FOREWORD WRITTEN BY ORIN THOMAS

MODERN IT AUTOMATION WITH POWERSHELL

FIRST EDITION

EDITED BY

MICHAEL ZANATTA
BILL KINDLE

STEVEN JUDD
NICHOLAS BISSELL

Modern IT Automation with PowerShell

Modern Automation with PowerShell

The DevOps Collective, Inc. and Michael Zanatta

This book is for sale at http://leanpub.com/modernautomationwithpowershell

This version was published on 2022-09-20

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2022 The DevOps Collective, Inc. All Rights Reserved

Tweet This Book!

Please help The DevOps Collective, Inc. and Michael Zanatta by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I’m supporting the future of our field and just bought a copy of The #pwshconftextbook First Edition!

The suggested hashtag for this book is #pwshconftextbook.

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

#pwshconftextbook

Contents

Foreword

By Orin Thomas

Organizations adopt information technology to solve a set of problems. The problems could be as simple as "how do we keep track of customer orders?" or more complicated ones involving the analysis of data to determine patterns that might provide some new insight that leads to a business advantage. Organizations will choose a technology not just because they think it is fun and cool, but because they can use it to solve a problem that they really need to be solved to accomplish an organizational objective. The creators of information technologies often have a set of problems in mind when building those technologies. It might be "how do I simplify the process of creating and presenting slides at conferences" or "how can we better automate administrative tasks on Microsoft platforms". Every system, application, or programming language has a particular set of tasks it was designed to do very well because the people that created it needed to scratch an itch that the currently available tools did not adequately address.

Over time creators and developers add and refine features to their products or tools because those features to allow users of the technology to solve additional or more complex problems that are considered important. But the key is understanding that to the creator and developer there is a set of problems that are "in scope" that they envisage their project solving and a lot that are beyond that scope that it was never intended to address. William Gibson, author of one of the earliest and most important cyberpunk novels said "The street finds its own uses for things". One of the interpretations of this is that users often find uses for products that go way beyond what the original developers of that product envisaged it being used for. Good products are useful for things that the original creators never imagined. No product can do everything, but with a bit of creativity, many products can do things that are a complete surprise to other users and even the original creators of that project.

Technical communities are groups of people that are enthusiastic about specific systems, applications, technologies, and languages. These communities spend a great amount of time not only sharing how to be more proficient in doing something that the technology that they are interested in was designed to do, but also sharing all of the amazing things that the technology does that no one could have imagined. Jeffrey Snover often remarks how surprised he is at all the ingenious things people find that they can do with PowerShell that he never conceived of it being used for.

And that's what this book is about—sharing with the reader a collection of fascinating things that you can do with PowerShell. Not only things that you already do that you might be able to do in a much more efficient or elegant way, but a collection of tasks that you can do with PowerShell that exceed what you and perhaps even Jeffrey Snover conceived the language was possible of accomplishing.

Contributors

This section includes the names and biographies of the authors and editors of this project in alphabetical order.

Alain Tanguy

Role: Author

Alain has been an avid PowerShell user throughout his IT career, automating and solving many problems. Now working as an IT Engineer, he is using his knowledge to answer IT Infrastructure Challenges. He enjoys pizza, pixel art, funky music, and naps. Alain is reachable on Twitter at @Alain__Tanguy[1] and LinkedIn[2].

Allen Chin

Role: Linguistic Editor

Allen first learned and made use of PowerShell in 2018 while working as technical support to maintain and improve existing scripts. A few years later, he is now an application development analyst and the main contact for applications, reports, and automation.

Amy Zanatta

Role: Cover Artist

Amy is a Video Editor, Motion Graphics Designer and Personal Stylist, working at the Nine Network Australia. Amy posts regularly on her YouTube Channel StyleWithinGrace[3], with Australian fashion ideas and inspirations. You can follow her on Instagram[4] and Twitter @StyleGraceAmy[5].

[1] https://twitter.com/Alain__Tanguy

[2] https://www.linkedin.com/in/alaintanguy

[3] https://www.youtube.com/channel/UCCF1px3YSkNKB6F0HtySI9g

[4] https://www.instagram.com/stylewithingrace/

[5] https://www.twitter.com/StyleGraceAmy/

Bill Kindle

Role: Senior Editor

Husband to a wonderful woman that he doesn't deserve, and the father of two adorable children. Bill is a former career Systems Administrator turned Cyber Security Engineer currently working for Corsica Technologies[6]. Bill was an author for The PowerShell Conference Book Volume 2 and an editor for Volume 3. His role focuses on automation engineering and supporting a Security Operations Center. Bill has a passion for helping others in IT and occasionally does presentations for @FortWayneVMUG[7]. You can find some of Bill's work at AdamTheAutomator[8] and TechSnips LLC[9].

C.J. Zuk

Role: Linguistic Editor

C.J. Zuk is a current college student and works as a federal government contractor in the Washington, D.C. Metropolitan Area. She began to use PowerShell in high school as her preferred scripting language because of its cross-platform compatibility. C.J. likes to spend her time with her fiancee, cats, and volunteering at her synagogue. You can reach her at cj.zuk@outlook.com and on LinkedIn[10].

Chad Miars

Role: Linguistic Editor

Chad's formal training is in mechanical engineering, but his current role is Director of Business Ops at Ascent Inc.[11]. He enjoys using PowerShell to connect and automate all parts of the business. You can find Chad on LinkedIn[12].

Christian Coventry

Role: Linguistic Editor

Christian is a recent graduate, and it was his tertiary studies that introduced him to PowerShell. He currently works as a Technical Officer with the Queensland Department of Education[13]. Christian was an editor for The PowerShell Conference Book Volume 3. You can find him on LinkedIn[14].

[6] https://corsicatech.com
[7] https://twitter.com/FortWayneVMUG
[8] https://adamtheautomator.com
[9] https://techsnips.io
[10] https://www.linkedin.com/in/cj-zuk/
[11] https://ascenthvac.com
[12] https://www.linkedin.com/in/chad-miars-6489a2122/
[13] https://education.qld.gov.au/
[14] https://au.linkedin.com/in/christian-coventry-9a0031157

Felipe Binotto

Role: Author

Felipe is a specialist in Microsoft technologies and has worked in various roles in the IT industry for the last 13 years. Currently, he works as a Senior Customer Engineer for the Customer Success Unit at Microsoft Australia. In this role, his focus is on Azure infrastructure, security, and automation. Felipe has contributed to various PowerShell forums and Microsoft Docs, and he currently blogs at Azure Gear[15]. You can follow him on Twitter at @felipebinotto[16] or on LinkedIn[17].

Greg Onstot

Role: Author

Husband, Father, Contributor to the PowerShell Conference Book Vol. 2. Greg has been working in IT Infrastructure and Cybersecurity Engineering for over 20 years. PowerShell is an integral part of automating that work.

James Petty

Role: DevOps Collective Liaison

James currently serves as the CEO of the DevOps Collective Inc., a nonprofit working in the technology education space. He helps manage a $1M+ annual budget that includes multiple conferences and PowerShell Saturdays events across the US. The nonprofit focuses on PowerShell, automation, and DevOps and runs numerous free online resources, including PowerShell.org. He is also a co-organizer and co-founder of the Chattanooga PowerShell UserGroup (established in September 2016). James is also a recipient of the Microsoft MVP award in Cloud and Datacenter Management. He currently lives in the beautiful Chattanooga, Tennessee area with his amazing wife. James' passion lies with automation using PowerShell and all things related to Windows Server. He has almost a decade of experience as an infrastructure admin for a large enterprise, helping manage thousands of users and machines. He knows a broad range of products, including patch management, Active Directory, Group Policy, and the Windows Server operating system.

[15] https://azuregear.com
[16] https://twitter.com/felipebinotto
[17] https://www.linkedin.com/in/felipebinotto/

Joe Houghes

Role: Technical Editor

Husband, Father, Community Geek. Joe Houghes is a co-leader of @ATXPowerShell[18] and @AustinVMUG[19] user groups in Texas and a member of the @vBrownBag[20] crew. He is currently a Solutions Architect for Veeam, focused on automation & integration. Joe spends most of his time working within VMware environments when he is not active in planning or hosting community events. You can find Joe on Twitter, @jhoughes[21], or his blog[22].

John Hermes

Role: Linguistic Editor

John is an agile software developer and systems engineer with a career focus on security and resilience. He frequently develops PowerShell modules supporting datacenter management, legacy systems, and cloud service integration. He is also an unabashed Unix greybeard who still enjoys learning new things and rarely updates his social media. John resides with his extremely patient and loving wife near Dayton, Ohio.

Jordan Borean

Role: Technical Editor

Jordan is a Software Engineer at Red Hat[23], working on the Windows integrations for Ansible. He originally worked on Java-based programs for a large company but felt the draw to open source software and has been an avid contributor since. Jordan mostly focuses on Python and PowerShell-based languages, and he is committed to trying to bridge the Windows and Linux worlds and make it easier for them to work with each other. Some projects that he works on are pypsrp[24], smbprotocol[25], pypsexec[26], and more recently pyspnego[27]. When finding some free time, Jordan blogs on Blogging for Logging[28] that cover technologies like PowerShell, Ansible, Windows protocols, and anything else that takes his fancy. You can usually get in contact with him on the PowerShell Discord server[29], or various IRC Freenode channels like `#ansible`, `#Powershell`, `#packer-tool`, and others.

[18] https://twitter.com/ATXPowerShell
[19] https://twitter.com/AustinVMUG
[20] https://twitter.com/vbrownbag
[21] https://twitter.com/jhoughes
[22] https://www.fullstackgeek.net/
[23] https://www.redhat.com/en
[24] https://github.com/jborean93/pypsrp
[25] https://github.com/jborean93/smbprotocol
[26] https://github.com/jborean93/pypsexec
[27] https://github.com/jborean93/pyspnego
[28] https://www.bloggingforlogging.com/
[29] https://aka.ms/psdiscord

Kevin Laux

Role: Author and Quality Assurance Editor

Kevin is a manager for an orchestration platform team. He is passionate about PowerShell and has been leading training classes for his colleagues since the release of PowerShell v3. Kevin also serves as co-leader of the Research Triangle PowerShell User Group[30]. In addition to PowerShell, he is always tinkering with new technology in his home lab and trying to learn everything he can. You can follow him on Twitter, @rsrychro[31], and GitHub[32].

Kieran Jacobsen

Role: Author and Technical Editor

Kieran Jacobsen (he/him) combines his passion for business process automation, systems integration, and cybersecurity to help organizations rapidly grow and evolve. Kieran's involvement in the technology community has seen him present at Microsoft's Ignite the Tour, NDC Sydney, and CrikeyCon. Kieran is well known for his security-focused presentations that blend real-world examples with storytelling. Microsoft has recognized Kieran's contributions to the community by awarding him with the Most Valuable Professional since 2017. Kieran is also a member of the GitKraken Ambassador Program. Kieran lives in Melbourne, Australia, with his husband and Burmese cat. In his spare time, Kieran enjoys computer games, Dungeons & Dragons, board games, and Melbourne's amazing food culture.

Kirill Nikolaev

Role: Author and Technical Editor

Kirill has more than 15 years of experience in IT, with specialization in Windows Server infrastructure, virtualization, information security, and automation. He began using PowerShell almost immediately after its release in 2006 and has been ever since. He is currently Head of the Windows Administration Team at Fozzy.com[33], an honest hosting provider, where he continues to give back to the community by sharing automation solutions through their GitHub account[34]. You can follow him on Twitter, @exchange12rocks[35], or subscribe to his technical blog[36].

[30] https://rtpsug.com
[31] https://twitter.com/rsrychro
[32] https://github.com/KevinLaux
[33] https://fozzy.com
[34] https://github.com/FozzyHosting
[35] https://twitter.com/exchange12rocks
[36] https://exchange12rocks.org

Martha Clancy

Role: Linguistic Editor

Martha came to PowerShell and DevOps by way of database administration and is passionate about using code and automation to help everyone do their jobs more easily. You can find Martha on Twitter, @marclancy[37], or her blog[38].

Matt Corr

Role: Author

Husband, father, passionate about automation and process improvement. Matt has over 20 years of IT industry experience and is currently working as a DevOps Solution Specialist for MOQdigital[39]. He is very passionate about PowerShell and is the go-to person in his teams for anything script or automation-related. He has experience with many build and deployment tools, such as Azure DevOps, Octopus Deploy, TeamCity, Terraform, and PowerShell. You can find Matt on Twitter (@mattcorr[40]), LinkedIn[41] or his blog[42].

Michael B. Smith

Role: Quality Assurance Editor

Michael is an IT professional with over 35 years of experience in IT. Michael began using PowerShell during the Exchange 2007 Server beta and has been deeply into scripting with PowerShell ever since. He is a 13-time recipient of the Microsoft MVP award in Exchange Server. He has written many articles about Exchange, Active Directory, PowerShell, Windows Server, and Azure topics; and is passionate about presenting/training as well. You can find Michael on Twitter @EssentialExch[43], at his blog The Essential Exchange[44], and in the Facebook group Azure Support[45].

[37] https://twitter.com/marclancy
[38] https://marthaclancy.com
[39] https://www.moqdigital.com.au
[40] https://www.twitter.com/mattcorr
[41] https://www.linkedin.com/in/mattcorr/
[42] https://www.intrepidintegration.com
[43] https://twitter.com/essentialexch
[44] https://www.essential.exchange
[45] https://www.facebook.com/groups/AzureSupport

Michael Lotter

Role: Author

Michael has nearly a decade of system and network administration experience between South Africa and the US, from small shops to large enterprises, with a focus on automation and cybersecurity. He currently works in the finance sector as a systems engineer, where he uses PowerShell to automate complex processes. In addition to automation through PowerShell, he enjoys leveraging Azure and Intune to reduce organizations' on-premises footprint while maintaining high availability.

Michael Zanatta

Role: Author and Editor-in-Chief

Michael is a Microsoft MVP (Cloud and Datacenter Management), PowerShell SME, Speaker, Advocate, and Streamer, contracting as a PowerShell Developer for the Australian Federal Government. Michael has contributed to the PowerShell Conference Book Volume 2 and Volume 3, first as an author and stand-in editor on Volume 2 and then as the Senior Editor on Volume 3. You can follow him on Twitter, @PowerShellMich1[46], or LinkedIn[47]. Michael is a co-founder of the Brisbane Infrastructure DevOps User Group[48] YouTube channel[49] and author of his Livestream on Twitch[50].

Nicholas Bissell

Role: Author and Senior Editor

Though Nicholas's formal background is in research chemistry, he has over ten years of programming experience. In his spare time, he is a freelance software developer and mentor. Nicholas has previously worked as a sound engineer and a game developer. He is passionate about automation, open-source software, and the PowerShell community. You can find him on GitHub[51], StackOverflow[52], or Reddit[53].

[46] https://twitter.com/PowerShellMich1
[47] https://www.linkedin.com/in/michael-zanatta-61670258/
[48] https://www.meetup.com/Brisbane-PowerShell-User-Group
[49] https://www.youtube.com/channel/UCQfLvFYohCCm_gTPEUfaAbw
[50] https://www.twitch.tv/PowerShellMichael
[51] https://github.com/TheFreeman193
[52] https://stackoverflow.com/users/12959131/thefreeman193
[53] https://www.reddit.com/user/thefreeman193

Rob Derickson

Role: Quality Assurance Editor

Rob is an IT professional with 19 years in the field. Since 2008, PowerShell and the PowerShell community have been instrumental in his career success. You can find him on the PowerShell Discord server[54] and tweeting nothing on Twitter @RobDerickson[55].

Steven Judd

Role: Senior Editor

Steven Judd is a 25+ year IT Pro and currently a Windows Systems Engineer at Meta Platforms Inc.[56] with an emphasis on Enterprise Messaging and Digital Loss Prevention. He has been using PowerShell since 2010. He was an author and editor on The PowerShell Conference Book Volume 3[57], has co-developed a custom training program for PowerShell, and is an occasional presenter at PowerShell user groups. He loves to help people learn and recognize the value of automation. He spends his free time learning more about PowerShell, digital security, and cloud technologies, along with creating and telling Dad jokes[58]. You can find him hanging out on the PowerShell Discord Server[59] bridge channel, taking care of his family, running marathons, playing the cello, plus a handful of other hobbies he can't seem to quit. Please follow him on Twitter, @stevenjudd[60], read his blog[61], and review, use, and improve his code on GitHub[62].

Wes Stahler

Role: Technical Editor

Wes Stahler has over 25 years of Information Technology experience as a Developer, Systems Administrator, and Manager of an Identity and Access Management team. He enjoys evangelizing PowerShell's merits and has presented numerous times nationally at the Microsoft Health Users Group and locally for the Central Ohio PowerShell Users Group. He strives to automate within Exchange and Active Directory and advocates on the "Power of the Shell." Available on Twitter at @stahler[63].

[54] https://aka.ms/psdiscord
[55] https://twitter.com/RobDerickson
[56] https://www.linkedin.com/in/stevenjudd/
[57] https://leanpub.com/psconfbook3
[58] https://www.youtube.com/watch?v=BZZM6i8AE1Y
[59] https://aka.ms/psdiscord
[60] https://twitter.com/stevenjudd/
[61] https://blog.stevenjudd.com/
[62] https://github.com/stevenjudd
[63] https://twitter.com/stahler

Acknowledgements

This book was made possible by a multitude of people, not just the initial team of editors and the writers, but their family, friends, mentors, peers, and—most of all—you, the readers.

By reading this book, you're helping to make sure that our field expands and grows, creating opportunities for folks who otherwise might never see them. That's incredible and needs no qualifiers.

This project owes itself to the PowerShell community and everyone who gave it their time, energy, and money.

Disclaimer

All code examples shown in this book have been tested by each chapter author and every effort has been made to ensure that they're error-free. However, since every environment is different, the examples should be run in a non-production environment and should be thoroughly tested before being used in a production environment. It's recommended that you use a non-production or lab environment to thoroughly test code examples used throughout this book.

All data and information provided in this book is for educational purposes only. The editors make no representations as to the accuracy, completeness, currentness, suitability, or validity of any information in this book and won't be liable for any errors, omissions, or delays in this information or any losses, injuries, or damages arising from its display or use. All information is provided on an as-is basis.

This disclaimer is provided simply because someone, somewhere will ignore this disclaimer and if they do experience problems or a "resume generating event," they have no one to blame but themselves. Don't be that person!

Introduction

By Michael Zanatta

Hello Reader!

Modern Automation with PowerShell was an initiative between Steven Judd and myself to create a textbook as a love-letter *for the community by the community*. We wanted to provide an intermediary resource, different in style from the previous PowerShell Conference Books. We wanted to focus on a deeper understanding of the inner workings of PowerShell, share best practices and tips, and have this book serve as a study resource and lesson guide. All royalties for this book will go to the "OnRamp" program (see below).

To our amazing Authors and Editors, and also their Partners and Families, thank you for your time and sacrifice. This book could not exist without you.

To you, the Reader, I hope you enjoy reading this book as much as we enjoyed writing it.

