ay also contain a PSData hashtable with additional module metadata used by PowerShel\
l.
PrivateData = @{

    PSData = @{

        # Tags applied to this module. These help with module discovery in online ga\
lleries.
        # Tags = @()

        # A URL to the license for this module.
        # LicenseUri = ''

        # A URL to the main website for this project.
        # ProjectUri = ''

        # A URL to an icon representing this module.
        # IconUri = ''

        # ReleaseNotes of this module
        # ReleaseNotes = ''

    } # End of PSData hashtable

} # End of PrivateData hashtable

# HelpInfo URI of this module
# HelpInfoURI = ''
# Default prefix for commands exported from this module. Override the default prefix\
 using Import-Module -Prefix.
# DefaultCommandPrefix = ''

}
```

A lot of this is commented out, which is the default when you use `New-ModuleManifest`. The specifics you *must* provide will differ based on your repository's requirements, but in general, we recommend at least the following be completed:

- `RootModule`. This is mandatory for the .psd1 to work, and it should point to the "main" .psm1 file of your module.
- `ModuleVersion`. This is generally mandatory, too, and is at the very least a very good idea.
- `GUID`. This is mandatory and generated automatically by `New-ModuleManifest`.
- `Author`. This could be your or your organization. You might even include an email address.
- `Description`. Don't be skimpy here. This is what people will read to decide if they want to use your module. Write a meaningful description that explains what your module does and why it's useful.

Take note of your author name and try to be consistent. You want to make it easy for people to find the other amazing tools you have published.

These are, incidentally, the minimums for publishing to PowerShell Gallery. we also recommend, in the strongest possible terms, that you specify the FunctionsToExport array, as well as VariablesToExport, CmdletsToExport, and AliasesToExport if those are applicable. Ours, as you'll see above, are set to `*`, which is a bad idea. In our specific example here, it makes sense, because our root module is empty - we aren't exporting anything; the module is just a kind of container for our sample code to live in. But in your case, the recommended best practice is to explicitly list function, alias, and variable (without the `$` sign) names which will achieve two benefits:

- Auto-discovery of your commands will be faster since PowerShell can just read the .psd1 rather than parsing the entire .psm1.
- Some repositories may be able to provide per-command search capabilities if you specify which commands your module offers.

Publishing to PowerShell Gallery

PowerShellGallery.com is a Microsoft-owned, public NuGet repository for released code. It can host PowerShell modules, DSC resources, and other artifacts. Start by heading over to PowerShellGallery.com and logging in or registering, using your Microsoft ID. Once signed in, click on your name. As part of your Gallery profile, you'll be able to request, view, and see your API key. This is a long hexadecimal identifier that you'll need when publishing code. Keep this secure.

With your API key in hand, it's literally as easy as going into PowerShell and running `Publish-Module` (which is part of the PowerShellGet module, which ships with PowerShell v5 and later and can be downloaded from PowerShellGallery.com for other PowerShell versions). Provide the name of your module, and your API key (via the `-NuGetApiKey` parameter), and you're good to go.

```
Publish-Module -path c:\scripts\MyAwesomeModule -nugetapikey $mykey
```

You may be prompted for additional information if it can't be found in your module manifest.

Be aware that publishing a module will include all files and folders in your module location. Hidden files and folders should be ignored but make sure you have cleaned up any scratch, test, or working files.

You'll likely receive a confirmation email from the Gallery, which may include a few PSScriptAnalyzer notifications. As we describe in the chapter on Analyzing Your Script, the Gallery automatically runs several PSScriptAnalyzer best practices rules on all submitted code, and you should try hard to confirm with these unless you have a specific reason not to.

So what's appropriate for PowerShell Gallery publication?

- Production-ready code. Don't submit untested, pre-release code unless you're doing so as part of a public beta test, and be sure to clearly indicate that the code isn't production-ready (for example, using `Write-Warning` to display a message when the module is loaded).
- Open-source code. Gallery code is, by implication, open-source; you should consider hosting your development code in a public OSS repository like GitHub, and only publish "released" code to the Gallery. Be sure not to include any proprietary information.
- Useful code. There are like thirty-seven million 7Zip modules in the Gallery. More are likely not needed. Try to publish code that provides unique value.

Items *can* be removed from Gallery if you change your mind, but Microsoft can't go out and delete whatever people may have already downloaded. Bear that in mind before contributing.

Publishing to Private Repositories or Galleries

Microsoft's vision is that organizations will host private and internal repositories. You may want to use a private repository merely for testing purposes. Ideally, these internal repositories will be based on Nuget. Setting up one of these is outside the scope of this book. However, you can set up a repository with a simple file share.

We've created a local file share and made sure that the admins group has write access.

```
New-SMBShare -name MyRepo -path c:\MyRepo -FullAccess Administrators `
-ReadAccess Everyone
```

Don't put any other files in this folder other than what you publish otherwise you will get errors when using `Find-Module`.

Next, you can register this file share as a repository.

```
Register-PSRepository -name MyRepo -SourceLocation c:\MyRepo `
-InstallationPolicy Trusted
```

We set the repository to be trusted because we know what is going in it, and we don't want to be bothered later when we try to install from it. If you forget, you can modify the repository later:

```
Set-PSRepository -Name MyRepo -InstallationPolicy Trusted
```

Now you can publish locally:

```
Publish-Module -Path c:\scripts\onering -Repository MyRepo
```

This local repository can be used just like the PowerShell gallery.

```
PS C:\> Find-Module -Repository MyRepo

Version    Name           Type      Repository      Description
-------    ----           ----      ----------      -----------
0.0.1.0    onering        Module    MyRepo          The module that ...
```

You can even install locally to verify everything works as expected.

```
PS C:\> Install-Module onering -Repository MyRepo
PS C:\> Get-Command -module OneRing -ListImported

CommandType     Name                                Version    Source
-----------     ----                                -------    ------
Function        Disable-Ring                        0.0.1.0    OneRing
Function        Enable-Ring                         0.0.1.0    OneRing
Function        Get-Ring                            0.0.1.0    OneRing
Function        Remove-Ring                         0.0.1.0    OneRing
Function        Set-Ring                            0.0.1.0    OneRing
```

We set this up locally as a proof of concept. It shouldn't take that much more work to set up a repository on a company file share. Just mind your permissions.

Your Turn

We aren't going to offer a real hands-on lab in this chapter, mainly because we think it's a bad idea to use a public repo like PowerShell Gallery as a "lab environment!" It's also non-trivial to set up your private repository, and if you go through that trouble, we think you'll want it to be in *production*, not in a lab so that you can benefit from that work.

That said, we do want to encourage you to sign into the PowerShell Gallery and create your API key, as we've described in this chapter. It's a first step toward getting ready to publish your code.

Let’s Review

We aren’t going to ask you to publish anything to the gallery. You may never need to publish or share your work. But let’s see if you picked up anything in this chapter.

1. The Microsoft PowerShell Gallery is based on what technology?
2. What important file is required to publish to the gallery that contains critical module metadata?
3. What should you publish to any repository?

Review Answers

Hopefully, you came up with answers like this:

1. Nuget
2. A module manifest.
3. Any unique project that offers value and is production-ready. You can publish your project that might be in beta or under development but that should be made clear to any potential consumer such as through version numbering.

Part 3: Controller Scripts and Visual Scripting

With your tools constructed and tested, it’s time to put them to work - and that means writing controller scripts. Not sure what that means? Keep reading. We’ll look at several kinds, from simple to complex, and look at a couple of other, unique ways in which you can put your tools to work.

Proxy Functions

In PowerShell, a *proxy function* is a specific kind of wrapper function. That is, it “wraps” around an existing command, usually with the intent of either:

- Removing functionality
- Hard-coding functionality and removing access to it
- Adding functionality

In some cases, a proxy command is meant to “replace” an existing command. This is done by giving the proxy the same name as the command it wraps; since the proxy gets loaded into the shell last, it’s the one that gets run when you run the command name.

There’s a way, using a fully-qualified command name, to regain access to the wrapped command, so proxy functions shouldn’t be seen as a security mechanism. They’re more of a functional convenience.

For Example

You’re probably familiar with PowerShell’s `ConvertTo-HTML` command. We’d like to make a version that “replaces” the existing command, providing full access to it but always injecting a particular CSS style sheet, so that the resulting HTML can be a bit prettier.

Creating the Proxy Base

PowerShell automates the first step, which is generating a “wrapper” that exactly duplicates whatever command you’re wrapping. Here’s how to use it (we’ll put our results into a Step1 subfolder in this chapter’s sample code):

```
$cmd = New-Object System.Management.Automation.CommandMetaData (Get-Command ConvertT\
o-HTML)
[System.Management.Automation.ProxyCommand]::Create($cmd) |
Out-File ConvertToHTMLProxy.ps1
```

Here’s the rather lengthy result (once again, apologies for the backslashes, which represent line-wrapping; it’s unavoidable in this instance, but the downloadable sample code won’t show them):

```
[CmdletBinding(DefaultParameterSetName='Page',
HelpUri='http://go.microsoft.com/fwlink/?LinkID=113290',
RemotingCapability='None')]
param(
    [Parameter(ValueFromPipeline=$true)]
    [PSObject]
    ${InputObject},