> This is a Leanpub "Agile-published" book. All the work for this book has been completed. However, as issues are reported, supplementary updates may be released. Leanpub will send out an email when the book is updated. These revisions will be available at no extra charge. To provide feedback, use the "Email the Authors" link on the book's Leanpub web page[64]. Whether it's a code error, a typo, or a request for clarification, our editors will review your feedback, make changes, and re-publish the book. Unlike the traditional paper publishing process, your feedback can have an immediate effect.

About OnRamp

OnRamp is an entry-level education program focused on PowerShell and Development Operations. It is a series of presentations that are held at the PowerShell + DevOps Global Summit[65] and is designed for entry-level technology professionals who have completed foundational certifications such as CompTIA A+ and Cisco IT Essentials. No prior PowerShell experience is required. Basic knowledge of server administration is beneficial. OnRamp ticket holders will be able to network with other Summit attendees who are attending the scheduled Summit sessions during keynotes, meals, and evening events.

Through fundraising and corporate sponsorships, The DevOps Collective, Inc.[66] will be offering several full-ride scholarships to the OnRamp track at the PowerShell + DevOps Global Summit.

All (100%) of the royalties from this book are donated to the OnRamp scholarship program.

[64] https://leanpub.com/modernautomationwithpowershell

[65] https://powershell.org/summit/

[66] https://devopscollective.org/

More information about the OnRamp track[67] at the PowerShell + DevOps Global Summit and their scholarship program[68] can be found on the PowerShell.org[69] website.

See the DevOps Collective Scholarships cause[70] on Leanpub.com[71] for more books that support the OnRamp scholarship program.

Prerequisites

Prior experience with PowerShell is recommended. This book is written for intermediate audiences with intermediate experience with PowerShell.

A Note on Code Listings

Paperback Readers can access digital copies of examples from the book at: https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/tree/main/Edition-01/Examples[72]

If you've read other PowerShell books from LeanPub, you probably have seen some variation on this code sample disclaimer. The code formatting in this book only allows for about 75 characters per line before the text will start automatically wrapping. All attempts have been made to keep the code samples within that limit, although sometimes you may see some awkward formatting as a result.

For example:

```
Get-CimInstance -ComputerName $computer -ClassName Win32_LogicalDisk -Filter 'DriveTyp\
e=3' -Property DeviceID, Size, FreeSpace
```

Here, you can see the default action for a line that is too long—it gets word-wrapped, and a backslash is inserted at the wrap point to let you know it has wrapped. Attempts have been made to avoid those situations, but they may sometimes be unavoidable. Instead of having a long command that wraps, splatting[73] is used instead.

[67] https://powershell.org/summit/summit-onramp/
[68] https://powershell.org/summit/summit-onramp/onramp-scholarship/
[69] https://powershell.org/
[70] https://leanpub.com/causes/devopscollective
[71] https://leanpub.com/
[72] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/tree/main/Edition-01/Examples
[73] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_splatting?view=powershell-7

```
$params = @{
    ComputerName = $computer
    ClassName    = 'Win32_LogicalDisk'
    Filter       = 'DriveType=3'
    Property     = 'DeviceID', 'Size', 'FreeSpace'
}

Get-CimInstance @params
```

> If you read this book on a Kindle, tablet, or another e-reader, use the PDF manuscript, not EPUB. EPUB has known formatting issues with Block Types (For Example, Warning, Tips, Errors), Code Samples, and Annotations.

When writing PowerShell expressions, you shouldn't be limited by these constraints. All downloadable code samples will be in their original form.

Feedback

Have a question, comment, or feedback about this book? Please share it via the Leanpub forum dedicated to this book. Once you've purchased this book, log in to Leanpub, and click on the "Join the Forum" link in the Feedback section of this book's webpage[74].

[74] https://leanpub.com/modernautomationwithpowershell

I Collaboration

> *"Coming together is a beginning, staying together is progress, and working together is success."*
> — Henry Ford

Collaboration within Software Development and DevOps teams is crucial for your team's success. In this section, you'll cover the fundamentals of writing, checking in code using Git, and performing code reviews for peers. Within Git, an emphasis will be placed on covering critical aspects of the technology, focusing on terminology, branching structure, creating and merging branches, and inevitably starting again when things *really* go wrong. Code reviews will focus on how to start, things to consider, and most importantly, best practices.

Onwards!

1. Introduction to Git

Git is source code management software that allows many individuals to contribute to a project at the same time. Every contributor of a Git project has their own local files to which they make and commit changes. This makes Git a *distributed* version control system, as opposed to the *server-client* model. After edits are finished, they can submit a pull request to have their changes pulled into the larger project. This chapter introduces you to Git, giving you the tools needed to contribute to a larger project or even track changes on your own project in a Git repository.

1.1 Understanding Terminology

There are many terms in Git. Below are a few terms you should understand before starting.

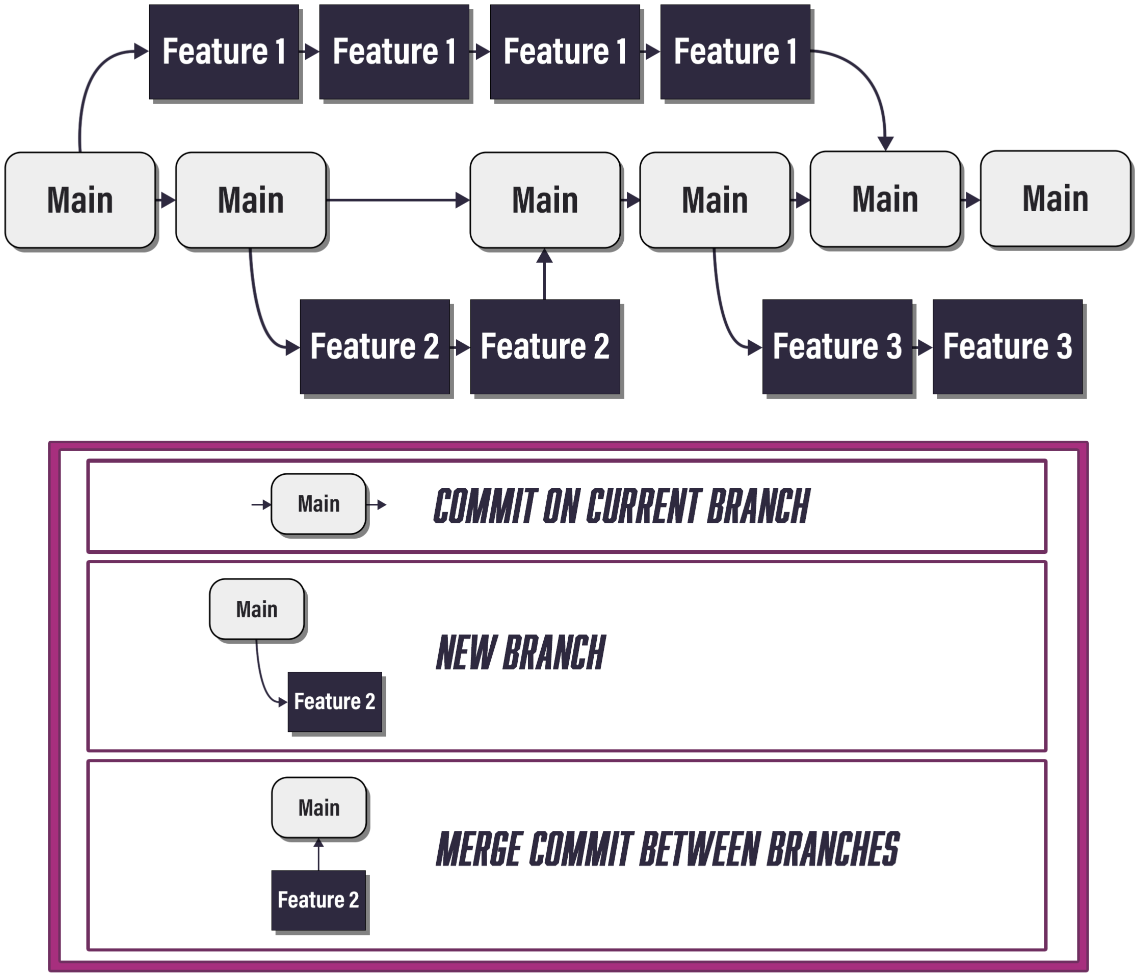
Branch: Separates the changes of a project. Creating branches allows features and fixes to be added in a way that allows the *main* branch to be unaffected by changes until the feature or fix is completed and tested. A branch should be incorporated back into the *main* branch when it's complete. The *main* branch of a project can be renamed, but is commonly initialized as either *main* or *master* and can also be referred to as the *default* branch.

Commit: A snapshot or 'checkpoint' of the changes to a branch. This can be thought of as hitting the save button on a project. All the commits in a branch make up its history and define its current state.

Fork: A new repository created ('copied') from an existing repository. Forks are typically used when a new project is created using another project's code as its base. The changes of a fork can be incorporated back into the original project, but this isn't always the case.

Repository (repo): A directory and its contents where Git has been initialized. Initializing Git creates a hidden `.git` folder that contains project metadata and tracks changes in the directory. You can start a Git repository on your local system or you can clone a remote repository stored on a host like GitHub.

This chapter uses the terms *folder* and *directory* interchangeably.



Git Sample

1.2 Creating a Local Repository

To follow along, you need Git installed on your system. There are several ways to install Git using package managers and other software installation tools. The official Git page can be found at git-scm.com[1] and should be referenced for installation steps.

Once Git is installed, configure your username and email address:

Example 1: Setting the Git global username

```
git config --global user.name "[firstname lastname]"
```

Set a name that's identifiable and can be used for credit in version history:

[1] https://git-scm.com/

Example 2: Setting the Git global email

```
git config --global user.email "[valid-email]"
```

If using GitHub or another service, use the email associated with your account. This allows the service to identify you and associate your account with each history marker.

Once configured, open PowerShell and create your project directory. These examples use `C:\Repo` as the base path. Depending on your system permissions, operating system, or preference, you may want to use a different directory:

Example 3: Creating a directory for the Git repository

```
New-Item -Type Directory -Path C:\Repo\Project
```

```


    Directory: C:\Repo


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         2/26/2022  10:29 AM                Project
```

Once created, navigate to the folder:

Example 4: Setting the working directory to where the Git repository will be

```
Set-Location -Path C:\Repo\Project\
```

The PowerShell prompt should display your current folder:

```
PS C:\Repo\Project>
```

Initialize the folder as a Git repository:

Example 5: Initializing a new repository with `git init`

```
git init -b main
```

```

Initialized empty Git repository in C:/Repo/Project/.git/
```

Currently, the default branch name of a new repository in Git is *master*, but this will soon change to *main*.[2] You can use the `-b <name>` parameter of `git init` to set the default branch. You can also control the default name for default branches with the `init.defaultbranch` config setting.

You'll notice that this directory now contains the `.git` subdirectory.

[2] Git developers. (2022, Jul. 04). *git-init Documentation*. Git Documentation. [Online]. Available: https://git-scm.com/docs/git-init. [Accessed: Jul. 15, 2022].

Example 6: Displaying the hidden .git directory

```
Get-ChildItem -Hidden
```

```


    Directory: C:\Repo\Project


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d--h--         2/26/2022  10:31 AM                .git
```

The `.git` directory includes files and folders that contain the metadata for your Git repository. Many of the files in the `.git` folder are human readable using a text editor, but you should avoid changing them unless you know what you're doing.

1.3 Cloning an Existing Repository

If you're contributing to an existing project, you'll want to clone the contents of the existing repository to a local directory on your system. Start by navigating to the path where you want to clone the project.

Example 7: Setting the working directory to the parent that'll contain the cloned Git repository

```
Set-Location -Path C:\Repo\
```

```
PS C:\Repo>
```

In the below example, you'll build the URL string in the variable `$Extras`, since the repository URL is long. Use the command `git clone` followed by the URL path of the repository you would like to clone. In this example, the path is from the variable `$Extras`, which points to the Extras[3] repository for this book. By default, the local directory name matches the remote repository name. To make the local directory name more friendly, add the string '*MITA-Extras*' as an additional parameter.

Example 8: Cloning a remote Git repository to create a local one

```
$Extras = 'https://github.com/devops-collective-inc/'
$Extras += 'Modern-IT-Automation-with-PowerShellExtras'
git clone $Extras 'MITA-Extras'
```

[3] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras

```
Cloning into 'MITA-Extras'...
remote: Enumerating objects: 47, done.
remote: Counting objects: 100% (25/25), done.
remote: Compressing objects: 100% (17/17), done.
remote: Total 47 (delta 6), reused 18 (delta 5), pack-reused 22
Receiving objects: 100% (47/47), 13.47 KiB | 1.22 MiB/s, done.
Resolving deltas: 100% (10/10), done.
```

This can take some time depending on the project. When done, you'll end up with a directory containing all the project files, and the `.git` subdirectory. As you'll see below, Git names the directory *MITA-Extras*, using the second parameter you pass to `git clone`.

Example 9: Displaying the newly cloned Git repository

```
Get-ChildItem
```

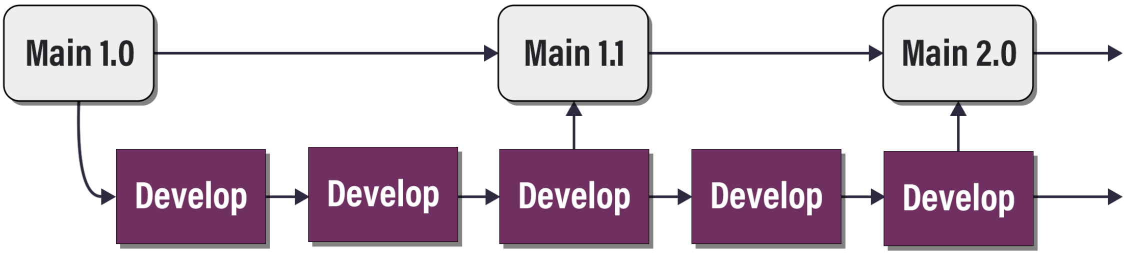
```


    Directory: C:\Repo


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         6/15/2022   7:10 PM                MITA-Extras
d-----         3/25/2022   3:47 PM                Project
```

1.4 Understanding the Flow of Working in Git

In the most simple projects, the workflow for Git is easy to understand. There's a primary working branch typically called *main* and a secondary branch for safely making changes (without breaking the working branch), typically called *develop*.

The case-sensitivity of Git branch names can vary between platforms. To avoid confusion, always use unique branch names regardless of capitalization.

An example of a simple workflow model would look like this:



Basic Workflow

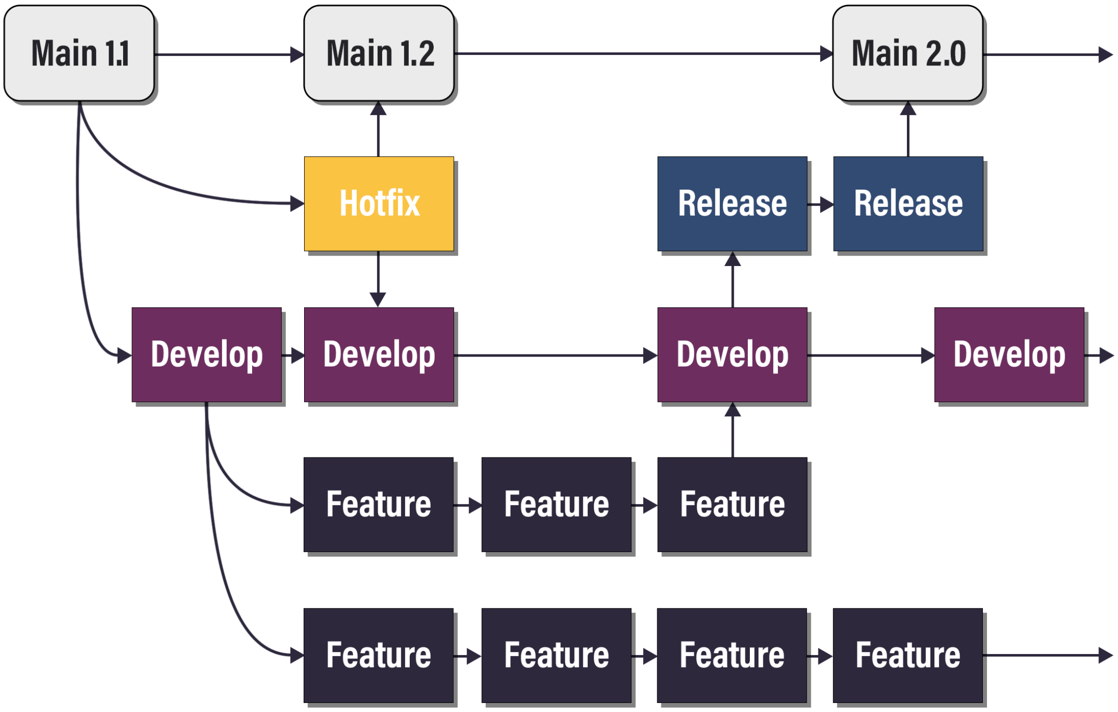
- ***main* Branch**: Working production code
- ***develop* Branch**: Where features are added and changes occur

In the above example, the *main* branch keeps the working copy of code meant to be deployed in production. Changes to the code should be made in the *develop* branch. When the code is tested and known to be working, the *develop* branch can be merged into *main* using a pull request.

As projects get more complex and more contributors are making changes, the workflow can become a bit more confusing. These projects can have many branches, each corresponding to a feature, release, or hotfix. They can also have automated actions that trigger when code is submitted. They may do things like lint, build, or run tests on the code. It's this versatility that makes Git a great tool for projects of all sizes.

One model for managing these branches is called *Gitflow*. With the Gitflow model, feature branches are only committed when they're feature-complete. Gitflow can lead to large commits, but is a well-known branching model.

An example of Gitflow might look like the following:



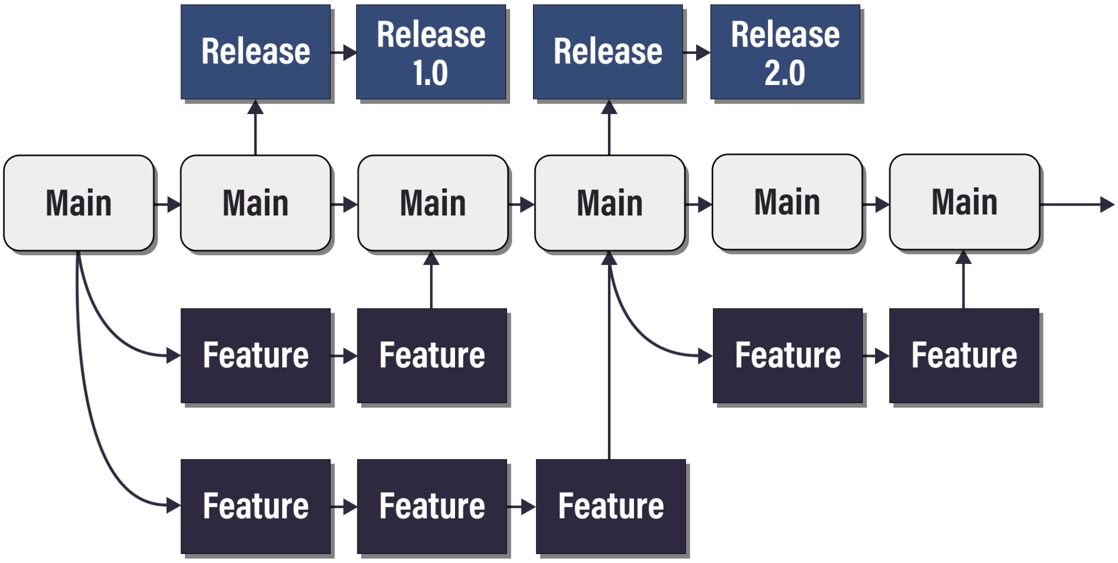
Gitflow Workflow

- ***main* Branch**: Working Code.
- ***hotfix* Branch**: For severe bugs that need to go into production quickly.
- ***release* Branch**: Branch to prepare for new production release. Allows for minor bug fixes and metadata changes.
- ***develop* Branch**: Features are merged into *develop* and used to create release branches.
- ***feature* Branch**: Created to work on a new feature and merged into *develop* when ready.

In the above example, the *main* and *develop* branches are the only ones that extend the length of the project. The other branches exist for the duration of their purpose. Once merged, *release*, *hotfix*, and *feature* branches are removed. Besides the branches mentioned above, there may be sub-branches that individual contributors use to work on their piece of a feature or other branch.

Another model known as *trunk-based* development has become more common with the rise of DevOps and continuous integration/continuous delivery (CI/CD) practices. The trunk-based model comprises smaller, more frequent updates, making it the preferred model for DevOps/CI projects.

An example of trunk-based development might look like the following:



Trunk Workflow

- ***main* Branch**: Working Code.
- ***release* Branch**: A branch to prepare for a new production release. Allows for minor bug fixes and metadata changes. Created from the *main* branch.
- ***feature* Branch**: Created to work on a new feature and merged into *main* when ready.

In the above example, the *main* branch or 'trunk' exists for the life of the project. Release branches are created to prepare for production release. Features are checked into the *main* branch. This workflow is common in environments where continuous integration is being used.

1.5 Your First Commit

After following the steps from above, you'll have a local Git repository created. A helpful command to use is `git status`. It provides information about the repository that you're currently using; if it exists, if there are any changes to commit, what branch you're on, and more. If you run `git status` now, it looks like the following:

Example 10: Using `git status` to view the current status

```
Set-Location -Path C:\Repo\Project
git status
```

```
On branch main

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

Git's messages can be extremely helpful. They'll often help you solve problems when you encounter them. Follow the suggestion of `git status` and create files to track:

Example 11: Creating and showing the presence of a text file in your repository

```
Set-Content ./test.txt "This is a test file"
Get-ChildItem
```

```
    Directory: C:\Repo\Project

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         3/12/2022   3:29 PM             21 test.txt
```

Using `git status` helps you discover what to do next:

Example 12: `git status` tells you when there are files that the repository doesn't track yet

```
git status
```

```
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        test.txt

nothing added to commit but untracked files present (use "git add" to track)
```

Add the file to the repository's *index* using `git add`.

Example 13: `git add` adds new or changed file to the index

```
git add ./test.txt
```

Check the status and you'll see the tracked file under 'changes to be committed'.

Example 14: `git status` also shows staged files or changes

```
git status
```

```
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   test.txt
```

Files and changes in the index are *staged* for the next commit. You'll also find the term *cached*, which means the same thing. Use `git rm --cached <file>` to remove files from the index. `git restore --staged <file>` removes changes to files in the index, which might include removing the whole file.

There are often multiple ways to do a similar task in Git.

The next step is to use `git commit` and provide a message using `-m` parameter followed by your message in quotation marks. If you don't provide a message when running `git commit`, you'll be prompted to enter one via the configured text editor. By default, this is a command-line text editor.

Change Git's text editor with `git config --global core.editor <path>`.

Example 15: Creating your first commit with `git commit`

```
git commit -m "added my first file"
```

```
[main (root-commit) 3bab6a6] added my first file
 Committer: John Doe <jdoe@doe.com>
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 test.txt
```

The term *root-commit* means the first commit of a Git repository, and all new commits and branches build on this. `3bab6a6` is the first seven characters of the hash of the data in the commit. The first seven or eight characters of a commit's hash are commonly used to identify the commit without displaying the whole hash.

Run `git status` to see if there are any other steps that need to be completed:

Example 16: `git status` no longer shows the changes because they're in the latest commit

```
git status
```

```
On branch main
nothing to commit, working tree clean
```

The *working tree* means the actual files and directories in the repository apart from `.git`.

Since there are no uncommitted changes, there's nothing to do.

By running the command `git log` you can show the status of commits and display useful information that'll be used in the event a commit needs to be rolled back.

Example 17: `git log` shows the commit history backward from the current point

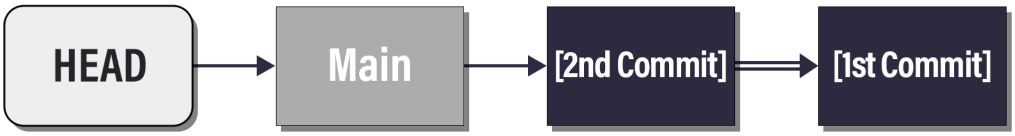
```
git log
```

```
commit 3bab6a6a6886f8b5e5f6a56dd4c01d5812cd2006 (HEAD -> main)
Author: John Doe <jdoe@doe.com>
Date:   Sat Mar 12 15:35:33 2022 -0500

    added my first file
```

As shown above, the commit hash, author, date, and message can be seen from the previous commit. Notice the full commit hash includes the seven characters `3bab6a6` you saw in Example 15.

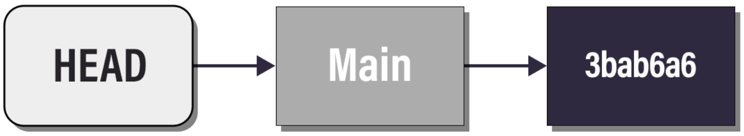
HEAD refers to what Git is looking at in the current repository. Think of it like a pointer to the latest snapshot of the current branch. *HEAD* currently points to *main*, the name of the current branch. *main* currently points to the commit you've just made.

If you made another commit, *main* would now point to that new commit. The new commit would, in turn, point to the last one, forming a **chain** of history and changes.



A Git Commit Chain

The single arrow is a pointer to another object, and the double arrow points to the parent of the commit. In these diagrams, the rightmost commits are the 'oldest' or, more correctly, the furthest ancestors. Using this logic, the current state of your Git repository should be:



The Current State

HEAD and *main* are references, or **refs**. `3bab6a6` is a shortened commit hash and represents a commit object. *HEAD* is a special ref that usually points to the head (*tip*) of a branch, such as *main*.

1.6 Creating a Branch

As mentioned previously, a branch is a way to separate changes in code. Each branch has a timeline of tracked changes that begin from the point the branch is created from its parent branch. In a simple workflow, the development branch is where all changes are made. When a change is finished and tested in the development branch, it can then be merged into the *main* branch using a pull request.

On a website like GitHub, you can create branches using the web interface under the repository page, but it can also be done from the command line. Creating and switching to an existing branch are similar commands leveraging the `git checkout` command. This command can do lots of other things, but creating and switching between branches is the most important to know for now.

From your project's directory, check the status to display the current branch:

Example 18: Making sure there are no uncommitted changes with `git status`

```
git status
```

```
On branch main
nothing to commit, working tree clean
```

Display all branches in the repository to check if your branch already exists:

Example 19: `git branch -a` lists all branches in the repository

```
git branch -a
```

```
* main
```

The asterisk `*` next to *main* tells you what branch you're currently on. If a branch exists on a remote repository (for example on GitHub) but doesn't show up locally, try `git fetch` to pull those branches from the remote repository. `git fetch` gets all the history from a remote repository. If you've cloned another repository, your local one already has a remote pointer called *origin*, which is the URL of the remote you cloned. If you've initialized a new Git repository, it won't have any remotes, yet.

In this example, the Git repository is only local and no remote branches exist. The command `git checkout -b` creates a new branch and switches to it.

Example 20: Creating and switching to a new branch with `git checkout -b`

```
git checkout -b develop
```

```
Switched to a new branch 'develop'
```

View the state of your repository now, with `git status`.

Example 21: The status shows you're now on a new branch called *develop*

```
git status
```

```
On branch develop
nothing to commit, working tree clean
```

Check if anything has changed in your working tree with `Get-ChildItem`.

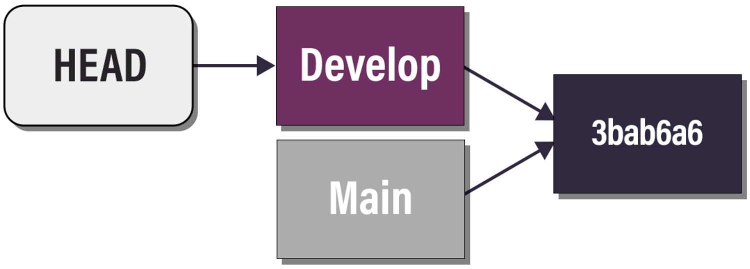
Example 22: Changing to a different branch doesn't affect the working tree

```
Get-ChildItem
```

```
    Directory: C:\Repo\Project


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         3/13/2022  11:25 AM             21 test.txt
```

In the above commands, a new branch was created called *develop*. `git status` shows that the active branch is *develop*, and the file system is unchanged.

Take a look at the diagram again. The repository now looks like this:



Current Repository State

Switching to another branch only changes what *HEAD* points to. Currently, both branches have the same history (they point to the same commit) since you created *develop* from *main*.

For this example, changes can be made to differentiate this branch from *main*, as if it was a feature added to the *develop* branch.

Example 23: Changing the working tree while on the *develop* branch

```
Add-Content ./test.txt "This is a test v2.0"
New-Item -Type File -Path ./newassets.lib
```

```


    Directory: C:\Repo\Project


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         3/24/2022   1:40 PM              0 newassets.lib
```

Inspect the working tree and observe the new file and the change to the size of `test.txt`.

Example 24: Showing the working tree after the changes

```
Get-ChildItem
```

```


    Directory: C:\Repo\Project


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         3/24/2022   1:40 PM              0 newassets.lib
-a----         3/24/2022   1:39 PM             42 test.txt
```

Inspect the contents of `test.txt`, and notice the new line.

Example 25: Showing the changes to *test.txt* with `Get-Content`

```
Get-Content ./test.txt
```

```
This is a test file
This is a test v2.0
```

In the example, a new line is added to the test file and a new file called *newassets.lib* is created.

In order for the changes to be tracked in the branch, the changes need to be added and committed. Running `git status` helps you if you don't know what to do.

Example 26: `git status` also provides tips about what to do next

```
git status
```

```
On branch develop
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   test.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        newassets.lib

no changes added to commit (use "git add" and/or "git commit -a")
```

Unlike before, there are both files not yet tracked, and changes not yet staged. All files in the directory can be added to the index with `git add .`, or individually added with a unique file name. In this example, all new files and changes are included:

Example 27: Staging all changes in the working tree into the index

```
git add .
git status
```

```
On branch develop
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   newassets.lib
        modified:   test.txt
```

As the output suggests, the next step is to commit those changes to the current branch:

Example 28: Committing changes on the new branch *develop*

```
git commit -m "First commit to develop"
```

```
[develop 2bc6d24] First commit to develop
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 newassets.lib
```

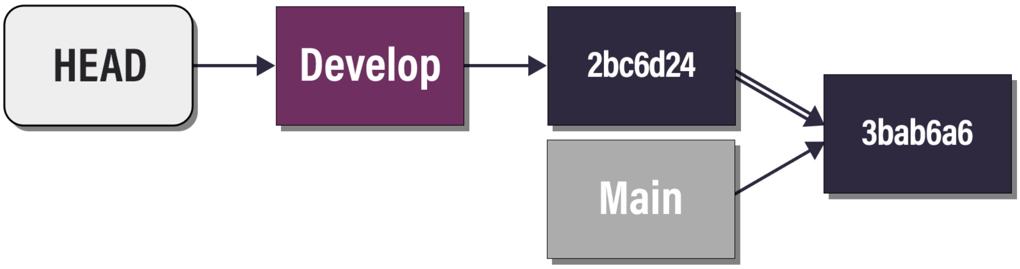
Notice the new short hash `2bc6d24` is different because the data has changed. Check the status once again with `git status`.

Example 29: The status after the commit shows the changes are now incorporated in the branch

```
git status
```

```
On branch develop
nothing to commit, working tree clean
```

The repository history now looks like this:



Current Repository History

The double arrow shows that `3bab6a6` is the parent of `2bc6d24`.

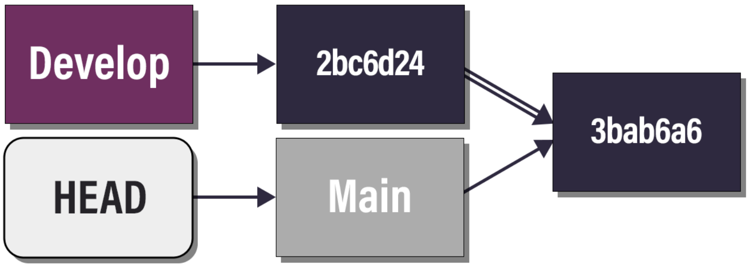
Now that the changes were committed to *develop*, switching branches shows the differences between *main* and *develop*. Since the *main* branch already exists, use `git checkout` without the `-b` parameter to switch branches without creating a new one:

Example 30: Switching back to the *main* branch

```
git checkout main
```

```
Switched to branch 'main'
```

Two things have happened this time. First, *HEAD* now points to *main* again.



HEAD Pointing to main

This time, though, the branches have different histories. Git changes all files in the working tree (the actual files in the repository) to match the state saved in the latest commit of *main*. In this case, it's the state at commit `3bab6a6`. Prove this by looking at the contents of the working tree.

Example 31: Inspecting the working tree after switching branches

```
# Example 31a: Inspect the working tree
Get-ChildItem

# Example 31b: Inspect the contents test.txt
Get-Content ./test.txt
```

```
# Example 31a:

    Directory: C:\Repo\Project


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         3/24/2022   1:59 PM             21 test.txt

# Example 31b:
This is a test file
```

In the example, *newassets.lib* no longer exists and *test.txt* has been reverted to its previous state where the 'v2.0' line wasn't included. The changes you made on the *develop* branch are gone, but not lost. These are safely saved in the commit (`2bc6d24`) you made on *develop*.

Example 32: Showing the two branches with different histories

```
git branch -a
```

```
  develop
* main
```

Once again, the asterisk `*` signifies the *main* branch is selected.

1.7 Merging Branches

The `git merge` command can be used when working in a local repository to combine the changes from *develop* into *main*.

When using a remote repository like GitHub, you can typically submit a *Pull Request* or *PR* from the repository's webpage. The PR notifies others on the project of changes and allows them to approve and merge those changes.

Another way you can use a pull request is to merge changes between your repository and another. As an example, if you wanted to contribute to an open-source project, you would start by creating a fork of the repository. The fork creates a copy of the repository under your account. You can change your forked copy and then submit a PR between the two repositories (the original and your forked version). If the owner of the original repository approves your changes, they'll become part of that repository.

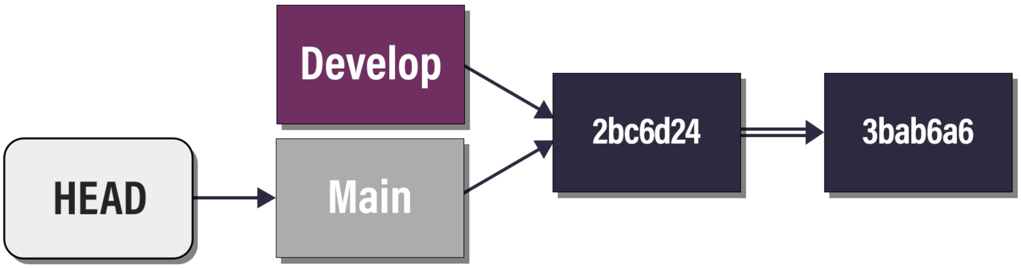
The example below merges the changes from *develop* into the *main* branch.

Example 33: Merging the contents of *develop* into *main*

```
# Example 33a: Switch to the 'main' branch
git checkout main

# Example 33b: Merge the changes from 'develop' into the current branch
git merge develop
```

```
# Example 33a:
Switched to branch 'main'

# Example 33b:
Updating 3bab6a6..2bc6d24
Fast-forward
 newassets.lib | 0
 test.txt      | 1 +
 2 files changed, 1 insertion(+), 0 deletions(-)
 create mode 100644 newassets.lib
```

Fast-forward in the output is the merge mode that Git used. Since there are no changes on *main* and it's an ancestor of *develop*, Git only needs to add the additional commit from *develop* to the history of *main*. The branch now looks like this:



The Current Repository State

main and *develop* now have the same histories. Inspect the working tree to prove this.

Example 34: Inspecting the working tree of *main* after merging

```
# Example 34a: Inspect the working tree
Get-ChildItem

# Example 34b: Inspect the contents of test.txt
Get-Content ./test.txt
```

```
# Example 34a:

    Directory: C:\Repo\Project


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         3/25/2022   2:06 PM              0 newassets.lib
-a----         3/25/2022   2:06 PM             42 test.txt

# Example 34b:
This is a test file
This is a test v2.0
```

All the changes from *develop* have been merged into the *main* branch. In the above example, you can see that *newassets.lib* was created and that *test.txt* contains the 'v2.0' update.

1.7.1 Merge Commits

When fast-forwarding isn't possible, Git creates a new commit that contains the changes from both of the branches you're merging. This commit has two parents—the latest commit from each branch—and is called a *merge commit.*

Start by resetting the *main* branch back to the state it was in before the fast-forward merge. You'll find out more about resets later in the chapter.

Example 35: Resetting the *main* branch to before the merge

```
git reset --hard 3bab6a6
```

```
HEAD is now at 3bab6a6 added my first file
```

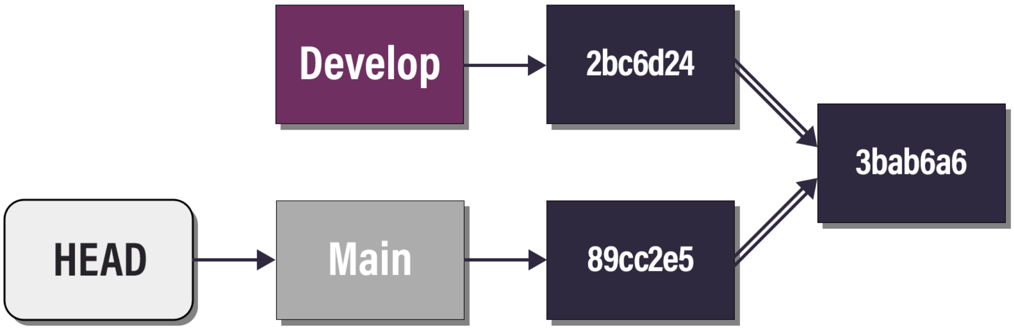
Add a new file and commit it to *main*:

Example 36: Resetting the *main* branch to before the merge