    [Parameter(Position=0)]
    [System.Object[]]
    ${Property},

    [Parameter(ParameterSetName='Page', Position=3)]
    [string[]]
    ${Body},

    [Parameter(ParameterSetName='Page', Position=1)]
    [string[]]
    ${Head},

    [Parameter(ParameterSetName='Page', Position=2)]
    [ValidateNotNullOrEmpty()]
    [string]
    ${Title},

    [ValidateNotNullOrEmpty()]
    [ValidateSet('Table','List')]
    [string]
    ${As},

    [Parameter(ParameterSetName='Page')]
    [Alias('cu','uri')]
    [ValidateNotNullOrEmpty()]
    [uri]
    ${CssUri},

    [Parameter(ParameterSetName='Fragment')]
    [ValidateNotNullOrEmpty()]
    [switch]
    ${Fragment},

    [ValidateNotNullOrEmpty()]
    [string[]]
    ${PostContent},

    [ValidateNotNullOrEmpty()]
    [string[]]
    ${PreContent})
```

```
begin
{
    try {
        $outBuffer = $null
        if ($PSBoundParameters.TryGetValue('OutBuffer', [ref]$outBuffer))
        {
            $PSBoundParameters['OutBuffer'] = 1
        }
        $wrappedCmd = $ExecutionContext.InvokeCommand.GetCommand('Microsoft.PowerShe\
ll.Utility\ConvertTo-Html',
        [System.Management.Automation.CommandTypes]::Cmdlet)
        $scriptCmd = {& $wrappedCmd @PSBoundParameters }
        $steppablePipeline = $scriptCmd.GetSteppablePipeline($myInvocation.CommandOr\
igin)
        $steppablePipeline.Begin($PSCmdlet)
    } catch {
        throw
    }
}

process
{
    try {
        $steppablePipeline.Process($_)
    } catch {
        throw
    }
}

end
{
    try {
        $steppablePipeline.End()
    } catch {
        throw
    }
}
<#

.ForwardHelpTargetName Microsoft.PowerShell.Utility\ConvertTo-Html
.ForwardHelpCategory Cmdlet

#>
```

This isn't wrapped in a function, so that's the first thing we'll do in the next step (which we'll put into a file in Step 2, so you can differentiate).

Modifying the Proxy

In addition to wrapping our proxy code in a function, we're going to play with the `-Head` parameter. We're not going to remove access to it; we want users to be able to pass content to `-Head`. We just want to intercept it, and add our stylesheet to it, before letting the underlying `ConvertTo-HTML` command take over. So we'll need to test and see if our command was even run with `-Head` or not, and if it was, grab that content and concatenate our own. The final result:

proxy-functions/Step2/ConvertToHTMLProxy.ps1

```
function NewConvertTo-HTML {
    [CmdletBinding(DefaultParameterSetName = 'Page',
        HelpUri = 'http://go.microsoft.com/fwlink/?LinkID=113290',
        RemotingCapability = 'None')]
    param(
        [Parameter(ValueFromPipeline = $true)]
        [PSObject]$InputObject,

        [Parameter(Position = 0)]
        [System.Object[]]$Property,

        [Parameter(ParameterSetName = 'Page', Position = 3)]
        [string[]]$Body,

        [Parameter(ParameterSetName = 'Page', Position = 1)]
        [string[]]$Head,

        [Parameter(ParameterSetName = 'Page', Position = 2)]
        [ValidateNotNullOrEmpty()]
        [string]$Title,

        [ValidateNotNullOrEmpty()]
        [ValidateSet('Table', 'List')]
        [string]$As,

        [Parameter(ParameterSetName = 'Page')]
        [Alias('cu', 'uri')]
        [ValidateNotNullOrEmpty()]
        [uri]$CssUri,

        [Parameter(ParameterSetName = 'Fragment')]
        [ValidateNotNullOrEmpty()]
        [switch]$Fragment,

        [ValidateNotNullOrEmpty()]
        [string[]]$PostContent,

        [ValidateNotNullOrEmpty()]
        [string[]]$PreContent

    begin {
        try {
            $outBuffer = $null
            if ($PSBoundParameters.TryGetValue('OutBuffer', [ref]$outBuffer)) {
                $PSBoundParameters['OutBuffer'] = 1
            }
            $wrappedCmd = $ExecutionContext.InvokeCommand.GetCommand('Microsoft.Powe\
rShell.Utility\ConvertTo-Html',
                [System.Management.Automation.CommandTypes]::Cmdlet)

            # create our css
            $css += @'
        <style>
        th { color:white; background-color: black;}
        body { font-family: Calibri; padding: 2px }
        </style>
'@

            # was -head specified?
            if ($PSBoundParameters.ContainsKey('head')) {
                $PSBoundParameters.head += $css
            }
            else {
                $PSBoundParameters += @{'Head' = $css }
            }

            $scriptCmd = { & $wrappedCmd @PSBoundParameters }
            $steppablePipeline = $scriptCmd.GetSteppablePipeline($myInvocation.Comma\
ndOrigin)
            $steppablePipeline.Begin($PSCmdlet)
        }
        catch {
            throw
        }
    }

    process {
        try {
            $steppablePipeline.Process($_)
        }
        catch {
            throw
        }
    }

    end {
        try {
            $steppablePipeline.End()
        }
        catch {
            throw
        }
    }
    <#

.ForwardHelpTargetName Microsoft.PowerShell.Utility\ConvertTo-Html
.ForwardHelpCategory Cmdlet

#>
}
```

Our changes begin at around line 50, with the `#create our css` comment.

Creating CSS

```
            # create our css
            $css += @'
        <style>
        th { color:white; background-color: black;}
        body { font-family: Calibri; padding: 2px }
        </style>
'@
```

Under that, we check to see if `-head` had been specified; if it was, we append our CSS to it. If not, we add a "head" parameter to `$PSBoundParameters`. Then we let the proxy function continue just as normal.

You may want to clean up references to the original version by deleting the HelpUri link in cmdletbinding as well as the forwarded help link at the end. Or if you have created your help documentation you can delete the forward links altogether. We also revised the parameter definitions to make them more like what we'd normally write in a function.

Adding or Removing Parameters

You're likely to run into occasions when you do want to add or remove a parameter. For example, a new parameter might simplify usage or unlock functionality; removing a parameter might enable you to hard-code a value that the user shouldn't be changing. The real key is the `$PSBoundParametersCollection`.

Adding a Parameter

Adding a parameter is as easy as declaring it in your proxy function's `Param()` block. Add whatever attributes you like, and you're good to go. You just want to *remove* the added parameter from `$PSBoundParameters` before the underlying command executes, since that command won't know what to do with your new parameter.

```
$PSBoundParameters.Remove('MyNewParam')
$scriptCmd = {& $wrappedCmd @PSBoundParameters }
```

Just remove it before that `$scriptCmd` line, and you're good to go.

Removing a Parameter

This is even easier - just delete the parameter from the `Param()` block! If you're removing a mandatory parameter, you'll need to internally provide a value with it. For example:

```
$PSBoundParameters += @{'RemovedParam'=$MyValue}
$scriptCmd = {& $wrappedCmd @PSBoundParameters }
```

This will re-connect the `-RemovedParam` parameter, feeding it whatever's in $MyValue, before running the underlying command.

Your Turn

Now it's your turn to create a proxy function.

Start Here

In this exercise, you'll be extending the `Export-CSV` command. However, you're not going to "overwrite" the existing command. Instead, you'll be creating a *new* command that uses `Export-CSV` under the hood.

Your Task

Create a proxy function named `Export-TDF`. This should be a wrapper around `Export-CSV`, and should not include a `-Delimiter` parameter. Instead, it should hard-code the delimiter to be a tab. Hint: you can specify a tab by putting a backtick, followed by the letter "t," inside double quotes.

Our Take

Here's what we came up with - also in the lab results folder in the downloadable code.

Export-TDF

```
function Export-TDF {
    [CmdletBinding(DefaultParameterSetName = 'Delimiter',
        SupportsShouldProcess,
        ConfirmImpact = 'Medium')]
    param(
        [Parameter(
            Mandatory,
            ValueFromPipeline,
            ValueFromPipelineByPropertyName
        )]
        [PSObject]$InputObject,

        [Parameter(Position = 0)]
        [ValidateNotNullOrEmpty()]
        [string]$Path,

        [Alias('PSPath')]
        [ValidateNotNullOrEmpty()]
        [string]$LiteralPath,

        [switch]$Force,

        [Alias('NoOverwrite')]
        [switch]$NoClobber,

        [ValidateSet('Unicode', 'UTF7', 'UTF8', 'ASCII', 'UTF32',
            'BigEndianUnicode', 'Default', 'OEM')]
        [string]$Encoding,