```
Set-Content anotherfile.txt "This is another file"
git add ./anotherfile.txt
git commit -m "Added another file"
```

```
[main 89cc2e5] Added another file
 1 file changed, 1 insertion(+)
 create mode 100644 anotherfile.txt
```

Your repository now looks like this:



Current Repository History

Try merging in the *develop* branch again:

Example 37: Creating a merge commit to merge two sets of changes

```
git merge develop
```

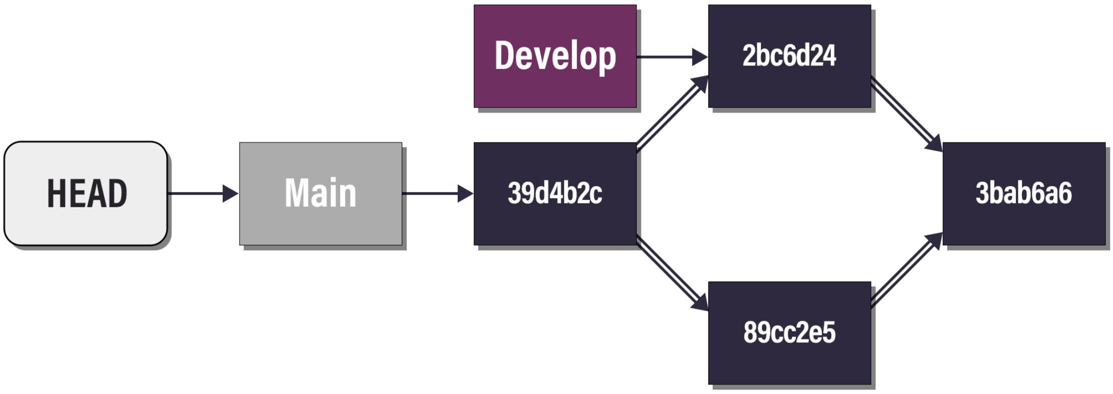
```
Merge made by the 'recursive' strategy.
 newassets.lib | 0
 test.txt      | 1 +
 2 files changed, 1 insertion(+)
 create mode 100644 newassets.lib
```

Git can't fast-forward *main*, because the `89cc2e5` commit would be lost, or *orphaned*. If the two branches have changed different files, or different lines of the same file, they're still merge-compatible. Git creates a new commit with changes from both branches. This commit has a new hash since the data differs from that of either parent commit. Check this commit by looking at the last log entry.

Example 38: Viewing the merge commit in the Git log

```
git log --max-count 1
```

```
commit 39d4b2c31e93b56f419555e1146568ef28711050 (HEAD -> main)
Merge: 89cc2e5 2bc6d24
Author: John Doe <jdoe@doe.com>
Date:   Sat Mar 12 15:45:26 2022 -0500

    Merge branch 'develop'
```



Current Repository History

The merge commit `39d4b2c` has two parents, `2bc6d24` from *develop* and `89cc2e5` from *main*. You've now reconciled the two histories that had diverged. If *develop* was a feature branch, it would now be safe to delete it. All the history from *develop* is reachable from *main*, so no commits would be *orphaned* by deleting the branch.

1.7.2 Merge Conflicts

If two branches that you want to merge change the same data in the same file, it creates a *merge conflict*. In these instances, Git can't automatically decide which data it should keep, or whether it should take bits from both branches.

When you run `git merge` under these conditions, Git pauses the merge, allowing you to resolve the conflicts manually. Git inserts both sets of changes into the conflicting file, with separators to help you identify where they're from.

To try this, reset *main* to the state it was in before the merge, again.

Example 39: Resetting the *main* branch to before the merge

```
git reset --hard 3bab6a6
```

```
HEAD is now at 3bab6a6 added my first file
```

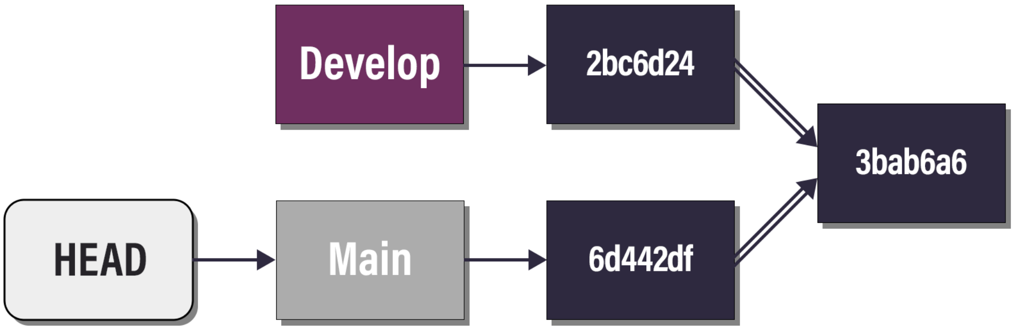
Add a different second line to `test.txt` than the one in *develop*.

Example 40: Adding a conflicting line to `test.txt` on *main*

```
# Example 39a: Commit changes to line 2 of test.txt
Add-Content ./test.txt 'This is a different line'
git add ./test.txt
git commit -m 'Added a different second line'

# Example 39b: View the changed file
Get-Content ./test.txt
```

```
# Example 39a:
[main 6d442df] Added a different second line
 1 file changed, 1 insertion(+)

# Example 39b:
This is a test file
This is a different line
```

Your repository now looks like this:



Current Repository State

This time, though, the two commits `2bc6d24` from *develop* and `6d442df` from *main* conflict with each other. Run `git merge` again and observe that a merge conflict results.

Example 41: Attempting to merge *develop* into *main* with conflicts in `test.txt`

```
# Example 40a: Try to merge
git merge develop

# Example 40b: View the status
git status

# Example 40c: View the conflicted file
Get-Content ./test.txt
```

```
# Example 40a:
Auto-merging test.txt
CONFLICT (content): Merge conflict in test.txt
Automatic merge failed; fix conflicts and then commit the result.

# Example 40b:
On branch main
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Changes to be committed:
        new file:   newassets.lib

Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both modified:   test.txt

# Example 40c:
This is a test file
<<<<<<< HEAD
This is a different line
=======
This is a test v2.0
>>>>>>> develop
```

Don't panic! There are a couple of approaches to resolving merge conflicts. The first is to choose the data you want to keep manually and change the file. Pick the line to keep, and remove the rest, including the three separators. Alternatively, you can keep all the changes in a file from one branch by *checking it out*.

Example 42: Resolving the conflict by keeping the `test.txt` changes from *develop*

```
# Example 41a:
git checkout --theirs ./test.txt
git add ./test.txt

# Example 41b:
git status
```

```
# Example 41a:
Updated 1 path from the index

# Example 41b:
On branch main
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:
        new file:   newassets.lib
        modified:   test.txt
```

Some important terminology to know is *ours* and *theirs*. *ours* refers to the changes from the current branch, which is *main* in this case. *theirs* refers to the changes from the incoming branch, which is *develop* in this case. If you wanted to keep the changes from *main*, you would run `git checkout` with `--ours`, instead. `git add` adds the resolved changes to the index so that Git can complete the merge.

As you can see from the output of `git status`, there's only one more step. To complete the merge and create the merge commit, run `git commit`.

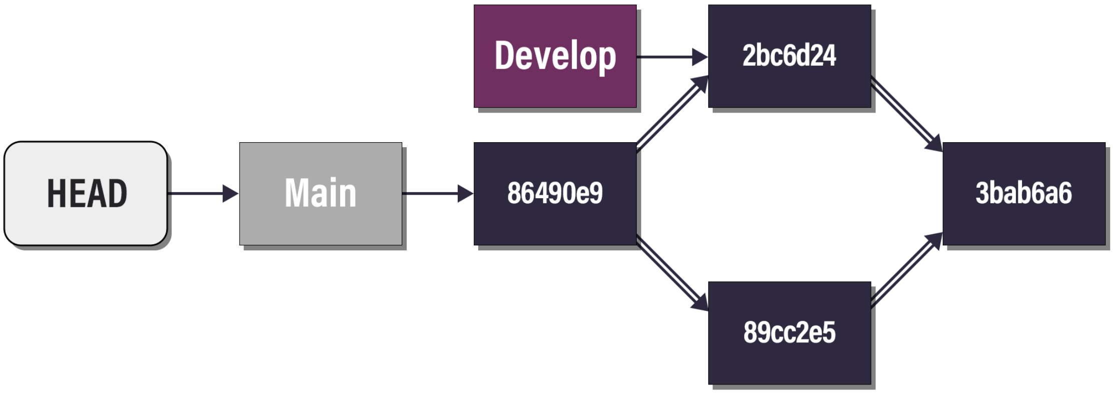
Example 43: Completing the conflicted merge with `git commit`

```
git commit
```

```
hint: Waiting for your editor to close the file...
[main 86490e9] Merge branch 'develop'
```

Git prepares a merge commit message, so you don't need to use `-m 'message'` unless you want to override it. The message opens in your chosen text editor so you can interactively add any notes about the conflict resolution. Once you close this, Git creates the merge commit, reconciling all the changes you picked.

Your repository now looks like this:



Current Repository History

The merge commit `86490e9` has two parents as usual, but deletes the lines you didn't keep.

Another option for resolving merge conflicts is to use a merge tool such as `vimdiff`,[4] but this chapter doesn't cover these.

1.8 Stashing Changes

An important task before merging is ensuring your working tree is clean, with no uncommitted changes. While it may sometimes be safe to perform a merge without affecting those changes, there are many instances where incoming changes from another branch interfere with your uncommitted changes.

`git stash push` creates a temporary commit with the uncommitted changes in your working tree, and stores a pointer to that commit in a special list. This means your changes are safely out of the way, allowing you to switch branches, merge, and make commits without affecting—or being affected by—those changes.

By default, `git stash push` only stashes changes that the repository is tracking, such as files in the index. To stash untracked files too, include the `-u` switch. The `-m '<message>'` parameter sets a custom message for the stash entry.

When you want the changes back, you can run `git stash pop` to restore them and remove that temporary commit from the stash list. Alternatively, `git stash apply` restores the changes but leaves the stash entry in the list.

The stash list acts like a stack, so each time you run `git stash push`, Git adds it to the top of the list. You can, however, address individual stash items using the syntax `stash@{n}` where `n` is the number in the list, starting at zero.

For example, `git stash apply 'stash@{1}'` restores the contents of the second-to-last stash entry, but leaves it in the list. To view all your stash entries, use `git stash list`.

1.9 Rolling Back When Things Go Wrong

Recall that every commit has a unique ID (hash) that you can use to revert changes to a previous state. In the event something goes wrong, using `git reset --hard [hash]` allows you to roll back the current branch to a previous commit. The `hard` parameter has the effect of removing all changes and forcing the reset to the provided commit. Other parameters for reset are `soft` and `mixed`. To understand the differences between these, there are several terms you must know.

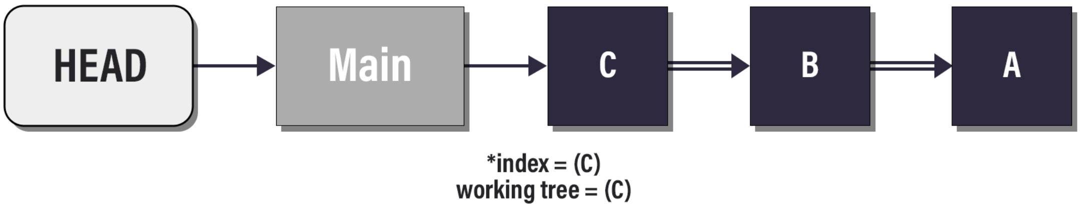
- **Working Tree**: The actual files and subdirectories in the repository except `.git`.
- **Index**: A list of files with uncommitted changes that have been staged using `git add`.
- **Head**: A pointer that points to the latest commit or *tip* of a branch, for example *main* or *develop*.

[4] Git developers. (2022, Jul. 04). *vimdiff Documentation*. Git Documentation. [Online]. Available: https://git-scm.com/docs/vimdiff. [Accessed: Jul. 16, 2022].

- **HEAD**: The special ref that points to a branch head.

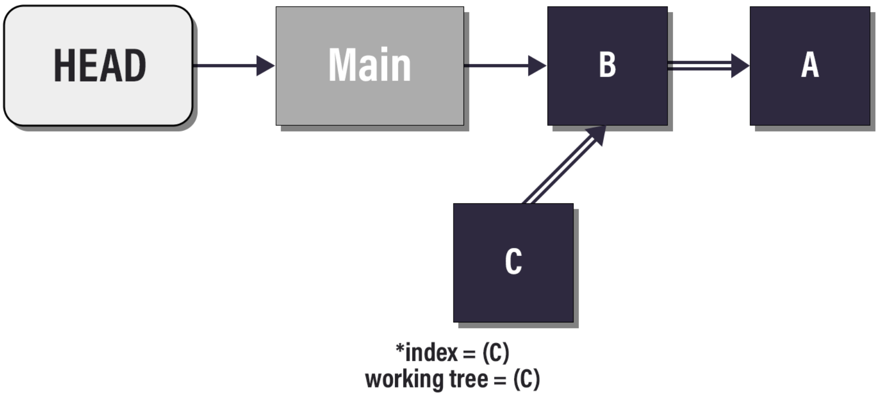
`git reset` changes where a branch pointer is pointing; the below example should help explain.

Imagine a branch called *main* has a commit history of three commits `A`, `B`, and `C`. The changes in those commits are represented as `(A)`, `(B)`, and `(C)`.



Imaginary Commit History

After running `git reset --soft B`, *main* points to `B`. Running `git status` shows the changes `(C)` as staged, and running `git commit` creates a new commit with the same changes `(C)`. The changes `(C)` show as staged because the index wasn't reset, and still contains them. A soft reset simply changes where a branch head is pointing. This can be useful if you want to remove a file that was erroneously committed or change the commit message.

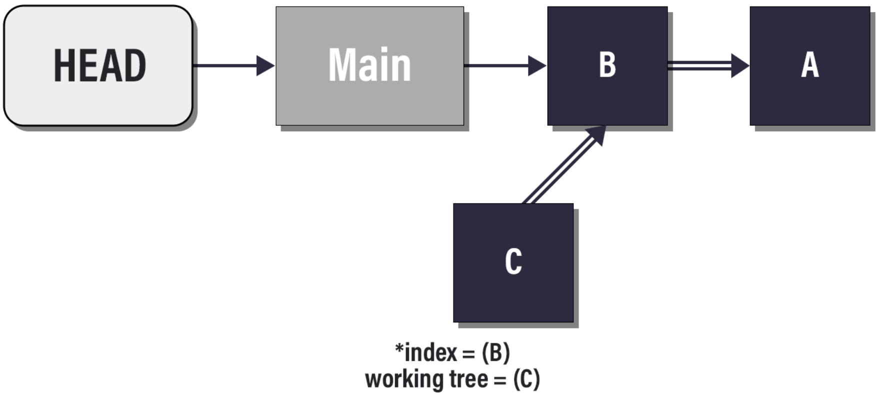


A Soft Reset

`C` is now an *orphaned* commit, meaning no refs or commits point to it. You can still access it using its hash, but Git will eventually delete it as part of its garbage collection (GC).

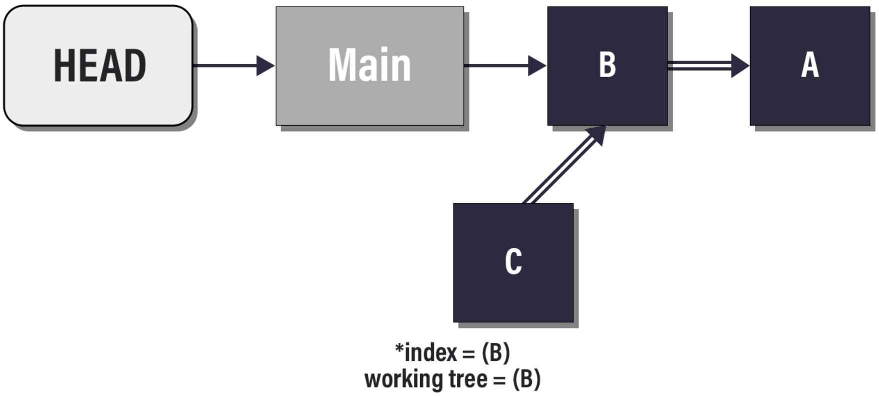
Returning to the three commits example, `A`, `B`, and `C`, running `git reset --mixed B` does two things. The branch head points to `B`, and the index matches `B`. If you run `git commit`, nothing would happen as there are no differences between the branch head and the index.

The files in the working tree are unaffected, so the changes `(C)` still exist in your working directory, but are not staged. Assuming you add the same files with `git add` and then commit with `git commit`, there'll now be a new commit with the changes `(C)`. A mixed reset changes the branch head and index, which is useful for fixing issues in a commit or removing a file that was erroneously committed.



A Mixed Reset

Finally, start over again with the three commits example, `A`, `B`, and `C`. Running `git reset --hard B` does three things. The branch head points to `B`, the index matches `(B)`, and all changes to files in the working directory are reverted to `(B)`. All later changes to files in the working directory are lost. Run `git status` to ensure there aren't changes that you may want to keep before doing a hard reset. A hard reset changes the branch head, index, and working tree. This is useful for getting back to a commit with a known good state.



A Hard Reset

Only tracked files in the working tree are reset during a hard reset. Use `git add --all` to ensure all files are reset. Alternatively, use `git clean -f` to delete all untracked files in the working tree.

1.9.1 Hard Reset in Action

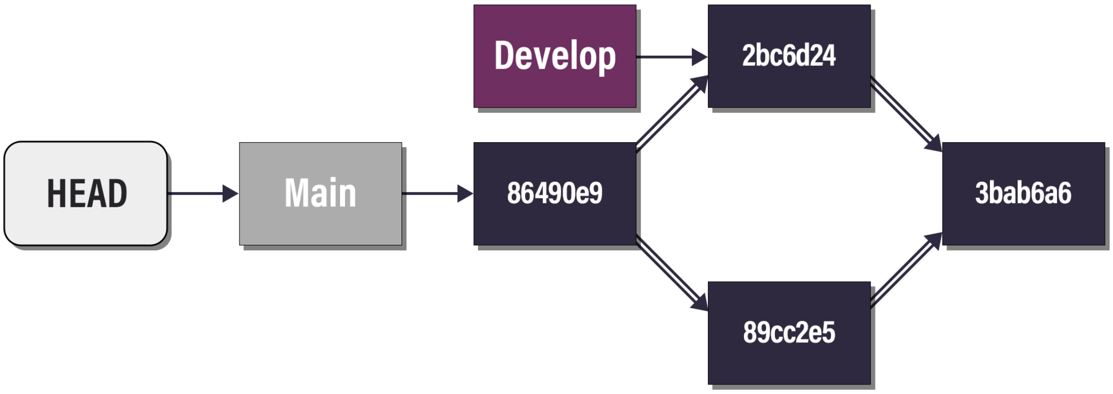
The following examples show you how to perform a hard reset.

Example 44: Inspecting the Git log before a hard reset

```
git log
```

```
commit 86490e9d4ec78c3fb9794ef3a4845d9c50f3e775 (HEAD -> main)
Merge: 6d442df 2bc6d24
Author: John Doe <jdoe@doe.com>
Date:   Thu Mar 24 14:45:11 2022 -0400

    Merge branch 'develop'

commit 6d442df13e3d5639572e3e66924355d467adbea4
Author: John Doe <jdoe@doe.com>
Date:   Thu Mar 24 14:10:29 2022 -0400

    Added a different second line

commit 2bc6d244bcd8eede120733d2bb1e455f2b959521 (develop)
Author: John Doe <jdoe@doe.com>
Date:   Thu Mar 24 13:56:20 2022 -0400

    First commit to develop

commit 3bab6a6a6886f8b5e5f6a56dd4c01d5812cd2006
Author: John Doe <jdoe@doe.com>
Date:   Sat Mar 12 15:35:33 2022 -0500

    added my first file
```

Using `git log`, you can see all previous commits in the current branch.



Current Repository History

This is where good commit messages come in handy. The commit to roll back to is the 'added my first file' commit with hash starting `3bab6a6`.

🔑 Only the first few unique characters in the hash are needed; enough that Git can be sure which commit hash you mean.

Example 45: Performing a hard reset to commit `3bab6a6` on *main*

```
git reset --hard 3bab6a6
```

```
HEAD is now at 3bab6a6 added my first file
```

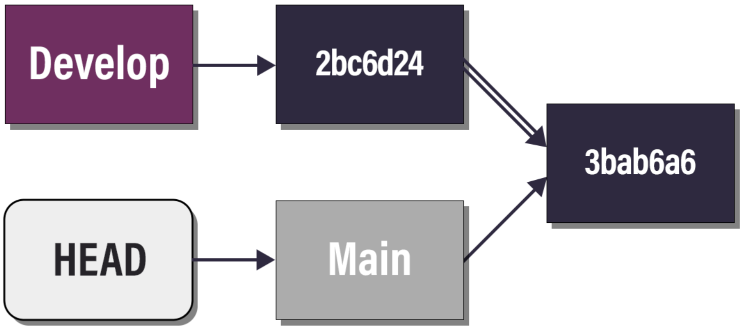
Inspect the working tree to confirm that the files in it were also reverted.

Example 46: Inspecting the working tree after the hard reset

```
# Example 46a: List the files in the working tree
Get-ChildItem

# Example 46b: Show the contents of test.txt
Get-Content ./test.txt
```

```
# Example 46a:

    Directory: C:\Repo\Project


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         3/25/2022   3:05 PM             21 test.txt

# Example 46b:
This is a test file
```

Everything is back as it was before merging *develop* into *main*.

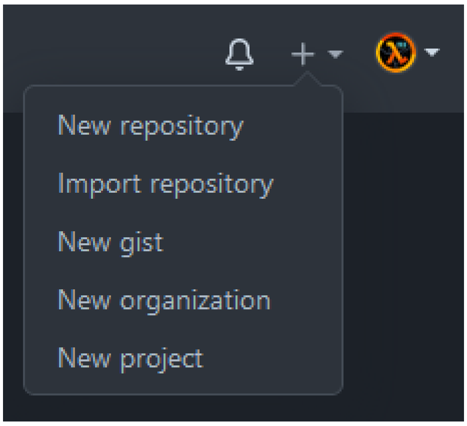


The Reverted State

1.10 Connecting to a Remote Repository

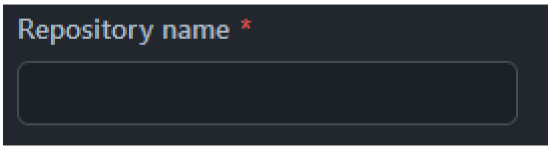
Earlier in the chapter, `git clone` is used to clone a remote repository on GitHub to a local directory. Everything else in this chapter has only made changes to the local Git repository. In this example, the local Git repository is pushed to a remote repository on GitHub. Once pushed to GitHub, the repository acts as a backup to your local files and a place where others can contribute.

To start, you'll need a GitHub account. Most services provided by GitHub are free for public repositories and many are also free for private ones.[5] You can create an account[6] on the GitHub website to get started. Once logged in, you'll need to create a new repository. On the homepage[7], click the plus + icon in the top-right corner, and then '*New repository*':



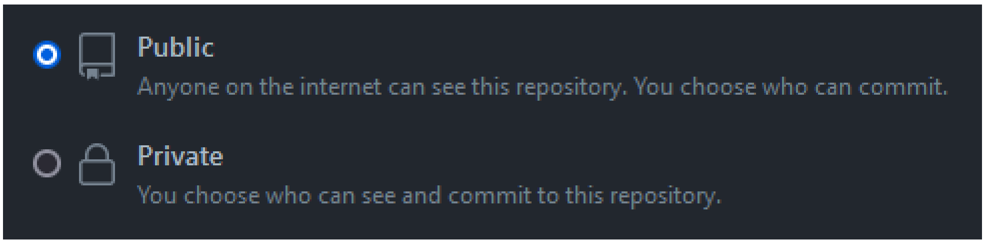
New repository button

Choose a name for the repository. This should usually be the same or similar to the name of the directory that contains your local one. The following examples assume you called the repository *test*.



Repository name box

Set your project to *Private* if you don't want others to see it. Some advanced features are disabled for private repositories with a free account.



● Public
Anyone on the internet can see this repository. You choose who can commit.

○ Private
You choose who can see and commit to this repository.

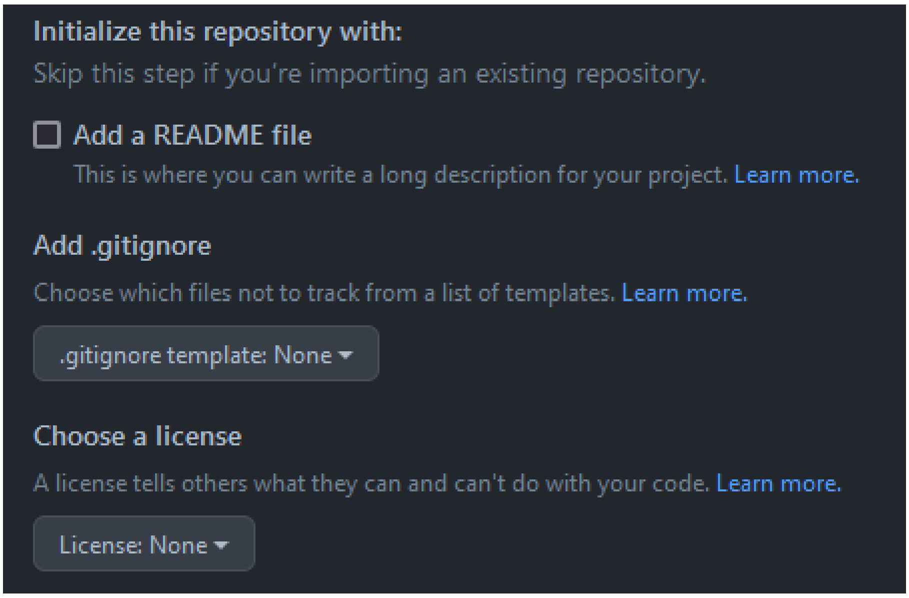
Repository visibility setting

There are normally options such as '*Add a README file*' or '*Add .gitignore.*' Ignore these, as they initialize the repo with files and a default branch. You don't want this since your local repository is already initialized.

[5] GitHub. (2022). *Pricing: Plans for every developer.* GitHub. [Online]. Available: https://github.com/pricing#compare-features. [Accessed: Jul. 16, 2022].

[6] https://github.com/signup

[7] https://github.com/



Initialize this repository with:

Skip this step if you're importing an existing repository.

□ Add a README file

This is where you can write a long description for your project. Learn more.

Add .gitignore

Choose which files not to track from a list of templates. Learn more.

.gitignore template: None ▾

Choose a license

A license tells others what they can and can't do with your code. Learn more.

License: None ▾

Repository initial files setting

Click '*Create repository*' to continue. On the next page, there are steps for initializing and pushing a local repository. Find the steps that say '*...or push an existing repository from the command line.*' Using these steps, push your local repository to the remote GitHub one.

In the examples ahead, replace `UserName` with your GitHub username. If you've used a different repository name than *test*, replace this too.

Example 47: Adding your new GitHub repository as a remote to your local one and pushing *main*

```
git remote add origin https://github.com/UserName/test.git
git branch -M main
git push -u origin main
```

```
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 247 bytes | 247.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/UserName/test.git
 * [new branch]      main -> main
Branch 'main' set up to track remote branch 'main' from 'origin'.
```

The example first adds a remote repository and calls it *origin*. The second command renames the current branch to *main*, to ensure a branch with that name exists. This shouldn't have any effect if you're using the same branch names as in the examples.

The final step is to push the *main* branch to the remote repository *origin*. The additional switch `-u` to `git push` sets your local branch up to track the status of the same branch in the remote repo. The term for this is setting the **upstream** branch.

GitHub now contains the *main* branch, and your local *main* branch tracks the remote copy, but the *develop* branch is missing. To push the *develop* branch, run `git push` with *develop* or specify all branches with `git push --all origin`. Use `-u` again to make sure the local *develop* branch tracks the one on GitHub.

Example 48: Pushing the *develop* branch to GitHub

```
git push -u origin develop
```

```
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 16 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 336 bytes | 336.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
remote:
remote: Create a pull request for 'develop' on GitHub by visiting:
remote:      https://github.com/UserName/test/pull/new/develop
remote:
To https://github.com/UserName/test.git
 * [new branch]      develop -> develop
Branch 'develop' set up to track remote branch 'develop' from 'origin'.
```

Information in the output points out that there are changes in *develop* that aren't in *main*; this is expected.

When it's time to merge changes from *develop* into *main*, the link provided allows you to create a pull request on GitHub. The pull request is like your local merge, but happens on the remote repository, using the web interface on GitHub. If you were to create, then merge a pull request from *develop* into *main*, the remote *main* branch and local *main* branch wouldn't be at the same commit anymore.

Changes made locally need to be *pushed* to the remote repository, and changes made remotely need to be *pulled* to your local repository. The following example shows you how to pull changes from the remote repository into your local one.

Example 49: Pulling changes to the remote *main* branch into the local one

```
git checkout main
git pull
```

```
Updating 3bab6a6..4ea0790
Fast-forward
 newassets.lib |   0
 test.txt      | Bin 21 -> 42 bytes
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 newassets.lib
```

Notice that you didn't need to pass the remote name *origin* to `git pull`. This is because the branch is set up to track the remote *main* branch of `origin`. If it wasn't, you could use `git pull origin`.

Under the hood, `git pull` runs two commands. The first, `git fetch`, downloads all the changes from the remote repository and stores them separately from your local changes. The second is usually `git merge`, in order to merge the changes from the remote with your local ones.

Likewise, when changes are made locally, those need to be pushed to the remote repository. In the following examples, you'll add changes to the *develop* branch, stage and commit them, and push them to the remote repository.

Example 50: Switching to the *develop* branch before making changes

```
git checkout develop
```

```
Switched to branch 'develop'
Your branch is up to date with 'origin/develop'.
```

Example 51: Adding a new line to `test.txt` and committing the change

```
Add-Content ./test.txt "This is version 3.0"
git commit -a -m "v3.0 release"
```

```
[develop b9f1bae] v3.0 release
 1 file changed, 0 insertions(+), 0 deletions(-)
```

Note the `-a` switch passed to `git commit`. This automatically stages all changes to tracked files before the commit. Now your local branch is one commit ahead of the remote one. Push the changes to the remote branch.

Example 52: Pushing local changes on *develop* to the remote on GitHub

```
git push
```

```
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 16 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 337 bytes | 337.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/UserName/test.git
   2bc6d24..c27046b  develop -> develop
```

Once again, you don't need to specify the remote and branch names since *develop* is already set up to track *develop* on *origin*. Inspect the `test.txt` file one more time to observe the three lines.

Example 53: Inspecting the three lines in `test.txt` after making changes and pushing them

```
Get-Content ./test.txt
```

```
This is a test file
This is a test v2.0
This is version 3.0
```

You can view the remote configuration of your repository by searching the local config.

Example 54: Inspecting the remote config of the local repo

```
git config --local --get-regexp '^remote'
```

```
remote.origin.url https://github.com/UserName/test.git
remote.origin.fetch +refs/heads/*:refs/remotes/origin/*
branch.main.remote origin
branch.develop.remote origin
```

1.11 Starting Over When Things Really Go Wrong

An advantage to using a remote repository is that there are multiple locations where the project is saved. At a minimum, there is a local and a remote (*origin* for example) copy of the repository. If other people are contributing to the project, they should have local copies as well.

If the local copy of the repository has issues, it can be reset based on the remote repository. As an example, change the history of the *main* branch by soft-resetting to the first commit and creating a new commit with all the changes.

Example 55: Squashing all the changes since the first commit into a single commit

```
git checkout main
git reset --soft 3bab6a6
git commit -m 'All changes in one commit'
```

```
[main a71bba5] All changes in one commit
 3 files changed, 3 insertions(+)
 create mode 100644 newassets.lib
 create mode 100644 test2.txt
```

Check the status of your branch after changing the history:

Example 56: Inspecting the *main* branch after changing its history

```
git status
```

```
On branch main
Your branch and 'origin/main' have diverged,
and have 1 and 4 different commits each, respectively.
  (use "git pull" to merge the remote branch into yours)

nothing to commit, working tree clean
```

The output shows that the histories are now different. The steps below reset the local branch back to the last commit on the remote one. First, update your local copy of the remote repo.

Example 57: Fetching the latest changes from the remote

```
git fetch --all -v
```

```
Fetching origin
POST git-upload-pack (165 bytes)
From https://github.com/JohnDoe/test
 = [up to date]      main       -> origin/main
 = [up to date]      develop    -> origin/develop
```

The `-v` switch means *verbose* and shows you the fetch result. Now, you can hard-reset the branch using the ref that points to the remote copy of *main*.

Example 58: Resetting *main* to the remote copy of *main*

```
git reset --hard origin/main
```

```
HEAD is now at 4ea0790 Merge pull request #1 from JohnDoe/develop
```

You've now rewritten your local history to match that of the remote branch.

Example 59: Checking the status after the remote hard reset