        [switch]$Append,

        [Parameter(ParameterSetName = 'UseCulture')]
        [switch]$UseCulture,

        [Alias('NTI')]
        [switch]$NoTypeInformation
    )

    begin {
        try {
            $outBuffer = $null
            if ($PSBoundParameters.TryGetValue('OutBuffer', [ref]$outBuffer)) {
                $PSBoundParameters['OutBuffer'] = 1
            }
            $wrappedCmd = $ExecutionContext.InvokeCommand.GetCommand('Microsoft.Powe\
rShell.Utility\Export-Csv',
                [System.Management.Automation.CommandTypes]::Cmdlet)
            $PSBoundParameters += @{'Delimiter' = "`t" }
            $scriptCmd = { & $wrappedCmd @PSBoundParameters }
            $steppablePipeline = $scriptCmd.GetSteppablePipeline($myInvocation.Comma\
ndOrigin)
            $steppablePipeline.Begin($PSCmdlet)
        }
        catch {
            throw
        }
    }
    process {
        try {
            $steppablePipeline.Process($_)
        }
        catch {
            throw
        }
    }
    end {
        try {
            $steppablePipeline.End()
        }
        catch {
            throw
        }
    }
} #close function
```

We just removed one parameter definition and added one line of code to hard-code the delimiter. We removed the {} around the parameter names and lined things up in the Param() block the way we would normally write code. We also removed the forwarded help links. We would still need to create new comment-based help for this command. Probably by copying a lot of the help from the original command.

Once you understand the concepts, you can use Jeff's Copy-Command[6] function from the PSScript-Tools module.

Let's Review

See if you can answer a couple of questions on proxy functions:

1. The boilerplate proxy function behaves exactly like what?
2. If you define an additional parameter in a proxy function, what must you do before the wrapped command is allowed to run?
3. If you delete a non-mandatory parameter definition in a proxy function, what must you do before the wrapped command is allowed to run?

Review Answers

Here are our answers:

1. The command it wraps.
2. Remove the new parameter from `$PSBoundParameters`.
3. You don't need to do anything since the wrapped command can run without the removed parameter.

[6]https://github.com/jdhitsolutions/PSScriptTools/blob/master/docs/Copy-Command.md

Part 4: The Data Connection

In this Part, we'll look at various kinds of structured data that you may need to work with from within PowerShell. It might mean grabbing something from some type of data source or perhaps putting something into a particular data type. We'll try and use some realistic examples that illustrate how to use these different structured data constructs and give you some tips for keeping out of trouble.

Working with SQL Server Data

We often see people struggle - really, really hard, in some cases - to store data in Microsoft Excel, using PowerShell to automate the process. This makes us sad. Programmatically, Excel is kind of a hunk of junk. Sure, it can make charts and graphs - but only with significant effort and a lot of delicacy. But, people say, "I already have it!" This also makes us sad, because for the very reasonable price of $FREE, you can have SQL Server (Express), and in fact, you probably have some flavor of SQL Server on your network that you could use. But why?

- SQL Server is easy to use from PowerShell code. Literally a handful of lines, and you're done.
- SQL Server Reporting Services (also free in the Express edition) can turn SQL Server data into *gorgeous* reports with charts and graphs - and can automate the production and delivery of those reports with zero effort from you.
- SQL Server is something that many computers can connect to at once, meaning you can write scripts that run on servers, letting those servers update their data in SQL Server. This is faster than a script that reaches out to query many servers to update a spreadsheet.

We don't know how to better evangelize using SQL Server for data storage over Microsoft Excel.

SQL Server Terminology and Facts

Let's quickly get some terminology and basic facts out of the way.

- SQL Server is a service that runs on a *server*. Part of what you'll need to know, to use it, is the server's name. A single machine (physical or VM) can run multiple *instances* of SQL Server, so if you need to connect to an instance other than the default, you'll need the instance name also. The naming pattern is `SERVER\INSTANCE`.
- A SQL Server instance can host one or more *databases*. You will usually want a database for each major data storage purpose. For example, you might ask a DBA to create an "Administration Data" database on one of your SQL Server computers, giving you a place to store stuff.
- Databases have a *recovery mode* option. Without getting into a lot of details, you can use the "Simple" recovery mode (configurable in SQL Server Management Studio by right-clicking the database and accessing its Options page) if your data isn't irreplaceable and you don't want to mess with maintaining the database's log files. For anything more complex, either take a DBA to lunch or read Don's *Learn SQL Server Administration in a Month of Lunches*.
- Databases contain *tables*, each of which is analogous to an Excel worksheet.
- Tables consist of *rows* (entities) and *columns* (fields), which correspond to the rows and columns of an Excel sheet.
- Columns have a *data type*, which determines the kind of data they can store, like text (NVARCHAR), dates (DATETIME), or numbers (INT). The data type also determines the data ranges. For example, NVARCHAR(10) can hold 10 characters; NVARCHAR(MAX) has no limit. INT can store smaller values than BIGINT, and bigger values than TINYINT.

- SQL Server defaults to Windows Authentication mode, which means the domain user account running your scripts must have permission to connect to the server (a *login*), and permission to use your database (a *database user*). This is the safest means of authentication as it doesn't require passwords to be kept in your script. If running a script as a scheduled task, the task can be set to "run as" a domain user with the necessary permissions.

Just seven little things to know, and you're good to go.

Even if you are the only person who will ever interact with stored data, you are still better off installing SQL Server Express (did we mention it is free?) instead of relying on Excel.

Connecting to the Server and Database

You'll need a *connection string* to connect to a SQL Server computer/instance, and connect to a specific database. If you're not using Windows Authentication, the connection string can also contain a clear-text username and password, which is a horrible practice. We use ConnectionStrings.com[7] to look up connection string syntax, but here's the one you'll use a lot:

Use `Server=SERVER\INSTANCE;Database=DATABASE;Trusted_Connected=True;` to connect to a given server and instance (omit the `\INSTANCE` if you're connecting to the default instance) and database. Note that SQL Server Express usually installs, by default, as an instance named `SQLEXPRESS`. You can run `Get-Service` in PowerShell to see any running instances on a computer, and the service name will include the instance name (or just `MSSQLSERVER` if it's the default).

With that in mind, it's simple to code up a connection:

```
$conn = New-Object -Type System.Data.SqlClient.SqlConnection
$conn.ConnectionString = 'Server=SQL1;Database=MyDB;Trusted_Connection=True;'
$conn.Open()
```

You can leave the connection open for your entire script; be sure to run `$conn.Close()` when you're done, though. It's not a tragedy to not close the connection; when your script ends, the connection object will vanish, and SQL Server will automatically close the connection a bit later. But if you're using a server that's pretty busy, the DBA is going to get in your face about leaving the connection open. And, if you run your script multiple times in a short period of time, you'll create a new connection each time rather than re-using the same one. The DBAs will definitely notice this and get agitated.

You do not need to have any SQL Server software installed locally for these steps as they are relying on out-of-the-box bits from the .NET Framework. And even if you are working with a local SQL installation, you should still follow SQL Server best practices.

Writing a Query

The next thing you need to do is retrieve, insert, update, or remove some data. This is done by writing queries in the Transact-SQL (T-SQL) language, which corresponds with the ANSI SQL standard, meaning most queries look the same on most database servers. There's a great free online SQL tutorial[8] if you need

[7]http://connectionstrings.com
[8]http://www.w3schools.com/sql/

one, but we'll get you started with the basics.

To do this, you'll need to know the *table* and *column names* from your database. SQL Server Management Studio is a good way to discover these.

For the following sections, we're going to focus on *query syntax*, and then give you an example of how we might build that query in PowerShell. Once your query is in a variable, it's easy enough to run it - and we'll cover how to do that in a bit. Also, we're not going to be providing exhaustive coverage of SQL syntax; we're covering the basics. There are plenty of resources, including the aforementioned online tutorial, if you need to dig deeper.

Adding Data

Adding data is done by using an `INSERT` query. The basic syntax looks like this:

```
INSERT INTO <tablename>
    (Column1, Column2, Column3)
    VALUES (Value1, Value2, Value3)
```

So you'll need to know the name of the table you're adding data to, and you'll need to know the column names. You also need to know a bit about how the table was defined. For example, if a "Name" column is marked as mandatory (or "NOT NULL") in the table design, then you *must* list that column and provide a value for it. Sometimes, a table may define a default value for a column, in which case you can leave the column out if you're okay with that default value. Similarly, a table can permit a given column to be empty (NULL), and you can omit that column from your list if you don't want to provide a value.

Whatever order you list the columns in, your values must be in the same order. You're not forced to use the column order that the table defines; you can list them in any order.

Numeric values aren't delimited in T-SQL. String values are delimited in single quotes; any single quotes *within* a string value (like "O'Leary") must be doubled ("O''Leary") or your query will fail. Dates are treated as strings and are delimited with single quotes.

It's dangerous to build queries from user-entered data. Doing so opens your code to a kind of attack called *SQL Injection*. We're assuming that you plan to retrieve things like system data, which shouldn't be nefarious, rather than accepting input from users. The safer way to deal with user-entered data is to create a stored procedure to enter the data, but that's well beyond the scope of this book.

We might build a query in PowerShell like this:

```
$ComputerName = "SERVER2"
$OSVersion = "Win2012R2"
$query = "INSERT INTO OSVersion (ComputerName,OS) VALUES('$ComputerName','$OSVersion\
')"
```

This assumes a table named OSVersion, with columns named ComputerName and OS. Notice that we've put the entire query into double quotes, allowing us to just drop variables into the `VALUES` list.

We always put our query in a variable, because that makes it easy to output the query text by using `Write-Verbose`. That's a great way to debug queries that aren't working since you get to see the actual query text with all the variables filled in.

Removing Data

A `DELETE` query is used to delete rows from a table, and it is almost always accompanied by a `WHERE` clause so that you don't delete *all the rows*. Be very careful, as there's no such thing as an "UNDO" query!

```
DELETE FROM <tablename> WHERE <criteria>
```

So, suppose we're getting ready to insert a new row into our table, which will list the OS version of a given computer. We don't know if that computer is already listed in the table, so we're going to just delete any existing rows before adding our new one. Our `DELETE` query might look like this:

```
$query = "DELETE FROM OSVersions WHERE ComputerName = '$ComputerName'"
```

There's no error generated if you attempt to delete rows that don't exist.

Changing Data

An `UPDATE` query is used to change an existing row, and is accompanied by a `SET` clause with the changes, and a `WHERE` clause to identify the rows you want to change.

```
UPDATE <tablename>
   SET <column> = <value>, <column> = <value>
   WHERE <criteria>
```

For example:

```
$query = "UPDATE DiskSpaceTracking `
          SET FreeSpaceOnSysDrive = $FreeSpace `
          WHERE ComputerName = '$ComputerName'"
```

We'd ordinarily do that all on one line; we've broken it up here just to make it fit more easily in the book. This assumes that `$FreeSpace` contains a numeric figure and that `$ComputerName` contains a computer name.

In SQL Server, column names aren't case-sensitive.

Retrieving Data

Finally, the big daddy of queries is the `SELECT` query. This is the only one that returns data (although the other three will return the number of rows they affected). This is also the most complex query in the language, so we're only tackling the basics.

```
SELECT <column>,<column>
       FROM <tablename>
       WHERE <criteria>
       ORDER BY <column>
```

The `WHERE` and `ORDER BY` clauses are optional, and we'll come to them in a moment.

Beginning with the core `SELECT`, you follow with a list of columns you want to retrieve. While the language permits you to use `*` to return all columns, this is a poor practice. For one, it performs slower than a column list. For another, it makes your code harder to read. So stick with listing the columns you want.

The `FROM` clause lists the table name. This can get a ton more complex if you start doing multi-table joins, but we're not getting into that in this book.

A `WHERE` clause can be used to limit the number of rows returned, and an `ORDER BY` clause can be used to sort the results on a given column. Sorting is ascending by default, or you can specify descending. For example:

```
$query = "SELECT DiskSpace,DateChecked `
          FROM DiskSpaceTracking `
          WHERE ComputerName = '$ComputerName' `
          ORDER BY DateChecked DESC"
```

Creating Tables Programmatically

It's also possible to write a *data definition language* (DDL) query that creates tables. The four queries we've covered up to this point are *data manipulation language* (DML) queries. The ANSI specification doesn't cover DDL as much as DML, meaning DDL queries differ a lot between server brands. We'll continue to focus on T-SQL for SQL Server; we just wanted you to be aware that you won't be able to re-use this syntax on other products without some tweaking.

```
CREATE TABLE <tablename> (
    <column> <type>,
    <column> <type>
)
```

You list each column name, and for each, provide a datatype. In SQL Server, you'll commonly use:

- Int or BigInt for integers
- VarChar(x) or VarChar(MAX) for string data; "x" determines the maximum length of the field while "MAX" indicates a binary large object (BLOB) field that can contain any amount of text.
- DateTime

You want to use the smallest data type possible to store the data you anticipate putting into the table, because oversized columns can cause a lot of wasted disk space.

Running a Query

You've got two potential types of queries: ones that return data (`SELECT`) and ones that don't (pretty much everything else). Running them starts the same:

```
$command = New-Object -Type System.Data.SqlClient.SqlCommand
$command.Connection = $conn
$command.CommandText = $query
```

This assumes `$conn` is an open connection object, and that `$query` has your T-SQL query. How you run the command depends on your query. For queries that don't return results:

```
$command.ExecuteNonQuery()
```

That *can* produce a return object, which you can pipe to `Out-Null` if you don't want to see it. For queries that produce results:

```
$reader = $command.ExecuteReader()
```

This generates a *DataReader* object, which gives you access to your queried data. The trick with these is that they're *forward-only*, meaning you can read a row, and then move on to the next row - but you can't go back to read a previous row. Think of it as an Excel spreadsheet, in a way. Your cursor starts on the first row of data, and you can see all the columns. When you press the down arrow, your cursor moves down a row, and you can only see *that* row. You can't ever press the up arrow, though - you can only keep going down the rows.

You'll usually read through the rows using a `While` loop:

```
while ($reader.read()) {
  #do something with the data
}
```

The `Read()` method will advance to the next row (you start "above" the first row, so executing `Read()` the first time doesn't "skip" any data), and return True if there's a row after that.

To retrieve a column, inside the `While` loop, you run `GetValue()`, and provide the *column ordinal number* of the column you want. This is why it's such a good idea to explicitly list your columns in your `SELECT` query; you'll know which column is in what position. The first column you listed in your query will be `0`, the one after that `1`, and so on.

So here's a full-fledged example:

```
$conn = New-Object -Type System.Data.SqlClient.SqlConnection
$conn.ConnectionString = 'Server=SQL1;Database=MyDB;Trusted_Connection=True;'
$conn.Open()

$query = "SELECT ComputerName,DiskSpace,DateTaken FROM DiskTracking"

$command = New-Object -Type System.Data.SqlClient.SqlCommand
$command.Connection = $conn
$command.CommandText = $query
$reader = $command.ExecuteReader()
```

```
while ($reader.read()) {
    [PSCustomObject]@{'ComputerName' = $reader.GetValue(0)
                        'DiskSpace' = $reader.GetValue(1)
                        'DateTaken' = $reader.GetValue(2)
                       }
}

$conn.Close()
```

This snippet will produce objects, one object for each row in the table, and with each object having three properties that correspond to three of the table columns.

If by chance you don't remember your column positions, you can use something like this to auto-discover the column number.

```
while ($reader.read()) {
    [PSCustomObject]@{
    'ComputerName' = $reader.GetValue($reader.GetOrdinal("computername"))
    'DiskSpace' = $reader.GetValue($reader.GetOrdinal("diskspace"))
    'DateTaken' = $reader.GetValue($reader.GetOrdinal("datetaken"))
    }
}
```

Regardless of the approach, we'd usually wrap this in a `Get-` function, so that we could just run the function and get objects as output. Or a corresponding `Set-`, `Update-` or `Remove-` function depending on your SQL query.

Invoke-SqlCmd

If you by chance have installed a local instance of SQL Server Express, you will also have a set of SQL-related PowerShell commands and a SQLSERVER PSDrive. We aren't going to cover them as this isn't a SQL Server book. But you will want to take advantage of `Invoke-SqlCmd`.

Instead of dealing with the .NET Framework to create a connection, command and query, you can simply invoke the query.

```
Invoke-SqlCmd "Select Computername,DiskSpace,DateTaken from DiskTracking" `
-Database MyDB
```

You can use any of the query types we've shown you in this chapter. One potential downside to this approach in your toolmaking is that this will only work locally, or where the SQL Server modules have been installed. Also, there is a bit of a lag while the module is initially loaded.

Thinking About Tool Design Patterns

If you've written a tool that retrieves or creates some data that you intend to put into SQL Server, then you're on the right track. The next step would be a tool that inserts the data into SQL Server (Export-Something), and perhaps a tool to read the data back out (Import-Something). This approach maintains a good design pattern of each tool doing one thing, and doing it well, and lets you create tools that can be composed in a pipeline to perform complex tasks. You can read a bit more about that approach, and even get a "generic" module for piping data in and out of SQL Server databases in Ditch Excel: Making Historical & Trend Reports in PowerShell[9], a free ebook.

Let's Review

Because we don't want to assume that you have access to a SQL Server computer, we aren't going to present a hands-on experience in this chapter. However, we do encourage you to try and answer these questions:

1. How to you prevent `DELETE` from wiping out a table?
2. What method do you use to execute an `INSERT` query?
3. What method reads a single database row from a reader object?

Review Answers

Here are our answers:

1. Specify a `WHERE` clause to limit the deleted rows.
2. The `ExecuteNonQuery()` method.
3. The `Read()` method.

[9]https://www.gitbook.com/book/devopscollective/ditch-excel-making-historical-trend-reports-in-po

Part 5: Seriously Advanced Toolmaking

In this Part of the book, we'll dive into some deep, "extra" topics. These are all things we're pretty sure you should know, but that you might not use right away, especially if you are an apprentice toolmaker. This Part isn't constructed in a storyline, so you can just pick and choose the bits you think you'll need or you find interesting.

Measuring Tool Performance

We PowerShell geeks will often get into late-night, at-the-pub arguments about which bits of PowerShell perform best under certain circumstances. You'll hear arguments like, "The `ForEach-Object` cmdlet is slower because its script block has to be parsed each time" or, "Storing all those objects in a variable will make everything take longer because of how arrays are managed." At the end of the day, if performance is *important* to you, this is the chapter for you.

Is Performance Important

Well, maybe. *Why* is performance important to you? Look, if you've written a command that will have to reboot a dozen computers, then we're going to be splitting hairs all night about which way is faster or slower. It won't matter. But if you're writing code that needs to manipulate *thousands* of objects, or *tens of thousands* or more, then a minute performance gain per object will add up quickly. The point is, before you sweat this stuff, know that tweaking PowerShell for millisecond performance gains isn't useful unless there are a *lot* of milliseconds to be saved.

Measure What's Important

But if performance *is* important, then you need to *measure it.* Forget every possible argument for or against any given technique, and *measure it.* And, as you measure, make sure you're measuring to the scale that your command will eventually run. That is, don't test a command with five objects when the plan is to run against five hundred thousand. Pressures like memory, disk I/O, network, and CPU won't interact in meaningful ways at a small scale, and so small-scale measurements won't prove out as you scale up your workload.

Think of it this way: just because a one-lane road can carry 100 cars an hour, doesn't mean a 4-lane road can carry 400 an hour. It's a different situation, with different dynamics. So *measure* against the workload you plan to run.

You'll perform that measurement using the `Measure-Command` cmdlet. Feed it your command, script, pipeline, or whatever, and it'll run it - and spit out how long it took it to complete. Take this short script as an example (this is test.ps1 in the sample files):

measuring-tool-performance/test.ps1