```
git status
```

```
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
```

This is very useful if something goes wrong in your local repository. It also shows that you can use `git reset` with refs as well as commit hashes.

1.11.1 Starting From Scratch

If things ever get messy in your local repository and you don't know what to do, there are a few brute force methods for fixing it.

If you don't care about any of the local changes:

- Delete the local repository.
- Use `git clone` to pull down a fresh copy of your repository.

If you made changes and you need them:

- Copy the changed files to a new location.
- Delete the local repository.
- Clone the remote repository.
- Copy your changed files back into the local repository.
- `git add .` to stage the files.
- `git commit -m "fixing a mess up"`.
- `git push` to push the changes to the remote repository.

1.12 Conclusion

Git is a powerful source control tool. You can use it in simple pipelines with ease or do complex merges of larger projects. Git is an essential tool for developing with others and is a great way to keep track of your code with time.

It's also a tool that allows you to participate in open source projects. It can save you from headaches when you're working on complex scripts, programs, documents and more. SCM tools like Git are essential in the modern day IT environment. Ensure you continue to learn about Git and leverage the many online resources available all over the internet.

1.13 Modern IT Automation With PowerShell Extras

Some chapters in this book include or rely on additional content. All of this is available in a Git repository on GitHub. Clone the Modern IT Automation With PowerShell Extras[8] repository as shown in the Cloning an Existing Repository section to access these.

1.14 Further Reading

- Official Git Documentation—git-scm.com[9]
- Git Commands Reference—git-scm.com[10]

[8] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras

[9] https://git-scm.com/doc

[10] https://git-scm.com/docs

- Complete List of Git Commands—git-scm.com[11]
- GitHub Cheat Sheet—GitHub[12]
- GitHub Signup Page—GitHub[13]
- Git for Windows Installation—git-scm.com[14]
- Git *nix Installation—git-scm.com[15]
- Git macOS Installation—git-scm.com[16]
- *Pro Git, 2nd edition* (free)—git-scm.com[17]
- **MITA Extras repository**[18]

[11] https://git-scm.com/docs/git#_git_commands

[12] https://training.github.com/

[13] https://github.com/signup

[14] https://git-scm.com/download/win

[15] https://git-scm.com/download/linux

[16] https://git-scm.com/download/mac

[17] https://git-scm.com/book/en/v2

[18] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras

2. Code Reviews

Engineers often see code reviews as part of running a formal software development team or project, or as part of a Software Development Lifecycle (SDLC)—not a process that also applies to teams or projects focused on IT automation. The sections ahead aim to dispel that thinking and explore how code reviews can improve the quality of your automation.

The thought of reviewing another's code and giving feedback can be overwhelming; something that's easier said than done. This chapter explores performing a code review, setting up code reviews within your project or team, best practices used within the community, and tools you can use to help you review code.

2.1 Purpose of Code Reviews

Code reviews aren't only part of formal software development projects, or just for those using Agile or Scrum methodologies and writing in programming languages like C#. Many operations teams now see how code reviews extend to their activities. Scripting and automation are often seen as a business-as-usual (BAU) activity, not project-led. While the approach to writing code might differ, the outcome is the same—code is executed in production.

So how can code reviews help?

Code reviews help teams to:

1. Find and reduce bugs and issues within code, preventing issues from occurring in production environments.
2. Improve the sharing of knowledge between team members.
3. Fulfill change management and change review processes under ITIL[1], ISO 20000[2], and other standards.
4. Increase the sense of mutual responsibility for systems between team members.
5. Find better solutions to problems.
6. Improve the overall quality of the codebase.

2.2 How to Start with Code Reviews

Starting the process of running code reviews is like starting a new habit; it's difficult to know where to begin. There isn't a right way or a wrong way; however, there are generally two critical steps to ensure you get the best value from your code reviews. You should:

1. Define code conventions for your team or project.
2. Define the code review process for your team or project.

[1] https://www.itil.org.uk/what-is-itil

[2] https://www.iso.org/standard/70636.html

2.2.1 Define Code Conventions for Your Team or Project

Code conventions are a set of rules defining programming style and practices. By following the same set of conventions, your code will be more readable and understandable by members of your project or team.

With PowerShell, you're likely aware of the *noun-verb* naming convention for cmdlets, but there are other topics that your code conventions could cover, including:

- File organization.
- Naming of variables, functions, and methods.
- Indentation.
- Comments.
- Declarations.
- Statements.
- Using blank space.
- Capitalization.
- Code structure, and much more.

When working within a team, it's critical that you reach a consensus on how code is written. This consensus ensures that all code, no matter who writes it, follows a standard pattern or progression. This improves the readability and understandability of the code, making contributions and maintenance easier. Anyone in the team should be able to pick up another's code, understand it, review it, and change it.

It's important that you record the rules and practices you're following within your project, team, or organization. They should be discoverable, allowing anyone to read and understand what conventions they're to use when writing code.

By developing conventions, you're making the task of understanding code significantly faster, and dramatically reducing the time required for those who review code to understand it and then provide comments and feedback.

When it comes to PowerShell, developers are lucky that they can *stand on the shoulders of giants*. The PowerShell Best Practices and Style Guide[3] was developed by Don Jones, Matt Penny, Carlos Perez, Joel Bennett, and other members of the PowerShell Community. These best practices have been developed over the last decade and have become a quasi-standard within the PowerShell community.

It's recommended to fork the repository, then adapt and change the guidelines to meet the requirements of your project, team, or organization.

Once you've defined your code conventions, link to them from your README files or contribution guides for your projects.

[3] https://github.com/PoshCode/PowerShellPracticeAndStyle

2.2.2 Define the Code Review Process for Your Team or Project

When defining code review processes, always try to think of the *what*, *who*, and *when*.

- What code should be reviewed?
- Who should be involved in code reviews?
- When should code reviews be performed?

2.2.2.1 What Code Should Be Reviewed?

All code should be reviewed before it's merged into `main` (your main or *trunk* branch) and deployed into production. Code shouldn't be excluded from review based upon the author's experience, role in the team or organization, or their seniority.

Branch policies must be configured to prevent changes being directly pushed into sensitive branches, such as `main`. Code repositories utilize branch policies to protect essential branches (e.g `main` or `master`) from accidental/intentional changes.[4] [5]

2.2.2.2 Who Should Be Involved in Code Reviews?

Everyone in your team or project must be involved in code reviews. Being involved in code reviews is a great way to get experience and form a deeper understanding of a codebase. By involving junior team members, they're presented with the opportunity to learn from others, even if more senior members might re-review the code.

A great piece of advice is to ensure that you don't have a single person performing all the code reviews for your team. This can lead to a huge bottleneck and be a source of delays and pain for your team. It also increases the chances of mistakes making it into production. Remember, even the most senior team members make mistakes.

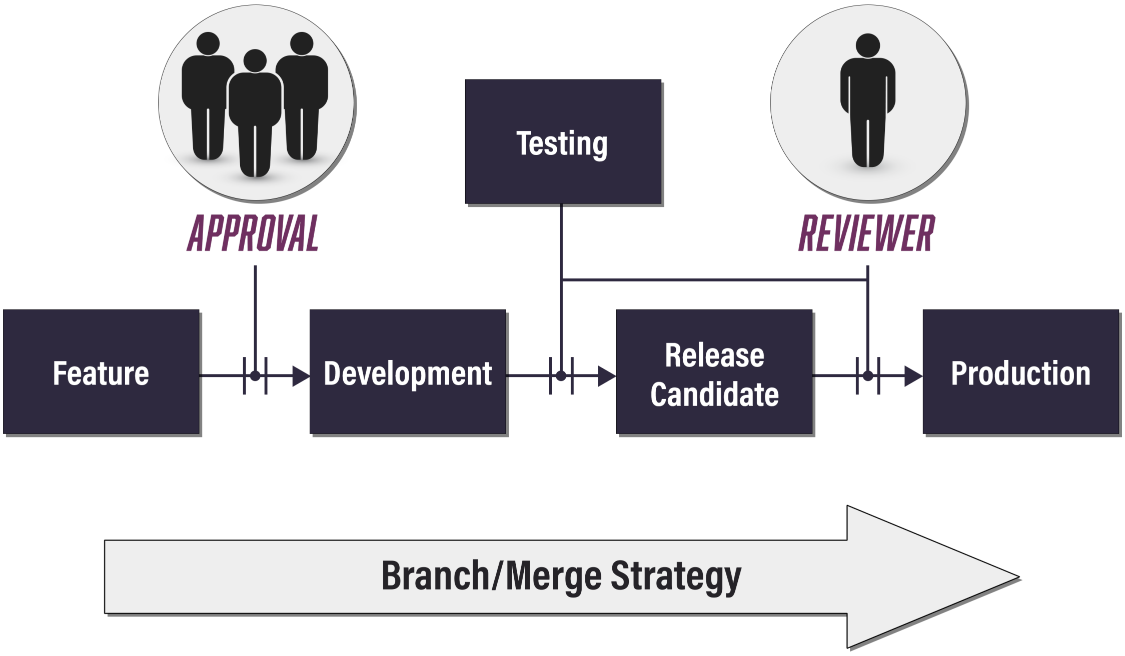
2.2.2.3 When Should Code Reviews Be Performed?

The common conception is that code reviews should be performed before code is pushed into production. This thinking is driven by the desire to protect production code and comes from change management processes like ITIL.

There is, however, an alternative perspective of when reviews should be performed. Many developers believe in moving processes—like reviews and security checks—as far to the left as possible, with code reviews performed when merging into a *production-like* environment.

[4] Microsoft. (2022, Apr. 30). *Branch policies and settings*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/azure/devops/repos/git/branch-policies?view=azure-devops&tabs=browser. [Accessed: Sep. 15, 2022].

[5] GitHub. (2022, Aug. 18). *About protected branches*. GitHub Docs. [Online]. Available: https://docs.github.com/en/repositories/configuring-branches-and-merges-in-your-repository/defining-the-mergeability-of-pull-requests/about-protected-branches. [Accessed: Sep. 15, 2022].



Code Review and Approval Flow

Consider a situation where you have development, testing, and production environments, with the testing environment being identical to production. In this situation, you could perform code reviews when code is pushed into the testing environment. Some might argue that code should be re-reviewed before being pushed into production. However, if the code hasn't changed, and all tests and validations passed in the first review, there's little value in an additional one. The exception here is when screening for hidden mistakes (that may only generate errors later) and edge-cases.

If the purpose of review before production is one of change control, apply rules on who can push changes to production, or who can approve these changes. You should then put controls within your deployment/release process (such as control gates).

To reduce the human effort spent on code reviews, it's recommended that code reviews aren't performed until after all build and testing tasks are completed. As the chapter discusses later, these tasks may include automated checks around code suitability and, as such, should pass before a human reviews the code.

2.3 Things to Consider When Performing a Code Review

Code reviews are like traditional change reviews that many operationally focused teams are familiar with. As a reviewer, the process is less about reviewing the actual lines of code, and more about those details *between the lines.*

When performing a code review, these are some useful questions to ask yourself. This isn't a comprehensive list, but these are common issues or questions that people encounter. Over time, you'll develop your own approach and criteria.

Questions to consider:

1. Why is this change being made? Does the change resolve an issue?
2. What's the context of the change within the bigger picture?
3. Does this change behave how the author intended? Are there potential situations that may cause the code to behave unexpectedly? Are there any unexpected outcomes to running the code?
4. Is the code overly complex? Could the issue be solved in a simpler way?
5. Has the change been tested appropriately? Do any tests need to be updated? Have all tests passed?
6. Are there appropriate comments? Are they clear and useful?
7. Has any comment-based help or related documentation been updated?
8. Does the code follow the team/project conventions?
9. Are there any potential security issues introduced with this change?
10. Are errors handled or thrown appropriately?

A *what-if?* approach is helpful here. Make educated guesses about any knock-on (*downstream*) effects that a change could cause. The more familiar you are with a codebase, the more accurately you can predict how a change will interact with existing code.

2.4 Code Review Best Practices

There are some practices that'll help you get the most value out of code reviews. These practices relate less to programmatic or code issues, which you should be on the lookout for, and more to ensuring that your overall code review process is streamlined and constructive.

2.4.1 Keep Your Changes Small

One of the most effective practices when making any change, bug fix, or a new feature is to keep the scope of changes as small as possible. Smaller pull requests can be faster to test, review, approve, and push into production, allowing teams to reduce the time for changes to move from development to production.

Two rules are often quoted for the size of code changes and reviews:

1. Each change or pull request should be for a single bug, issue, or feature.
2. Limit code changes to under 400 lines—under 200 is better still.

So, what's the driver of these rules? The processing ability of the human brain.

Ensuring a pull request solves a single bug, issue, or feature means a reviewer needs to keep track of only a single purpose for the code—think "I need to eat an apple" compared to "I need to eat an apple, tie my shoelaces, and sing at the same time."

There's also evidence that human performance in detecting issues and defects decreases markedly with larger code changes.[6]

Introducing new features to an existing code base, a new function in a PowerShell module, for instance, can often make it hard to follow these rules. One approach is to break up new features where you can. For instance, if a new feature requires two new functions, you might break those up into different pull requests. If one function depends upon the other, then you can work to get the dependency's review completed first. You could create a third pull request for changes to other functions that should interact with these new ones if needed.

Code reviews require significant attention to be spent, and it can be difficult for individuals to maintain their concentration for longer than 60 minutes. With interruptions and meetings, spending more than this may not be possible. There's some evidence that most defects can be found within 60 to 90 minutes.[7] If after an hour you've found several major defects that need to be addressed, consider that any other issues still in the code could be found in the next code review. They may also be resolved as part of resolutions for defects found already.

2.4.2 Provide Constructive Feedback

Providing and accepting feedback can be a difficult process. The process can be improved through these three rules:

1. Feedback provided should be constructive. The reason an issue has been raised should be clear.
2. Where possible, ask open-ended questions as they encourage deeper thought and understanding of issues raised. Don't tell them how to fix an issue, guide them to the correct solution.
3. Don't use strong or opinionated statements.

Consider the following code and review comments:

Example 1: Don't use strong or opinionated statements

```
# Review-Comment: Fix the name.
Function GetHelloWorld {
    # Review-Comment: Don't compare with $null this way.
    If ($MyVariable -eq $null) {

        $MyVariable = 'Hello World'
    }

    # Review-Comment: Don't display output using Write-Host.
    Write-Host $MyVariable
}
```

What problems do you see with this feedback? The comments:

[6] T. Baum, K. Schneider, A. Bacchelli. (2019). *Associating working memory capacity and code change ordering with code review performance.* Empir Software Eng, vol. 24, no. 4, pp. 1762–1798. DOI: 10.1007/s10664-018-9676-8.

[7] O. Baysal, O. Kononenko, R. Holmes, et al. (2016). *Investigating technical and non-technical factors influencing modern code review.* Empir Software Eng, vol. 21, no. 3, pp. 932-959. DOI: 10.1007/s10664-015-9366-8.

- Don’t describe what the issues identified are.
- Don’t provide any direction as to how to fix the identified issues.

Now consider:

Example 2: A better approach to review comments

```
# Review-Comment: Should the function's name follow the verb-noun syntax?
Function GetHelloWorld {

    # Review-Comment: Comparing like this can cause $null to be cast.
    #    You should instead use $null -eq $MyVariable to ensure $null is
    #         not cast.
    If ($MyVariable -eq $null) {
        $MyVariable = 'Hello World'
    }

    # Review-Comment: Is there a better way for output to be displayed?
    Write-Host $MyVariable
}
```

How’s this feedback more constructive? The comments:

- Identify what each issue is.
- Provide a direction as to how to solve each issue.
- Avoid strong opinions and orders

Don’t underestimate the impact that code reviews can have on people’s confidence. Remember to applaud and praise good solutions to problems. Not all comments need to be for issues, you can also leave praise.

> Several years ago, a junior developer in my team had spent several days working through a difficult bug fix. Several iterations of the code were needed before the issue could be solved. I remember how ecstatic they were when the only comment in the review was “Everything looks good, there are no issues. Great job!!”

Always remember that everyone has a different experience. While some may have in-depth automation knowledge, others may only be starting on their journey.

2.4.3 Balance Nit-Picks with Major Comments

Nit-picks are somewhat unimportant comments about code—they don’t prevent merging of the code or point out major errors. Having said this, nit-picks are still important feedback that the author of the code needs to fix. In practice, nit-picks are often related to code style.

For example, during a code review, you might see the following code:

Example 3: Misaligned hashtable

```
$MyHashtable = @{
    Name=''
    Description=''
    Usage=''
}
```

However, your team's style guide notes that values within a hash table should be aligned as:

Example 4: Aligned hashtable

```
$MyHashtable = @{
    Name        = ''
    Description = ''
    Usage       = ''
}
```

This issue doesn't affect the execution of the code; however, the code doesn't meet the code style expectations of the team.

As a reviewer, you need to keep a balance with nit-picking comments. No one wants to see 100 comments for 200 lines of code. Some recommendations are:

1. If you see a particular nit-pick repeated throughout the code, leave a single comment. Don't add a comment to every instance.
2. There can be value to marking and finding nit-picks. Consider prefixing comments with `[nit-pick]` so that requesters and other reviewers can quickly differentiate critical comments from non-critical nit-picks.

You can reduce nit-picks by using automated validation tools such as PSScriptAnalyzer[8] and testing suites like Pester[9]. You can learn more about Pester and testing in the Testing part of the book. The PowerShell extension within Visual Studio Code[10] also includes many settings that can aid with sticking to a particular coding style.

If you're struggling with the number of nit-picks within your team or project, it could be a sign that the tooling and standards maintained within the team need to be reviewed and discussed.

2.4.4 Create Pull Request Templates

Writing a good pull request description is a great way to help reviewers know what to expect when reviewing your code. They can aid in tracking tasks that need to be performed for every change, such as testing, updating tests, or updating documentation. Templates can help requesters create clear pull request descriptions and meet your organization, team, or project's standards.

[8] https://learn.microsoft.com/en-us/powershell/utility-modules/psscriptanalyzer/overview

[9] https://pester.dev/

[10] https://code.visualstudio.com/

Pull request templates are typically created at a repository level and written in Markdown. While templates can be placed in different locations within a repository, a good option is the `docs` folder, as this folder is supported by both GitHub and Azure DevOps.[11] [12]

Other common locations include the repository root, or a service-specific folder such as `.github` or `.azuredevops`. Many services also support additional templates by placing files in a folder, such as `.github/PULL_REQUEST_TEMPLATE/` or `.azuredevops/pull_request_template/`.

Here's an example of a basic pull request template for PowerShell code:

Example 5: Example pull request template with checklist

```markdown
# PR Summary

<!-- Summarize your PR between here and the checklist. -->

## PR Checklist

+ [ ] PSScriptAnalyzer has been run against all new/changed code.
+ [ ] Pester test cases have all passed.
+ [ ] Pester tests have been updated as required.
+ [ ] Comment-based help has been updated as required.
```

Completing the checklist provides an opportunity for the requester to stop and think about their code. Have they performed the necessary tests and checks? Are the tests all passing? Have they followed the defined style guide?

Pull request templates aren't only for the requester—they can be used by the reviewers as well. Reviewers may have several steps to be completed as part of the review; these could be included in the template and then updated or marked as completed by the reviewer.

2.4.5 When to Approve

When performing a code review, a common decision is whether a change should be approved, rejected, or left waiting for a response from whomever requested the change.

The general principle is that reviews shouldn't be marked as approved while there are any outstanding open-ended questions. You should be firm but flexible on this practice.

As a team or project formalizing your code review process, ensure that everyone agrees about when to approve, reject, or request changes or feedback. There shouldn't be confusion about why some changes are approved and others aren't.

2.4.6 Talk to Each Other

Communication is critical to successful code reviews. A review should be a two-way conversation between the requester and one or more reviewers.

[11] GitHub. (2022). *Creating a pull request template for your repository.* GitHub Docs. [Online]. Available: https://docs.github.com/en/communities/using-templates-to-encourage-useful-issues-and-pull-requests/creating-a-pull-request-template-for-your-repository. [Accessed: Aug. 08, 2022].

[12] Microsoft. (2022, Feb. 11). *Improve pull request descriptions using templates.* Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/azure/devops/repos/git/pull-request-templates. [Accessed: Aug. 12, 2022].

If you're the change requester, don't be afraid to ask questions about the feedback that's been provided to you. Don't think of reviews as a blocker to production; they're an opportunity for you to learn and improve your craft.

Don't rely only on text communication on your favorite source code platform. There's value in having face-to-face discussions, be they in-person or virtual. When a requester is new to a team or project, or there are a substantial number of questions and issues, try to provide—as a reviewer—the opportunity for dedicated time with the requester. This provides them with an opportunity to ask questions and more opportunities to share knowledge.

If you lead or manage a team, be vigilant that code reviews don't become a back-and-forth *slugging match*. This isn't productive and doesn't help anyone.

2.4.7 Use Automation

Automation is also key to code review productivity. There are plenty of ways automation can help with code reviews. Automation is exceptionally useful for detecting common errors and issues within code—the *low-hanging fruit*.

Any automated checks should occur before anyone performs a code review. If your automated tests and checks discover issues, address these before review.

Your continuous integration and continuous delivery (CI/CD) pipelines should include Pester[13] and PSScriptAnalyzer[14]. Pull request checks running these should succeed before a change is reviewed.

2.5 Tools to Help with Code Reviews

As with other programming languages, there are tools available to help you in ensuring the quality of code within PowerShell projects. These tools can be used as standalone tools or as part of automated CI/CD deployment processes.

2.5.1 PSScriptAnalyzer

PSScriptAnalyzer is a powerful module that can help you write better PowerShell code. It's a static code checker for PowerShell modules and scripts, which checks the quality of the code specified against a set of rules. These rules are based on best practices identified by the PowerShell Team and the community. Rules include checks for uninitialized variables, usage of sensitive types, the use of `Invoke-Expression`, and many more. PSScriptAnalyzer can also check for compatibility issues between PowerShell versions, and it also has some linting support.[15] If you aren't using PSScriptAnalyzer, you have really been missing out.

2.5.1.1 Using PSScriptAnalyzer

When you execute PSScriptAnalyzer using `Invoke-ScriptAnalyzer`, it'll check your specified PowerShell code using a default collection of built-in rules.

[13] https://pester.dev/

[14] https://learn.microsoft.com/en-us/powershell/utility-modules/psscriptanalyzer/overview

[15] Microsoft. (2022, Mar. 23). *PSScriptAnalyzer module.* Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/utility-modules/psscriptanalyzer/overview. [Accessed: Aug. 12, 2022].

Example 6: Using Invoke-ScriptAnalyzer

```
Invoke-ScriptAnalyzer -Path /path/to/script.ps1
```

PSScriptAnalyzer includes some pre-defined rule sets, which you can specify using the `-Settings` parameter.

Example 7: Invoke-ScriptAnalyzer with PSGallery rule set