```
Write-Host 'Round 1' -ForegroundColor Green
Measure-Command -Expression {
    Get-Service |
    ForEach-Object { $_.Name }
}

Write-Host 'Round 2' -ForegroundColor Yellow
Measure-Command -Expression {
    Get-Service |
    Select-Object Name
}

Write-Host 'Round 3' -ForegroundColor Cyan
Measure-Command -Expression {
    ForEach ($service in (Get-Service)) {
        $service.name
    }
}
```

This does the same thing in different ways. Let's run that to see what happens:

```
Round 1

Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 0
Milliseconds      : 148
Ticks             : 1486572
TotalDays         : 1.72056944444444E-06
TotalHours        : 4.12936666666667E-05
TotalMinutes      : 0.00247762
TotalSeconds      : 0.1486572
TotalMilliseconds : 148.6572

Round 2
Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 0
Milliseconds      : 37
Ticks             : 379826
TotalDays         : 4.39613425925926E-07
TotalHours        : 1.05507222222222E-05
TotalMinutes      : 0.000633043333333333
TotalSeconds      : 0.0379826
```

```
TotalMilliseconds : 37.9826

Round 3
Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 0
Milliseconds      : 38
Ticks             : 389199
TotalDays         : 4.50461805555556E-07
TotalHours        : 1.08110833333333E-05
TotalMinutes      : 0.000648665
TotalSeconds      : 0.0389199
TotalMilliseconds : 38.9199
```

There's a significant penalty, time-wise, for the first method, while the second two are almost tied. Neat, right?

Be Careful!

The thing to remember is that whatever you're measuring ***will run*** and ***will do stuff***. This isn't a "safe test mode" or something. So you may need to modify your script a bit so that you can test it without actually performing the task at hand. Of course, that can backfire, too. You can imagine that a tool designed to modify Active Directory might run a *lot* faster if it wasn't actually communicating with Active Directory, and so your measurement wouldn't be real-world or useful.

One thing to watch for when running `Measure-Command` is that a single test isn't necessarily absolute proof. There could be any number of factors that might influence the result. Sometimes it helps to run the test several times. Jeff wrote a command called `Test-Expression` in the PSScriptTools module that allows you to run a test multiple times, giving you (hopefully) a more meaningful result. There's even a GUI version.

Factors Affecting Performance

There are a bunch of things that can impact a tool's performance.

Collections and arrays can get very slow if they get really big and you keep adding objects to them one at a time. This slowdown has to do with how .NET allocates and manages memory for these things.

Anything storing a lot of data in memory can be affected if .NET has to stop and garbage-collect variables that are no longer referenced. Generally, you want to try and manage reasonable amounts of data in memory, not great huge wodges of 60GB text files.

Compiling script blocks - as `ForEach-Object` requires - can incur a performance penalty. It's not always avoidable, but it isn't the fastest operation on the planet in some cases.

Wasting memory can result in disk paging, which can slow things down. For example, in the below fragment, we're still storing a potentially and unnecessarily huge list of users in $users long past the point where we're done with it.

```
$users = Get-ADUser -filter *
$filtered = $users | Where { $_.Department -like '*IT*' }
$final = $filtered | Select Name,Cn
$final | Out-File names.txt
```

It'd be better do to this entirely without variables, and let the filtering happen on the domain controller:

```
Get-ADUser -filter "Department -like '*IT*'" |
Select Name,Cn |
Out-File names.txt
```

Now we're getting massively less data back from Active Directory, and storing none of it in persistent variables. Or to put it more precisely, this is an example of the benefits of early filtering.

Here's the problem - We often see beginners write a command like this:

```
Get-CimInstance win32_service -computername server01 |
where state -eq 'running'
```

This may not seem like a big deal but imagine the CIM command was going to return 1000 objects. With the approach we just showed, the first command has to complete and send all 1000 objects, in this case across the wire and *then* the results are figured. Compared to letting `Get-CimInstance` do the filtering in place - on the server - and then only sending the filtered results back.

```
Get-CimInstance win32_service -computername server01 -filter "state = 'running'"
```

There's one other feature you should take advantage of when using `Get-CimInstance` and not many people do. Let's say you are using code like this:

```
Get-CimInstance win32_service -computername $computers-filter "state = 'running'" |
Select-Object -property Name,StartMode,StartName,ProcessID,SystemName
```

The `$Computers` variable is a list of computer names. This pattern is pretty common. Get something and then select the things that matter to you. However, the remote server is assembling a complete Win32_Server instance with all the properties. But you are throwing most of them away. The better approach is to limit what `Get-CimInstance` will send back.

```
$CimParams=@{
  ClassName    = 'win32_service'
  ComputerName = $computers
  Filter       = "state = 'running'"
  Property     = 'Name','StartMode','StartName','ProcessID','SystemName'
}
Get-CimInstance  @CimParams |
Select-Object -property Name,StartMode,StartName,
ProcessID,@{Name="Computername";Expression={$_.SystemName}}
```

PowerShell will want to display the results using its default formatting. You'll most likely use `Select-Object` or create your custom object. Regardless, this approach *should* run slightly faster. It may be small, but it can add up. You'll appreciate this when you are querying `500` servers.

Always look for ways to limit or filter as early in your command as possible. Take advantage of parameters like Filter, Include, Exclude, ID, and Name.

Key Take-Away

You should get used to using `Measure-Command` to test your code, especially if there are several ways you could go. We'll look at other performance-related concepts in the Scripting at Scale chapter. But for now your key take-away should be that good coding practices can go a long way toward avoiding performance problems!

Part 6: Pester

We provided an introduction to Pester earlier in this book, but now we’d like to dig deep. Pester is a pretty important part of the PowerShell universe these days, and if you’re going to be a professional-grade PowerShell toolmaker, you should make Pester a big part of your world.

Why Pester Matters

In the world of DevOps and automation, it's crucial that your code - you know, the thing that *enables* your automation - be reliable. In the past, you'd accomplish reliability, or attempt to, by manually testing your code. The problems with manual testing are legion:

- You're likely to be inconsistent. That is, you might forget to test some things sometimes, which opens the door to devastating bugs.
- You're going to spend a lot of time, if you're doing it right (and the time commitment is what makes most people not "do it right" in the first place).
- You end up wasting time setting up "test harnesses" to safely test your code, amongst other "supporting" tasks.

This is where Pester comes in. Simply put, it's a testing automation tool for PowerShell code, as we explained earlier in this book.

- Pester is consistent. It tests the same things, every time, so you never "miss" anything. And, if you discover a new bug that you weren't testing for, you can add it to your automated tests to make sure that bug never "sneaks by" again.
- Pester can be automated, so it takes none of your time to perform tests.
- Pester integrates well with continual integration tools, like Visual Studio Team Services (VSTS), Jenkins, Team City, and so on, so that spinning up test environments and running tests can also be completely automated.

The vision goes something like this:

1. You check in your latest PowerShell code to a code repository, like Git or VSTS. That code includes Pester tests.
2. A miracle occurs.
3. Your tested code is either rejected due to failed tests (and you're notified), or your code appears in a production repository, such as a NuGet repository where it can be deployed via PowerShellGet.

The "miracle" here is some kind of automated workflow. VSTS, for example, might spin up a test environment, load your code into it, and run your Pester tests against your code. We're not going to cover how to make the miracle work, as it's not technically a PowerShell thing per se, and because there are so many combinations of options you could choose. We *are* going to focus on how to write those Pester tests, though.

The big thing here is that *you need to be writing testable code*, a concept we'll devote a specific chapter to. But if you're looking for the short answer to, "What is testable code?" It's basically "follow the advice we've been giving you in this book." Write concise, self-contained, single-task functions to do *everything*.

The other thing you'll want to quickly embrace is *to write your Pester tests immediately*, if not actually *in advance* of your code (something we'll discuss more in the chapter on test-driven development). This is going to require an act of will for most PowerShell folks because we tend to want to just dive in and start experimenting, rather than worrying about writing tests. But the difference between the adults and the babies, here, is that the adults do the right thing because they know it's the right thing to do. Having tests available *from the outset of your project* is how you reap the advantages of Pester, and indeed of PowerShell more generally.

So that's why Pester is important. We don't think anyone should write *any* code unless they're also going to write automated tests for it.

It's also important to understand what Pester *does*, and this gets a bit squishy. First, it's worth considering the different kinds of testing you might want to perform in your life. Here are a few testing patterns but by no means all:

- *Unit testing* is nothing more than making sure your code *runs*. You want to make sure it behaves properly when passed various combinations of parameters, for example, and that its internal logic behaves as expected. You usually try to test the code in isolation, meaning you prevent it from making any permanent changes to systems, databases, and so on. You're also *just testing your code*, not anybody else's. If your code internally runs `Get-CimInstance`, then you *prevent it from doing so*, since `Get-CimInstance` isn't *your* code. Unit testing is what Pester is all about, and it contains functionality to help you achieve all of the above. The idea is to isolate your code as much as possible to make testing more practical, and to make debugging easier.
- *Integration testing* is a bit more far-reaching. It's designed to test your code *running in conjunction* with whatever other code is involved. This is where you'd go ahead and let internal `Get-CimInstance` calls run correctly, so make sure your code operates well *when integrated* with other code. Integration testing is more "for real" than unit testing, and typically runs in something close to a production environment, rather than in isolation.
- *Infrastructure validation* can be thought of as an extension of integration testing. Because so much of our PowerShell code is about modifying computer systems, such as building VMs or deploying software, infrastructure validation runs our code and then *reaches out to check the results*. Pester can also be used for this kind of testing, and we'll get into it more later in this book.

All of this is important to understand because it helps you better understand what a Pester test looks like. If you've written a function that's little more than a wrapper around `ConvertTo-HTML`, for example, then your Pester tests aren't going to be very complex, because you probably didn't write much code. You're not trying to make sure `ConvertTo-HTML` itself works because that's not your code, so it's not your problem in a *unit test*. Because *so much* of our PowerShell code is leveraging other people's code, our own Pester tests are often simpler and easier to grasp.

Part 7: PowerShell 7 Scripting

PowerShell 7, which is essentially today's PowerShell and runs cross-platform, offers a few new features that you can use in your scripting projects. Of course, these will only work when the code is run on a PowerShell 7 platform. The chapters in this section will introduce you to some new operators, variables, and parameters.

Cross Platform Scripting

When Microsoft moved PowerShell to the open-source world a few years ago, it opened up an entirely new world of scripting possibilities. Microsoft wants you to be able to manage anything from anywhere on whatever platform you need. Of course, you'll want to build scripts and tools to use in this new world. And even if you aren't in a situation now that requires cross-platform management, you never know what's gonna happen tomorrow. And frankly, some of the things you need to keep in mind when it comes to cross-platform scripting can make you a better scripter for Windows PowerShell.

What do we mean when we say "cross-platform"? We mean that you can develop a tool that can be used on any PowerShell-supported platform and ideally, can manage or work with any platform. Instead of writing one tool to run on Linux and another to run on Windows, you write one tool that can run on both. Don't build multiple tools to manage remote servers depending on the target operating system. Write one command that you can use regardless of the OS. To meet these requirements, there are a few things we want you to keep in mind. We are under no illusion that everything you want to do cross-platform will work. It won't. There *will* be situations where a true cross-platform solution simply doesn't work.

Many of the things we're covering in this chapter, technically also apply to PowerShell Core which was the PowerShell 6. x branch.

Know Your OS

When you are building something that can potentially be used cross-platform you need to think about the operating system where your code will be running. As cool as PowerShell 7 is, not every feature or command is supported on every operating system. You don't need to be an expert-level Linux engineer, but you do need to understand some basics. For example, Linux doesn't have the same concept of services as we understand them in the Windows world, so there is no `Get-Service` command in PowerShell 7 on non-Windows systems. Likewise, Linux doesn't have WMI or CIM so you won't find `Get-CimInstance`. Nor could you use `Get-CimInstance` to target a remote Linux machine. The OS doesn't support these features so they aren't available.

What this means is that you need to know what OS your code is running on and there are some things you can do to make your life easier.

State Your Requirements

First off, if you are writing a PowerShell function that uses PowerShell 7 features, such as the new ternary operator, you need to make sure the person running your script is using PowerShell 7. This means adding a #requires statement at the top of your file:

```
#requires -version 7.0
```

Honestly, this is something you should have been doing all along but you **absolutely** need it now. Because PowerShell 7 features and commands can also change between versions, you should be as specific as possible.

```
#requires -version 7.4
```

If you need specific modules that might be Windows-specific, state that requirement as well.

```
#requires -modules CIMCmdlets
```

If someone runs this on a Linux box they'll get an error like:

```
The script 'Get-Miracle.ps1' cannot be run because the following modules that are sp\
ecified by the "#requires" statements of the script are missing: CimCmdlets.
```

If you are building a module, in the manifest you can use this setting to set requirements.

```
# Supported PSEditions
CompatiblePSEditions = @('Desktop','Core')
```

`Desktop` means Windows PowerShell. `Core` means the open-source and cross-platform version of PowerShell, *regardless of operating system*. This can be a little tricky. Because even though `Get-CimInstance` doesn't work on a Mac in PowerShell 7, it will work just fine on a Windows desktop running PowerShell 7. Still, this high-level compatibility setting should be set. Delete whatever isn't supported.

Testing Variables

Even though the `Core` setting is potentially problematic, there are some new automatic variables you can use in your code, to validate and test.

- `PSEdition` - On PowerShell 7, this will have a value of `Core`. On Windows PowerShell, it will return `Desktop`. Look familiar?
- `IsWindows` - A boolean value indicating if you are running Windows. Requires PowerShell 7.
- `IsLinux` - A boolean value indicating if you are running Linux. Requires PowerShell 7.
- `IsMac` - A boolean value indicating if you are running MacOS. Requires PowerShell 7.
- `IsCoreCLR` - A boolean value indicating if you are running .NET Core which most likely means you are running PowerShell 7. This variable isn't defined in Windows PowerShell.

And of course, don't forget `$PSVersionTable`.

```
PS C:\> $PSVersionTable

Name                           Value
----                           -----
PSVersion                      7.4.2
PSEdition                      Core
GitCommitId                    7.4.2
OS                             Microsoft Windows 10.0.22631
Platform                       Win32NT
PSCompatibleVersions           {1.0, 2.0, 3.0, 4.0â€¦}
PSRemotingProtocolVersion      2.3
SerializationVersion           1.1.0.1
WSManStackVersion              3.0
```

Which can vary by platform.

```
PS /home/jeff> $PSVersionTable

Name                           Value
----                           -----
PSVersion                      7.4.2
PSEdition                      Core
GitCommitId                    7.4.2
OS                             Ubuntu 22.04.4 LTS
Platform                       Unix
PSCompatibleVersions           {1.0, 2.0, 3.0, 4.0â€¦}
PSRemotingProtocolVersion      2.3
SerializationVersion           1.1.0.1
WSManStackVersion              3.0
```

It is also different on Windows PowerShell.

```
PS C:\> $PSVersionTable

Name                           Value
----                           -----
PSVersion                      5.1.22621.2506
PSEdition                      Desktop
PSCompatibleVersions           {1.0, 2.0, 3.0, 4.0...}
BuildVersion                   10.0.22621.2506
CLRVersion                     4.0.30319.42000
WSManStackVersion              3.0
PSRemotingProtocolVersion      2.3
SerializationVersion           1.1.0.1
```

These are all potentially useful items you can use to build `If` constructs or even dynamic parameters.

Environment Variables

One of the major challenges in cross-platform scripting is breaking out of the old way of doing things. Here's a great example using a common parameter definition you might see in Windows PowerShell.

```
[string[]]$Computername = $env:ComputerName
```

This sets the `%COMPUTERNAME%` environment variable as the default value for `$Computername`. This will work just fine on Windows PowerShell and even PowerShell 7 running on Windows. But non-Windows platforms have different environment variables and this will fail. Instead, you can use a .NET code snippet to get the same information.