```
Invoke-ScriptAnalyzer -Path /path/to/module/ -Settings PSGallery -Recurse
```

At the time of writing, the current pre-defined rule sets are:[16]

Settings Value	Description
`CmdletDesign`	Cmdlet-specific checks.
`CodeFormatting`	Code style checks. At the time of writing, this rule set uses *Stroustrup* style.[17] [18]
`CodeFormattingAllman`	Style checks based upon the Allman style.
`CodeFormattingOTBS`	Style checks for the OTBS (One True Brace Style) style.
`CodeFormattingStroustrup`	Style checks for the Stroustrup style. Currently equivalent to `CodeFormatting`.
`DSC`	Checks related to DSC (Desired State Configuration).
`PSGallery`	Includes the checks that are performed by the PowerShell Gallery when a module is submitted for inclusion into the gallery.[19]
`ScriptFunctions`	Checks for common issues in PowerShell script functions.
`ScriptSecurity`	Checks related to potential security issues within scripts.
`ScriptingStyle`	This focuses solely on style issues for scripts, including the usage of comment-based help and avoiding the use of `Write-Host`.

2.5.1.2 Customizing PSScriptAnalyzer Settings

There are often situations where you'll want to change the behavior of PSScriptAnalyzer and `Invoke-ScriptAnalyzer`. While you can specify and control the execution of `Invoke-`

[16] Microsoft. (2021). *PSScriptAnalyzer - Engine/Settings files*. PowerShell/PSScriptAnalyzer on GitHub. [Online]. Available: https://github.com/PowerShell/PSScriptAnalyzer/tree/5797a04a61228eb3a64287d56413a035d25191d5/Engine/Settings/. [Accessed: Aug. 12, 2022].

[17] Microsoft. (2021, Jan. 06). *PSScriptAnalyzer - CodeFormatting.psd1*. PowerShell/PSScriptAnalyzer on GitHub. [Online]. Available: https://github.com/PowerShell/PSScriptAnalyzer/blob/master/Engine/Settings/CodeFormatting.psd1. [Accessed: Aug. 12, 2022].

[18] Microsoft. (2021, Jan. 06). *PSScriptAnalyzer - CodeFormattingStroustrup.psd1*. PowerShell/PSScriptAnalyzer on GitHub. [Online]. Available: https://github.com/PowerShell/PSScriptAnalyzer/blob/master/Engine/Settings/CodeFormattingStroustrup.psd1. [Accessed: Aug. 12, 2022].

[19] Microsoft. (2022, Mar. 23). *Using PSScriptAnalyzer*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/utility-modules/psscriptanalyzer/using-scriptanalyzer. [Accessed: Aug. 12, 2022].

ScriptAnalyzer with parameters, you can also define these within a .psd1 settings file. This is better for long-term maintainability in your code review process. You then call Invoke-ScriptAnalyzer, specifying the path to the file using the -Settings parameter.

Example 8: Invoke-ScriptAnalyzer with custom settings

```
Invoke-ScriptAnalyzer -Path MyScript.ps1 -Settings PSScriptAnalyzerSettings.psd1
```

2.5.1.2.1 Including and Excluding Rules

You can specify which rules to include using the IncludeRules element.

Example 9: Including rules in a custom settings file

```
# PSScriptAnalyzerSettings.psd1
@{
    IncludeRules = @(
        'PSAvoidUsingPlainTextForPassword'
        'PSAvoidUsingConvertToSecureStringWithPlainText'
    )
}
```

You can also exclude rules using the ExcludeRules element.

Example 10: Excluding rules in a custom settings file

```
# PSScriptAnalyzerSettings.psd1
@{
    ExcludeRules = @(
        'PSAvoidUsingCmdletAliases'
        'PSAvoidUsingWriteHost'
    )
}
```

IncludeRules and ExcludeRules both support the use of wildcards. In the example below, only rules starting with PSDSC (the included DSC rules) are included.

Example 11: Including rules using a wildcard

```
# PSScriptAnalyzerSettings.psd1
@{
    IncludeRules = @('PSDSC*')
}
```

Every PSScriptAnalyzer rule has an associated severity, and you can specify which rules are included based on their severity.[20] The example below includes only rules whose alert has a severity of error or warning.

[20] Microsoft. (2022, Mar. 23). *PSScriptAnalyzer rules and recommendations.* Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/utility-modules/psscriptanalyzer/rules-recommendations. [Accessed: Aug. 12, 2022].

Example 12: Including rules based on severity level

```
# PSScriptAnalyzerSettings.psd1
@{
    Severity = @('Error', 'Warning')
}
```

2.5.1.2.2 Including Custom Written Rules

You can also write your own rules for PSScriptAnalyzer as PowerShell modules, and include these in your PSScriptAnalyzer settings.[21] The `CustomRulePath` element provides a mechanism to specify the location of PowerShell modules containing your custom rules.

Example 13: Including custom written rules

```
# PSScriptAnalyzerSettings.psd1
@{
    # Import 2 custom rules modules
    CustomRulePath = @(
        '.\output\RequiredModules\DscResource.AnalyzerRules'
        '.\tests\QA\AnalyzerRules\SqlServerDsc.AnalyzerRules.psm1'
    )

    # Include the custom rules
    IncludeRules   = @('Measure-*')
}
```

If you want to include the default rules together with your custom rules, you can set the `IncludeDefaultRules` element to `$true`.

Example 14: Including default rules and custom rules

```
# PSScriptAnalyzerSettings.psd1
@{
    CustomRulePath      = @(
        '.\output\RequiredModules\DscResource.AnalyzerRules'
        '.\tests\QA\AnalyzerRules\SqlServerDsc.AnalyzerRules.psm1'
    )

    IncludeRules        = @('Measure-*')

    IncludeDefaultRules = $true
}
```

2.5.1.2.3 Defining Optional Rule Settings

Some rules have definable behavior, allowing you to change what they'll look for based on your own requirements. `PSUseConsistentIndentation` is an example of one such rule. This rule allows you to specify what kind of indentation is used (`space` or `tab`), the indentation size in the number of space characters, and how indentation applies to pipelines.

You can define these rules through settings using the `Rules` element:

[21] Microsoft. (2022, Mar. 23). *Creating custom rules*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/utility-modules/psscriptanalyzer/create-custom-rule. [Accessed: Aug. 12, 2022].

Example 15: Specifying optional rule settings

```
#PSScriptAnalyzerSettings.psd1
@{
    IncludeRules = @(
        'PSUseConsistentIndentation'
    )

    Rules = @{
        PSUseConsistentIndentation = @{
            Enable              = $true
            Kind                = 'space'
            PipelineIndentation = 'IncreaseIndentationForFirstPipeline'
            IndentationSize     = 4
        }
    }
}
```

2.5.2 PowerShell Extension for Visual Studio Code

The PowerShell Extension[22] for Visual Studio Code provides a rich PowerShell authoring experience within Visual Studio Code. The extension includes support for syntax highlighting, code snippets, PSScriptAnalyzer, and much more.

Through the configuration of the PowerShell extension and Visual Studio Code, you can ensure that common code formatting issues are identified earlier in the development process, before code review. This improves code quality and reduces the effort needed for code reviews.

Visual Studio Code and extension settings are defined in the `settings.json` file, at either a user or workspace level. User-level settings apply globally to every Visual Studio Code window you open as that user. Workspace-level settings are stored and applied only when that workspace is opened. Workspace settings override those found in user settings.[23]

The advantage of workspace settings is that they apply to a specific project and are shared across all developers on a project. A workspace is the root folder of your project, with a `settings.json` file found in the `.vscode` subfolder.

If you wish to enforce that a particular setting has its default value, even when the setting is configured differently at the user level, add it to the workspace `settings.json` file.

Let's look at an example of a workspace `settings.json` file:

[22] https://marketplace.visualstudio.com/items?itemName=ms-vscode.PowerShell

[23] Microsoft. (2022, Aug. 04). *User and Workspace Settings*. Visual Studio Code Docs. [Online]. Available: https://code.visualstudio.com/docs/getstarted/settings. [Accessed: Aug. 12, 2022].

Example 16: An example VSCode `settings.json` file

```
{
    // Recommends that PowerShell extension is installed if missing.
    "recommendations": [
        "ms-vscode.powershell"
    ],

    // Number of spaces for tab stops.
    // Default: 4
    "editor.tabSize": 4,

    // Uses spaces instead of horizontal tabs.
    // Default: true
    "editor.insertSpaces": true,

    // Trims trailing spaces when you save a file.
    // Default: false
    "files.trimTrailingWhitespace": true,

    // Adds a space before and after the pipeline operator ('|') if missing.
    // Default: True
    "powershell.codeFormatting.addWhitespaceAroundPipe": true,

    // Aligns assignment statements in a hash table or a DSC Configuration.
    // Default: True
    "powershell.codeFormatting.alignPropertyValuePairs": true,

    // Replaces aliases with their aliased name.
    // Default: False
    "powershell.codeFormatting.autoCorrectAliases": true,

    // Doesn't reformat one-line code blocks, such as:
    //  "if (...) {...} else {...}".
    // Default: True
    "powershell.codeFormatting.ignoreOneLineBlock": false,

    // Adds a newline (line break) after a closing brace.
    // Default: True
    "powershell.codeFormatting.newLineAfterCloseBrace": true,

    // Adds a newline (line break) after an open brace.
    // Default: True
    "powershell.codeFormatting.newLineAfterOpenBrace": true,

    // Places open brace on the same line as its associated statement.
    // Default: True
    "powershell.codeFormatting.openBraceOnSameLine": true,

    // Multi-line pipeline style settings.
    // Default: NoIndentation
    "powershell.codeFormatting.pipelineIndentationStyle": "NoIndentation",

    // Sets the code formatting options to follow the given indent style
    //  in a way that's compatible with PowerShell Syntax.
    // Default: Custom
    "powershell.codeFormatting.preset": "OTBS",

    // Trims extraneous spaces (more than 1 space character) before and
    //  after the pipeline operator ('|').
    // Default: False
    "powershell.codeFormatting.trimWhitespaceAroundPipe": true,

    // Use single quotes if a string is not interpolated
    //  and its value does not contain a single quote.
    // Default: False
    "powershell.codeFormatting.useConstantStrings": true,

    // Use correct casing for cmdlets.
    // Default: False
    "powershell.codeFormatting.useCorrectCasing": true,

    // Adds a space after a separator (',' and ';').
    // Default: True
    "powershell.codeFormatting.whitespaceAfterSeparator": true,

    // Adds spaces before and after an operator ('=', '+', '-', etc.).
    // Default: True
    "powershell.codeFormatting.whitespaceAroundOperator": true,

    // Adds a space between a keyword and its associated opening brace.
    // Default: True
    "powershell.codeFormatting.whitespaceBeforeOpenBrace": true,

    // Adds a space between a keyword (if, elseif, while, switch, etc.)
    //  and its associated conditional expression.
    // Default: True
    "powershell.codeFormatting.whitespaceBeforeOpenParen": true,

    // Removes redundant spaces between parameters.
    // Default: False
    "powershell.codeFormatting.whitespaceBetweenParameters": true,

    // Adds a space after an opening brace ('{')
    //  and before a closing brace ('}').
    // Default: True
    "powershell.codeFormatting.whitespaceInsideBrace": true,

    // Enables real-time script analysis from PowerShell Script Analyzer.
    // Default: True
    "powershell.scriptAnalysis.enable": true
}
```

Settings elements that begin with `powershell.` are for the PowerShell extension specifically.

In the example, you can see that the PowerShell extension is recommended and a specific brace and indentation style has been specified, as have other code format options.

You can change many of these settings, including extensions settings, in the Visual Studio Code interface. In the toolbar, navigate to `File` → `Preferences` → `Settings` or use the keyboard shortcut `Ctrl + ,` (comma).

The user-level `settings.json` file can be found in the following locations, depending on your platform:

- **Windows**: `%APPDATA%\Code\User\settings.json`.
- **Linux**: `$HOME/.config/Code/User/settings.json`.
- **macOS**: `$HOME/Library/Application Support/Code/User/settings.json`.

`%APPDATA%` is generally `C:\Users\<username>\AppData\Roaming`, and `$HOME` is generally `/home/<username>`.

2.6 Further Reading

- The PowerShell Best Practices and Style Guide—GitHub[24]
- PSScriptAnalyzer Overview—Microsoft Docs[25]
- PSScriptAnalyzer—PowerShell Gallery[26]
- Creating a GitHub Pull Request Template—GitHub Docs[27]
- Creating an Azure DevOps Pull Request Template—Microsoft Docs[28]
- Pester Testing Suite[29]
- Visual Studio Code[30]
- Using Visual Studio Code for PowerShell Development—Microsoft Docs[31]
- PowerShell Extension for VS Code—Visual Studio Marketplace[32]
- IT Infrastructure Library Overview—ITIL[33]
- ISO 20000 Standard—ISO[34]

[24] https://github.com/PoshCode/PowerShellPracticeAndStyle

[25] https://learn.microsoft.com/en-us/powershell/utility-modules/psscriptanalyzer/overview

[26] https://www.powershellgallery.com/packages/PSScriptAnalyzer/

[27] https://docs.github.com/en/communities/using-templates-to-encourage-useful-issues-and-pull-requests/creating-a-pull-request-template-for-your-repository

[28] https://learn.microsoft.com/en-us/azure/devops/repos/git/pull-request-templates

[29] https://pester.dev/

[30] https://code.visualstudio.com/

[31] https://learn.microsoft.com/en-us/powershell/scripting/dev-cross-plat/vscode/using-vscode

[32] https://marketplace.visualstudio.com/items?itemName=ms-vscode.PowerShell

[33] https://www.itil.org.uk/what-is-itil

[34] https://www.iso.org/standard/70636.html

II PowerShell Testing

> *"Testing leads to failure and failure leads to understanding."* — Burt Rutan

First, let's make something clear:

"If you write code, you're a developer."

Yes, this includes PowerShell. Anybody can write code; however, for it to be *good code*, it must be *maintainable* and *testable*. This section will explore the Arrange, Act, and Assert framework, Mocking, Unit Testing, and Parameterized Testing. Chapters will include nuances such as test framework version, mocking applications, .NET objects, and project structure layout.

3. The AAA Approach

This chapter aims to explain using demos how to apply the Arrange, Act, and Assert (AAA) principles when using the Pester testing framework. So what's AAA, and what's Pester?

3.1 Arrange, Act, and Assert

AAA stands for Arrange, Act, and Assert. This is a testing pattern where your tests are broken into three sections. Each section is only responsible for itself.

3.1.1 Arrange

Arrange is where the test conditions are configured and set up. Any test variables or required mocked systems can be created here if needed.

3.1.2 Act

Act is the process of invoking the actual test itself. This would call the PowerShell function being tested directly and store the result *(if there is one)* in a variable.

3.1.3 Assert

Assert is where we check the test results or another condition to determine if it's a pass or a fail. If a fail is detected, then an error should be logged with as much information as possible.

3.1.4 Benefits of the AAA Approach

This pattern is clear and straightforward, which are two characteristics sometimes lacking in code these days. Especially for test code, when project time and costs overrun, the first components that are scaled back or even dropped are automated testing and documentation.

3.2 Pester 5.0

Pester[1] is considered the de facto testing framework for PowerShell. The maintainer is Jakub Jareš[2], and it has been around for a few years now. There have been several major version releases, which come with both breaking changes and continual improvements.

[1] https://github.com/pester/Pester

[2] https://github.com/sponsors/nohwnd

If you are new to Pester, version 5.0 is an ideal version to start with, as the project has had ample time to mature and the process and documentation are of a high standard.

If you have previously used earlier versions of Pester, it's recommended to review the 5.0 documentation for the list of changes to see what's required[3] to bring your test projects up to date. For this chapter, we're using Pester version 5.3.0, which is current at the time of writing.

3.2.1 Pester Installation

To install Pester on your system, use the following commands:

This command is optional but generally recommended. Trusting the official PowerShell Gallery[4] for your module installations will save hassle and time later on when installing other modules.

```
Set-PSRepository -Name PsGallery -InstallationPolicy Trusted
```

Install the Pester module.

```
Install-Module -Name Pester
```

Import the Pester module.

```
Import-Module -Name Pester
```

Get a list of the commands in the module to confirm everything is all working as expected.

```
Get-Command -Module Pester
```

3.3 The Star Wars API Example

The example code will be a wrapper around a publicly available Star Wars Data API, which will be hopefully more interesting than the typical calculator example.

[3] https://pester.dev/docs/usage/importing-tested-functions#migrating-from-pester-v4

[4] https://www.powershellgallery.com/

3.3.1 So How Does It Work?

A public website is available here: https://mc-starwars-data.azurewebsites.net[5].
This website can query a range of Star Wars data, such as films, people, and planets. While this API doesn't currently contain a comprehensive list of data, it's suitable for demonstration purposes.

The example script wraps this API and provides helper functions to search and obtain detailed data quickly. The example contains simple, complex, and mocked Pester tests.

PowerShell modules can be tested with Pester, but this example focused on testing a script file for simplicity.

This example code is available via a GitHub repo[6].
Users are welcome to clone and experiment with the repo to assist with their Pester learning.

The example script is in a file called `StarWarsData.ps1`. Because we're utilizing features introduced in PowerShell version 7, the start of the file contains the following line.

```
#Requires -Version 7.0
```

The `#Requires` statement ensures that users who run the code have the correct version of PowerShell installed.[7]

3.3.2 Example Code

There is a function for calling the API directly. You should isolate the calling of external APIs to their own function, so if there are changes (like a new URL to a slightly different API), all changes will be isolated to the function level.

Example 1: A function to interface with the Star Wars API

```
$swApiUrl = 'https://mc-starwars-data.azurewebsites.net'
function Invoke-StarWarsApi
{
    param (
        [Parameter(Mandatory)]
        [ValidateSet('Planets', 'Films', 'People')]
        [string] $ObjectType,

        [int] $id = -1
    )
    try {
        $suffix = $id -ne -1 ? "?id=$id" : ""
        $path = "$($objectType.ToLower())$suffix"

        $output = Invoke-RestMethod -Uri "$swApiUrl/api/$path" -Method GET
        Write-Output $output
    }
    catch {
```

[5] https://mc-starwars-data.azurewebsites.net

[6] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/tree/main/Edition-01/Starwars-Demo/

[7] Microsoft. (2022, Mar. 18). *About Requires (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_requires. [Accessed: Apr. 27, 2022].