```
[environment]::MachineName
```

This is also true with `$env:UserName` which can be replaced with `[environment]::UserName`. If you are used to referencing environment variables, you'll need to add a step to verify they are defined or begin using .NET alternatives.

Paths

PowerShell has always never cared about the direction of slashes in paths. You can run `Test-Path c:\windows` or `Test-Path c:/windows`. This is even true on non-Windows systems. You can use `test-path /etc/ssh` or `test-path \etc\ssh`. But you really should be careful. Many of you probably have used or seen code like this:

```
$file = "$foo\child\file.dat"
```

There's a good chance it will work cross-platform. But the better approach, which you should be using anyway, is to use the path cmdlets like `Join-Path`. Here's a sample.

Export-Data

```
Function Export-Data {
  [cmdletbinding()]
  Param(
    [ValidateScript({ Test-Path $_ })]
    [string]$Path = '.'
  )

  $time = Get-Date -Format FileDate
  $file = "$($time)_export.json"
  $ExportPath = Join-Path -Path (Convert-Path $Path) -ChildPath $file
  Write-Verbose "Exporting data to $exportPath"

  # code ...
}
```

Instead of worrying about which direction to put the slashes, let PowerShell do it for you.

If you need it, you can use `[System.IO.Path]::DirectorySeparatorChar` to get the directory separator character. For the %PATH% variable, you can use `[System.IO.Path]::PathSeparator`. Let's say you want to split the %PATH% environment variable into an array of locations. A simple cross-platform approach would be `$env:PATH -split [System.IO.Path]::PathSeparator`. Don't forget that Linux is case-sensitive.

Watch Your Aliases

You've most likely heard us go on and on about the downside of using aliases in your PowerShell scripts. Use them all you want interactively at a prompt but in written code, use full cmdlet names. Now you need to.

In the old days, we'd happily write a command like this:

```
Get-Process | sort handles -descending
```

And it would work just fine on Windows, even under PowerShell 7. But not Linux.

```
PS /home/jeff> Get-Process | sort handles -descending
/usr/bin/sort: invalid option -- 'e'
Try '/usr/bin/sort --help' for more information.
```

What happened? In PowerShell 7 on Linux, any alias that could resolve to a native command has been removed. So there is no `sort` alias for `Sort-Object`. PowerShell thinks you want to run the native `sort` command. If you've been in the habit of using Linux aliases like `ps` or `ls`, you'll need to get over it. You need to start writing expressions like:

```
Get-Process | Sort-Object handles -descending
```

Now there's no mistaking what you want to do and it will run everywhere.

If you are using VS Code, and we don't know why you're not, it is very easy to convert aliases. We get it. You have muscle memory and you find it easier to write code using aliases. Fine. When you are finished and before you release your code into the world, open the command palette (`Ctrl+Shift+P`) and run 'Powershell: Expand alias". Done.

Using Culture

If you are writing code that relies on culture information, such as from `Get-Culture`, note that in PowerShell 7, you won't get the same information. In Windows PowerShell, we are used to getting output like this:

```
PS C:\> Get-Culture

LCID             Name             DisplayName
----             ----             -----------
1033             en-US            English (United States)
```

On *Windows* platforms running PowerShell 7, this will still work as expected. But not on non-Windows platforms. You'll get output like this:

```
PS /home/jeff> Get-Culture

LCID             Name             DisplayName
----             ----             -----------
127                               Invariant Language (Invariant Country)

PS /home/jeff> Get-UICulture

LCID             Name             DisplayName
----             ----             -----------
127                               Invariant Language (Invariant Country)
```

You still have access to culture-specific information from the .NET Framework.

```
PS /home/jeff> [System.Globalization.CultureInfo]::GetCultureInfo("en-gb")

LCID             Name             DisplayName
----             ----             -----------
2057             en-GB            English (United Kingdom)
```

But PowerShell can't detect the culture information on non-Windows platforms.

You might be able to use native commands like *locale* to get the information you need.

You'll need to be aware of this and test your code accordingly if you are relying on detected culture information.

Leverage Remoting

One of the most anticipated features of PowerShell 7 is the use of `ssh` for remoting. Non-Windows systems won't support the WSMan protocol which means no remoting the way you used to do it. But you may still want to use remoting in your toolmaking. In our opinion, leveraging remoting is a smart idea. Going to PowerShell 7 just means a little more work on your part.

We're not going to dive into the mechanics of getting SSH remoting to work in PowerShell 7. That's all up to you and your organization. We're simply going to assume it already works.

One relatively easy approach you can use is parameter sets. Define one parameter set for a computer name and another for PSSession objects. PowerShell already follows this model. You can too. Here's a proof of concept function.

Get-RemoteData

```
Function Get-RemoteData {
  [cmdletbinding(DefaultParameterSetName = 'computer')]
  Param(
    [Parameter(
      Position = 0,
      Mandatory,
      ValueFromPipeline,
      ParameterSetName = 'computer'
    )]
    [Alias('cn')]
    [string[]]$Computername,
    [Parameter(ParameterSetName = 'computer')]
    [alias('RunAs')]
    [PSCredential]$Credential,
    [Parameter(ValueFromPipeline, ParameterSetName = 'session')]
    [System.Management.Automation.Runspaces.PSSession]$Session
  )
  Begin {
    $sb = { "Getting remote data from $([environment]::MachineName) [$PSEdition]" }
    $PSBoundParameters.Add('Scriptblock', $sb)
  }
  Process {
    Invoke-Command @PSBoundParameters
  }
  End {}
}
```

The person running the function can either pass a computername with an optional credential, or a previously created PSSession object. They may have existing connections to a mix of platforms, some using SSH connections. Now they can run one command that works for all.

```
PS C:\> Get-PSSession | Get-RemoteData
Getting remote data from SRV2 [Desktop]
Getting remote data from FRED [Core]
Getting remote data from SRV1 [Desktop]
```

In this example, FRED is a Linux server running Fedora. Start simple like this.

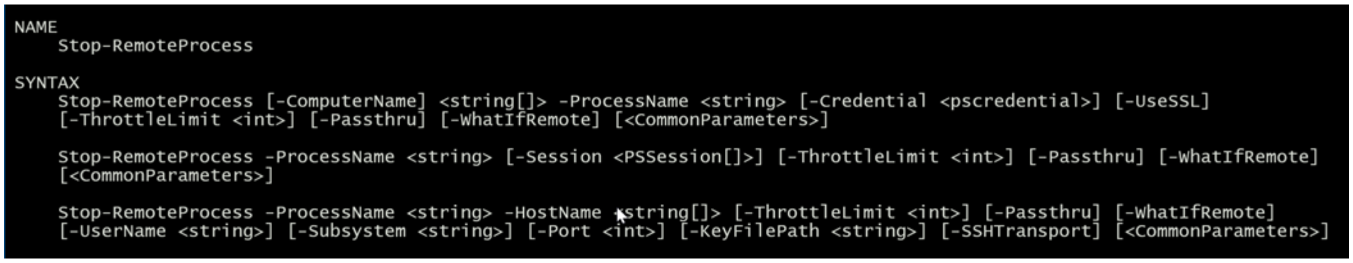
But when you are ready, you can get very creative. Here's a function that defines dynamic parameters if the user is running PowerShell 7.

cross-platform-scripting/Stop-RemoteProcess.ps1

```
#requires -version 5.1

Function Stop-RemoteProcess {
    [cmdletbinding(DefaultParameterSetName = 'computer')]
    Param(
        [Parameter(
            ParameterSetName = 'computer',
            Mandatory,
            Position = 0,
            ValueFromPipeline,
            ValueFromPipelineByPropertyName,
            HelpMessage = 'Enter the name of a computer to query.'
        )]
        [ValidateNotNullOrEmpty()]
        [Alias('cn')]
        [string[]]$ComputerName,
        [Parameter(
            ParameterSetName = 'computer',
            HelpMessage = 'Enter a credential object or username.'
        )]
        [Alias('RunAs')]
        [PSCredential]$Credential,
        [Parameter(ParameterSetName = 'computer')]
        [switch]$UseSSL,

        [Parameter(
            ParameterSetName = 'session',
            ValueFromPipeline
        )]
        [ValidateNotNullOrEmpty()]
        [System.Management.Automation.Runspaces.PSSession[]]$Session,

        [ValidateScript( { $_ -ge 0 })]
        [int32]$ThrottleLimit = 32,

        [Parameter(Mandatory, HelpMessage = 'Specify the process to stop.')]
        [ValidateNotNullOrEmpty()]
        [string]$ProcessName,

        [Parameter(HelpMessage = 'Write the stopped process to the pipeline')]
        [switch]$Passthru,

        [Parameter(HelpMessage = 'Run the remote command with -WhatIf')]
        [switch]$WhatIfRemote
    )
    DynamicParam {
        #Add an SSH dynamic parameter if in PowerShell 7
        if ($IsCoreCLR) {
            $paramDictionary = New-Object -Type System.Management.Automation.Runtime\
DefinedParameterDictionary

            #a CSV file with dynamic parameters to create
            #this approach doesn't take any type of parameter validation into account
            $data = @'
Name,Type,Mandatory,Default,Help
HostName,string[],1,,"Enter the remote host name."
UserName,string,0,,"Enter the remote user name."
Subsystem,string,0,"powershell","The name of the ssh subsystem. The default is power\
shell."
Port,int32,0,,"Enter an alternate SSH port"
KeyFilePath,string,0,,"Specify a key file path used by SSH to authenticate the user"
SSHTransport,switch,0,,"Use SSH to connect."
'@

            $data | ConvertFrom-Csv | ForEach-Object -Begin { } -Process {
                $attributes = New-Object System.Management.Automation.ParameterAttri\
bute
                $attributes.Mandatory = ([int]$_.mandatory) -as [bool]
                $attributes.HelpMessage = $_.Help
                $attributes.ParameterSetName = 'SSH'
                $attributeCollection = New-Object -Type System.Collections.ObjectMod\
el.Collection[System.Attribute]
                $attributeCollection.Add($attributes)
                $dynParam = New-Object -Type System.Management.Automation.RuntimeDef\
inedParameter($_.name, $($_.type -as [type]), $attributeCollection)
                $dynParam.Value = $_.Default
                $paramDictionary.Add($_.name, $dynParam)
            } -End {
                return $paramDictionary
            }
        }
    } #dynamic param

    Begin {
        $start = Get-Date
        #the first verbose message uses a pseudo timespan to reflect the idea we're \
just starting
        Write-Verbose "[00:00:00.0000000 BEGIN  ] Starting $($MyInvocation.MyCommand\
)"

        #a script block to be run remotely
        Write-Verbose "[$(New-TimeSpan -Start $start) BEGIN  ] Defining the scriptbl\
ock to be run remotely"
```

```
        $sb = {
            param([string]$ProcessName, [bool]$Passthru, [string]$VerbPref = 'Silent\
lyContinue', [bool]$WhatPref)

            $VerbosePreference = $VerbPref
            $WhatIfPreference = $WhatPref

            Try {
                Write-Verbose "[$(New-TimeSpan -Start $using:start) REMOTE ] Getting\
 Process $ProcessName on $([System.Environment]::MachineName)"
                $Processes = Get-Process -Name $ProcessName -ErrorAction stop
                Try {
                    Write-Verbose "[$(New-TimeSpan -Start $using:start) REMOTE ] Sto\
pping $($Processes.count) Processes on $([System.Environment]::MachineName)"
                    $Processes | Stop-Process -ErrorAction Stop -PassThru:$Passthru
                }
                Catch {
                    Write-Warning "[$(New-TimeSpan -Start $using:start) REMOTE ] Fai\
led to stop Process $ProcessName on $([System.Environment]::MachineName). $($_.Excep\
tion.message)."
                }
            }
            Catch {
                Write-Verbose "[$(New-TimeSpan -Start $using:start) REMOTE ] Process\
 $ProcessName not found on $([System.Environment]::MachineName)"
            }

        } #scriptblock

        #parameters to splat to Invoke-Command
        Write-Verbose "[$(New-TimeSpan -Start $start) BEGIN  ] Defining parameters f\
or Invoke-Command"

        #remove my parameters from PSBoundParameters because they can't be used with\
 New-PSSession
        $myParams = 'ProcessName', 'WhatIfRemote', 'passthru'
        foreach ($my in $myParams) {
            if ($PSBoundParameters.ContainsKey($my)) {
                [void]($PSBoundParameters.remove($my))
            }
        }

        $icmParams = @{
            Scriptblock      = $sb
            ArgumentList     = @($ProcessName, $Passthru, $VerbosePreference, $WhatI\
fRemote)
            HideComputerName = $False
            ThrottleLimit    = $ThrottleLimit
            ErrorAction      = 'Stop'
            Session          = $null
        }

        #initialize an array to hold session objects
        [System.Management.Automation.Runspaces.PSSession[]]$All = @()
        If ($Credential.username) {
            Write-Verbose "[$(New-TimeSpan -Start $start) BEGIN  ] Using alternate c\
redential for $($credential.username)"
        }
    } #begin

    Process {
        Write-Verbose "[$(New-TimeSpan -Start $start) PROCESS] Detected parameter se\
t $($PSCmdlet.ParameterSetName)."

        $remotes = @()
        if ($PSCmdlet.ParameterSetName -match 'computer|ssh') {
            if ($PSCmdlet.ParameterSetName -eq 'ssh') {
                $remotes += $PSBoundParameters.HostName
                $param = 'HostName'
            }
            else {
                $remotes += $PSBoundParameters.ComputerName
                $param = 'ComputerName'
            }

            foreach ($remote in $remotes) {
                $PSBoundParameters[$param] = $remote
                $PSBoundParameters['ErrorAction'] = 'Stop'
                Try {
                    #create a session one at a time to better handle errors
                    Write-Verbose "[$(New-TimeSpan -Start $start) PROCESS] Creating \
a temporary PSSession to $remote"
                    #save each created session to $tmp so it can be removed at the e\
nd
                    $all += New-PSSession @PSBoundParameters -OutVariable +tmp
                } #Try
                Catch {
                    #TODO: Decide what you want to do when the new session fails
                    Write-Warning "Failed to create session to $remote. $($_.Excepti\
on.Message)."
                    #Write-Error $_
                } #catch
            } #foreach remote
        }
        Else {
            #only add open sessions
            foreach ($sess in $session) {
                if ($sess.state -eq 'opened') {
                    Write-Verbose "[$(New-TimeSpan -Start $start) PROCESS] Using ses\
sion for $($sess.ComputerName.ToUpper())"
                    $all += $sess
                } #if open
            } #foreach session
        } #else sessions
    } #process

    End {
        $icmParams['session'] = $all
        Try {
            Write-Verbose "[$(New-TimeSpan -Start $start) END    ] Querying $($all.c\
ount) computers"

            Invoke-Command @icmParams | ForEach-Object {
                #TODO: PROCESS RESULTS FROM EACH REMOTE CONNECTION IF NECESSARY
                $_
            } #foreach result
        } #try
        Catch {
            Write-Error $_
        } #catch

        if ($tmp) {
            Write-Verbose "[$(New-TimeSpan -Start $start) END    ] Removing $($tmp.c\
ount) temporary PSSessions"
            $tmp | Remove-PSSession
        }
        Write-Verbose "[$(New-TimeSpan -Start $start) END    ] Ending $($MyInvocatio\
n.MyCommand)"
    } #end
} #close function
```

This function essentially follows the same model as the previous example. But this one creates several dynamic parameters if PowerShell 7 is detected. Notice we're using one of the new variables. When the user runs help in PowerShell 7, they'll get the new parameters.



```
NAME
    Stop-RemoteProcess

SYNTAX
    Stop-RemoteProcess [-ComputerName] <string[]> -ProcessName <string> [-Credential <pscredential>] [-UseSSL]
    [-ThrottleLimit <int>] [-Passthru] [-WhatIfRemote] [<CommonParameters>]

    Stop-RemoteProcess -ProcessName <string> [-Session <PSSession[]>] [-ThrottleLimit <int>] [-Passthru] [-WhatIfRemote]
    [<CommonParameters>]

    Stop-RemoteProcess -ProcessName <string> -HostName <string[]> [-ThrottleLimit <int>] [-Passthru] [-WhatIfRemote]
    [-UserName <string>] [-Subsystem <string>] [-Port <int>] [-KeyFilePath <string>] [-SSHTransport] [<CommonParameters>]
```

Stop-RemoteProcess Help

The last parameter set is the dynamic one. Instead of having to write several versions of the command, you can write a single function. Check out https://jdhitsolutions.com/blog/powershell/7458/a-powershell-remote-function-framework/[10] for a bit more detail on this function.

Custom Module Manifests

The last cross-platform scripting feature to consider is a custom module manifest. In the code downloads for this chapter, you'll find a demo module called CrossDemo. The module has several commands, some of which will only work in PowerShell 7. The goal is to only export the commands (and aliases) that are supported. Here's how.

Normally, a psd1 is static and can't contain code. But there is an exception for a module manifest. You can use a simple If statement to tell PowerShell what functions to export.

```
FunctionsToExport = if ($PSEdition -eq 'desktop') {
    'Export-Data','Get-DiskFree'
    }
    else {
    'Export-Data','Get-DiskFree','Get-Status','Get-RemoteData'
    }
...
AliasesToExport = if ($PSEdition -eq 'desktop') {
    'df'
    }
    else {
    'df','gst'
    }
```

When the module is imported on Windows PowerShell, only 2 functions and 1 alias are exported. PowerShell 7 systems get everything.

Even so, you still may need to fine-tune platform requirements. For example, if you look at our sample code for the `Get-DiskFree` function, you'll see code like this:

```
 if ($IsWindows -OR $PSEdition -eq 'desktop') {
        $Drive = (Get-Item $Path).Root -replace "\\"
        Write-Verbose "Getting disk information for $drive in $As"
        ...
    } #if Windows
    else {
        Write-Warning 'This command requires a Windows platform.'
    }
```

Because the function uses `Get-CimInstance` it requires a Windows platform. It will work in PowerShell 7 on Windows but not Linux. So even though we're exporting the function for PowerShell 7, it will only run on Windows. You'll need to keep things like this in mind. Will it work in PowerShell 7 and what are the platform dependencies?

[10]https://jdhitsolutions.com/blog/powershell/7458/a-powershell-remote-function-framework/

Not every sample function in our demo module uses this logic. Feel free to update the code as an exercise.

We'll be honest with you and say that a lot of community-accepted best practices for cross-platform scripting are still being developed. But if you use a little common sense and follow the best practices that *are* accepted, you shouldn't have too much trouble.