```
        $msg = "Error calling $swApiUrl/api/$path. $($_.Exception.Message)"
        Write-Host $msg -f Red
        Write-Output $null
    }
}
```

There are some sample search functions available that call `Invoke-StarWarsApi`. Only a subset of the API functionality is covered.

You can find the code from Examples 2 and 3 in the StarWarsData.ps1[8] file of the Extras repository for this book on GitHub.

Example 2: Three search functions that call the API interface function

```
function Search-SWPerson {
    param (
        [Parameter(Mandatory)]
        [string] $Name
    )
    # load all the people
    $response = Invoke-StarWarsApi -objectType People
    # filter on the name
    $results = $response | Where-Object name -like "*$Name*"

    if ($null -eq $results) {
        Write-Output @{ Error = "No person results found for '$Name'."}
    }
    else {
        # return all matches with some properties
        $personDetails = $results | ForEach-Object {
            Invoke-StarWarsApi -objectType People -id $_.id
        }

        Write-Output $personDetails | Select-Object @{
            Name = "id";
            Expression = { $_.id}
        }, name, gender, height,
        @{
            Name = "weight"
            Expression = {$_.mass}
        }
    }
}
```

[8] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/blob/main/Edition-01/Starwars-Demo/src/StarWarsData.ps1

```
function Search-SWPlanet {
    param (
        [Parameter(Mandatory)]
        [string] $Name
    )
    # load all the planets
    $response = Invoke-StarWarsApi -objectType Planets
    # filter on the name
    $results = $response | Where-Object name -like "*$Name*"

    if ($null -eq $results) {
        Write-Output @{ Error = "No planet results found for '$Name'."}
    }
    else {
        $planetDetails = $results | ForEach-Object {
            Invoke-StarWarsApi -objectType Planets -id $_.id
        }
        # return all matches with some attributes
        Write-Output $planetDetails | Select-Object @{
            Name = "id";
            Expression = {$_.id}
        },
        name,
        population,
        diameter,
        terrain
    }
}
```

```
function Search-SWFilm {
    param (
        [Parameter(Mandatory)]
        [string] $Name
    )
    # load all the films (currently does not include the new trilogy)
    $response = Invoke-StarWarsApi -objectType Films
    # filter on the name
    $results = $response | Where-Object title -like "*$Name*"

    if ($null -eq $results) {
        Write-Output @{ Error = "No film results found for '$Name'."}
    }
    else {
        # return all matches with some attributes
        $filmDetails = $results | ForEach-Object {
            Invoke-StarWarsApi -objectType Films -id $_.id
        }
        Write-Output $filmDetails | Select-Object @{
            Name="id";
            Expression = { $_.id}
        },
        title,
        director,
        release_date,
        characters,
        planets
    }
}
```

There is a single function that wraps multiple calls and returns a composite object.

Example 3: A more complex function that builds composite objects using multiple API calls

```
function Get-SWPerson {
    param (
        [Parameter(Mandatory)]
        [int] $Id
    )
    # get the person
    $person = Invoke-StarWarsApi -objectType People -id $Id

    if ($null -eq $person)
    {
        Write-Output @{
            Error = "Unable to find a person record given Id: $Id"
        }
    }
    else {
        # get the homeworld planet and the films
        $planet = Invoke-StarWarsApi -objectType Planets -id $person.homeworld
        $films = Invoke-StarWarsApi -objectType Films

        # get detailed info of all films
        $filmDetails = $films | ForEach-Object {
            Invoke-StarWarsApi -objectType Films -id $_.id
        }

        # build the result object as a mix of all the data returned
        $result = [PSCustomObject]@{
            Name = $person.Name
            BodyType = $person |
                Select-Object height, mass, gender, skin_color, eye_color
            HomeWorld = $planet |
                Select-Object name, population, gravity, terrain
            Films = $filmDetails |
                Where-Object people -contains $person.id |
                Select-Object title, director, release_date
        }
        Write-Output $result
    }
}
```

3.3.3 Example Code Output

When executing `Search-SWPerson`, observe the output:

Example 4: Calling the Search-SWPerson function

```
Search-SWPerson -name walker
```

```
id     : 1
name   : Luke Skywalker
gender : male
height : 172
weight : 77

id     : 9
name   : Anakin Skywalker
gender : male
height : 188
weight : 84

id     : 27
name   : Shmi Skywalker
gender : female
height : 163
weight : unknown
```

When executing `Get-SWPerson` using the ID from **Anakin Skywalker** above, it returns an object with multiple properties:

> The output is formatted slightly to fit in this book correctly.

Example 5: Calling the Get-SWPerson function with the ID for Anakin Skywalker

```
Get-SWPerson -Id 9 | Format-List
```

```
Name      : Anakin Skywalker
BodyType  : @{height=188; mass=84; gender=male; skin_color=fair; eye_color=blue}
HomeWorld : @{name=Tatooine; population=200000;
              gravity=1 standard; terrain=desert}
Films     : {@{title=The Phantom Menace; director=George Lucas;
                release_date=1999-05-19},
             @{title=Attack of the Clones; director=George Lucas;
                release_date=2002-05-16},
             @{title=Revenge of the Sith; director=George Lucas;
                release_date=2005-05-19}}
```

3.4 Pester Tests

The default standard for Pester tests is to be defined in a file in the same folder as the script or module itself. The file extension should end with `.Tests.ps1`.[9] The below sample would be in a file called `StarWarsData.Simple.Tests.ps1`.

[9] Pester Team. (2021, Oct. 12). *File placement and naming*. Pester Docs. [Online]. Available: https://pester.dev/docs/usage/file-placement-and-naming. [Accessed: Apr. 27, 2022].

3.4.1 Simple Tests

Below is the contents for the simple tests.

You can find the code from Example 6 in the StarWarsData.Simple.Tests.ps1[10] file of the Extras repository for this book on GitHub.

Example 6: Simple tests for the Search-SWPerson function

```
# Arrange
BeforeAll {
    . $PSCommandPath.Replace('.Simple.Tests.ps1','.ps1')
}

Describe 'Search-SWPerson' -Tag 'Unit' {
    It 'Returns a single match' {
        # Arrange
        $testName = 'Vader'

        # Act
        $result = Search-SWPerson -Name $testName

        # Assert
        $result.Count | Should -Be 1
        $result.name | Should -BeLike "*$testName*"
    }
    It 'Returns no matches' {
        # Arrange
        $testName = 'Invalid'

        # Act
        $result = Search-SWPerson -Name $testName

        # Assert
        $result.Error | Should -Be "No person results found for '$testName'."
    }
    It 'Returns multiple matches' {
        # Arrange
        $testName = 'walker'

        # Act
        $result = Search-SWPerson -Name $testName

        # Assert
        $result.Count | Should -BeGreaterThan 1
        $result.Name -like "*$testName*" | Should -HaveCount $result.Count
    }
}
```

The comments above detail which parts of the script correspond to the Arrange, Act, and Assert.

The **Arrange** sections do the following:

- Load the script file into memory so Pester can call it

[10] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/blob/main/Edition-01/Starwars-Demo/src/StarWarsData.Simple.Tests.ps1

- Define any test data needed for each test

The **Act** sections call the function `Search-SWPerson`, which is the one being tested.

The **Assert** sections utilize various Pester commands to confirm that the result of the Act section is as expected. The `Should` command has many parameters like `-Be`, `-BeLike`, or `-HaveCount` that make the assertion code read like English.
See this help page[11] for more details.

Simple tests like these are ideal, as they're straightforward to follow, which reduces the time taken for developers to understand the test process.

3.4.2 Pester Verbosity

The default output verbosity for running Pester tests is **Normal**.[12] This default value will display the Pester files that are executed and will only include test details if they fail. If you prefer to have more verbose logging that shows all test results regardless of whether they passed or failed, then include the parameter `-Output Detailed` with the `Invoke-Pester` command.

Alternatively, if Visual Studio Code triggers the tests, the Pester plugin settings define the default Output verbosity value. This value can be updated to the preferred verbosity.[13]

3.4.3 Simple Test Output

The output of the tests is:

Example 7: Output from the simple tests in example 6

```
Pester v5.3.0

Starting discovery in 1 files.
Discovery found 6 tests in 4ms.
Running tests.

Running tests from 'StarWarsData.Simple.Tests.ps1'
Describing Search-SWPerson
  [+] Returns a single match 730ms (729ms|1ms)
  [+] Returns no matches 133ms (132ms|1ms)
  [+] Returns multiple matches 594ms (592ms|2ms)

Describing Search-SWPlanet
  [+] Returns a single match 676ms (671ms|5ms)
  [+] Returns no matches 132ms (130ms|1ms)
  [+] Returns multiple matches 429ms (427ms|1ms)
Tests completed in 2.75s
Tests Passed: 6, Failed: 0, Skipped: 0 NotRun: 0
```

[11] https://pester.dev/docs/commands/Should

[12] Pester Team. (2021, Apr. 17). *Invoke-Pester - Output*. Pester Docs. [Online]. Available: https://pester.dev/docs/commands/Invoke-Pester#-output. [Accessed: Apr. 27, 2022].

[13] Pester Team. (2021, May. 15). *VSCode*. Pester Docs. [Online]. Available: https://pester.dev/docs/usage/vscode. [Accessed: Apr. 27, 2022].

3.4.4 Mocked Tests

Pester tests that call external APIs are great when the external APIs are operational. This means you are testing against a real server endpoint, ensuring that your scripts are working correctly. But what about when they're not?

Mocking in Pester gives us the ability to run tests when external APIs aren't available.

In the script below, the `Invoke-StarWarsApi` function is mocked out. It's defined four times and returns different data objects based on the input parameters `objectType` and `id`. Defining several ParameterFilter[14] options allows for detailed customization in the mocked responses.

You can find the code from Example 8 in the StarWarsData.Mocked.Tests.ps1[15] file of the Extras repository for this book on GitHub.

Example 8: Tests for the Search-SWPerson function that use mocking to simulate the API

```
# Arrange
BeforeAll {
    . $PSCommandPath.Replace('.Mocked.Tests.ps1','.ps1')

    Mock Invoke-StarWarsApi {
        $output1 = [PSCustomObject]@{
            id     = 4
            name   = 'Darth Vader'
            gender = 'male'
            height = '202'
            weight = '136'
        }
        $output2 = [PSCustomObject]@{
            id     = 1
            name   = 'Luke Skywalker'
            gender = 'male'
            height = '172'
            weight = '77'
        }
        $output3 = [PSCustomObject]@{
            id     = 9
            name   = 'Anakin Skywalker'
            gender = 'male'
            height = '188'
            weight = '84'
        }
        Write-Output @($output1, $output2, $output3)
    } -Verifiable -ParameterFilter { $objectType -eq 'People'}
```

[14] https://pester.dev/docs/commands/Mock#-parameterfilter

[15] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/blob/main/Edition-01/Starwars-Demo/src/StarWarsData.Mocked.Tests.ps1

```
    Mock Invoke-StarWarsApi {
        $output = [PSCustomObject]@{
            height     = '172'
            mass       = '77'
            hair_color = 'blonde'
            skin_color = 'fair'
            eye_color  = 'blue'
            birth_year = '19BBY'
            gender     = 'male'
            name       = 'Luke Skywalker'
            homeworld  = 1
            id         = 1
        }
        Write-Output @($output)
    } -Verifiable -ParameterFilter { $objectType -eq 'People' -and $id -eq 1}

    Mock Invoke-StarWarsApi {
        $output = [PSCustomObject]@{
            height     = '202'
            mass       = '136'
            hair_color = 'none'
            skin_color = 'white'
            eye_color  = 'yellow'
            birth_year = '41.9BBY'
            gender     = 'male'
            name       = 'Darth Vader'
            homeworld  = 1
            id         = 4
        }
        Write-Output @($output)
    } -Verifiable -ParameterFilter { $objectType -eq 'People' -and $id -eq 4}

    Mock Invoke-StarWarsApi {
        $output = [PSCustomObject]@{
            height     = '188'
            mass       = '84'
            hair_color = 'blonde'
            skin_color = 'fair'
            eye_color  = 'blue'
            birth_year = '41.9BBY'
            gender     = 'male'
            name       = 'Anakin Skywalker'
            homeworld  = 1
            id         = 9
        }
        Write-Output @($output)
    } -Verifiable -ParameterFilter { $objectType -eq 'People' -and $id -eq 9}
```

```
Describe 'Search-SWPerson' -Tag 'Unit', 'Mocked' {
    It 'Returns a single match' {
        # Arrange
        $testName = 'Vader'

        # Act
        $result = Search-SWPerson -Name $testName

        # Assert
        $result.Count | Should -Be 1
        $result.name | Should -BeLike "*$testName*"
    }
    It 'Returns no matches' {
        # Arrange
        $testName = 'Invalid'

        # Act
        $result = Search-SWPerson -Name $testName

        # Assert
        $result.Error | Should -Be "No person results found for '$testName'."
    }
    It "Returns multiple matches" {
        # Arrange
        $testName = 'walker'

        # Act
        $result = Search-SWPerson -Name $testName

        # Assert
        $result.Count | Should -BeGreaterThan 1
        $result.Name -like "*$testName*"| Should -HaveCount $result.Count
    }
}
```

The `Invoke-Pester` command is used to run Pester Tests. For More information, refer to: 5.4.6.2 Command Line in Unit Testing.

The tests above are tagged with both **Unit** and **Mocked**, so they can be filtered if required. When you run `Invoke-Pester`, you can provide a `-Tag <name>` parameter to filter which tests are run based on the tag.[16]

For the script above, if you wanted to run just mocked tests, the command line would be something like:

```
Invoke-Pester -Path ./StarWarsData.*.Tests.ps1 -Tag Mocked -Output Detailed
```

Any configuration of Mocked functions in Pester would be in the **Arrange** section of AAA. Care needs to be taken when mocking functions, as they will need to be maintained if the source system they're mocking gets updated over time.

3.4.5 Mocked Test Output

Mocked tests have an additional advantage of speed, as shown below in the "Tests Completed in" line when running the same tests against the actual API and a mocked API.

Real API:

```
Tests completed in 6.69s
Tests Passed: 6, Failed: 0, Skipped: 0 NotRun: 0
```

Mocked API:

[16] Pester Team. (2022, Jun. 19). *Tags*. Pester Docs. [Online]. Available: https://pester.dev/docs/usage/tags. [Accessed: Sep. 04, 2022].

```
Tests completed in 172ms
Tests Passed: 6, Failed: 0, Skipped: 0 NotRun: 0
```

3.4.6 Complex Tests

There are times when more complex Pester tests are warranted. This includes when there is a repeatable set of tests data to pass through the same test. This can rapidly test different parts of the function with minimal coding.

Below is a sample script that uses defined objects containing properties for the test's *arrange* and *assertion* stages. As with previous sample scripts, the comments indicate which areas of the script are Arrange, Act, or Assert sections.

You can find the code from Example 9 in the StarWarsData.Complex.Tests.ps1[17] file of the Extras repository for this book on GitHub.

Example 9: More complex tests covering more functions, using test cases to generate multiple tests

```
# Arrange
BeforeAll {
    . $PSCommandPath.Replace('.Complex.Tests.ps1','.ps1')
}

Describe 'Search-SWFilm' -Tag 'Unit' {
    $itName = "Returns film with release date '<year>' & director " +
    "'<director>' given title fragment '<name>'"

    It $itName -TestCases @(
        # Arrange
        @{
            name = 'Phantom'
            year = '1999-05-19'
            director = 'George Lucas'
        }
        @{
            name = 'Empire'
            year = '1980-05-17'
            director = 'Irvin Kershner'
        }
        @{
            name = 'Return'
            year = '1983-05-25'
            director = 'Richard Marquand'
        }
    )  {
        # Act
        $result = Search-SWFilm -name $name

        # Assert
        $result.Count | Should -Be 1
        $result.title | Should -BeLike "*$name*"
        $result.release_date | Should -Be $year
        $result.director | Should -Be $director
    }
}
```

[17] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/blob/main/Edition-01/Starwars-Demo/src/StarWarsData.Complex.Tests.ps1

```
Describe 'Get-SWPerson' -Tag 'Unit' {
    $itName = "Returns person metadata for '<fullname>' with gender " +
    "'<gender>', eye colour '<eyeColour>' & film count of <filmCount>"

    It $itName -TestCases @(
        # Arrange
        @{
            name = 'maul'
            fullName = 'Darth Maul'
            gender = 'male'
            eyeColour = 'yellow'
            homeWorld = 'Dathomir'
            filmCount = 1
        }
        @{
            name = 'luke'
            fullName = 'Luke Skywalker'
            gender = 'male'
            eyeColour = 'blue'
            homeWorld = 'Tatooine'
            filmCount = 6
        }
        @{
            name = 'mothma'
            fullName = 'Mon Mothma'
            gender = 'female'
            eyeColour = 'blue'
            homeWorld = 'Chandrila'
            filmCount = 1
        }
    )  {
        # Act
        $result = Search-SWPerson -name $name
        $result.Count | Should -Be 1
        $details = Get-SWPerson -Id $result.Id

        # Assert
        $details | Should -Not -BeNullOrEmpty
        $details.Name | Should -Be $fullName
        $details.BodyType.gender | Should -Be $gender
        $details.BodyType.eye_color | Should -Be $eyeColour
        $details.HomeWorld.name | Should -Be $homeWorld
        $details.Films | Should -HaveCount $filmCount
    }
}
```

One advantage of using Pester tests with predefined TestCases[18] is you can inject properties into the test name. This gives us a more dynamic description rather than the static names provided in more simplistic tests. These are called templates[19].

3.4.7 Complex Test Output

Running the tests above generates the following output.

[18] https://pester.dev/docs/commands/It#-testcases
[19] https://pester.dev/docs/usage/data-driven-tests#using--templates

> The output is formatted slightly to fit in this book correctly.

Example 10: Output from the complex tests in example 9

```
Describing Search-SWFilm
  [+] Returns film with release date '1999-05-19' & director 'George Lucas'
      given title fragment 'Phantom' 1.14s (1.14s|2ms)
  [+] Returns film with release date '1980-05-17' & director 'Irvin Kershner'
      given title fragment 'Empire' 1.18s (1.18s|3ms)
  [+] Returns film with release date '1983-05-25' & director 'Richard Marquand'
      given title fragment 'Return' 1.16s (1.16s|5ms)

Describing Get-SWPerson
  [+] Returns person metadata for 'Darth Maul' with gender 'male',
      eye colour 'yellow' & film count of 1 3.96s (3.96s|5ms)
  [+] Returns person metadata for 'Luke Skywalker' with gender 'male',
      eye colour 'blue' & film count of 6 4.37s (4.37s|2ms)
  [+] Returns person metadata for 'Mon Mothma' with gender 'female',
      eye colour 'blue' & film count of 1 4.19s (4.19s|3ms)
Tests completed in 16.11s
Tests Passed: 6, Failed: 0, Skipped: 0 NotRun: 0
```

3.5 Conclusion

As seen from the code samples above, implementing the AAA approach using Pester is straightforward. Pester's use of commands with parameters that give English-like syntax goes a long way to make it easy to read and understand code, helping developers understand and write tests quickly.

3.6 Further Reading

- Extras for this chapter[20].
- Pester on GitHub[21].
- Pester documentation[22].
- AAA overview[23].
- PowerShell Gallery[24].

[20] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/tree/main/Edition-01/Starwars-Demo/
[21] https://github.com/pester/Pester
[22] https://pester.dev/docs/quick-start
[23] https://developers.mews.com/aaa-pattern-a-functional-approach/
[24] https://www.powershellgallery.com/

4. Mocking

Mocking is an approach to testing that involves the replacement of dependencies with simulated equivalents. Most useful at the unit testing stage, mocking ensures consistent behavior of functions or data that the code unit you're testing depends on.

4.1 Mocking and Mock Testing

There are several important advantages to mocking:

- **Performance**. Often, external APIs have a latency or take substantial processing time. This is also true for complex local functions that perform many operations. By simulating the behavior of these locally, tests will run much quicker and use fewer processor cycles. This saves both time and, in the case of third-party hosted test runners, money.
- **Consistency**. The reliability of dependencies outside of the code unit you're testing is assumed during unit tests. Using real dependencies can result in unexpected behavior and cause false test passes or failures. By simulating these, you only need to worry about the code in the test, since you know for sure what your mocked dependencies are doing.
- **Analysis**. Without modification, there may be no way to monitor when or how often a dependency is called. You can monitor mocked dependencies and create assertions to check that the code you're testing has called them.
- **Isolation**. A unit test should never have downstream effects on production environments. Mocking prevents this by eliminating real calls to external functions or APIs.

Mock testing is a complement to unit testing, focussing on behavior-based verification, as opposed to traditional unit testing, which is state-based.[1]

This difference is highlighted with some code that refreshes a local data store. Later, you will learn how to replace this with mocks.

Example 1: A function that updates a local data store from a remote one

```
function Update-DataStore {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory)][string]$Name,
        [Parameter(Mandatory)][string]$Source
    )
    process {
        # Get list of updates from API
        $RequestUri = '{0}/{1}' -f $Source.TrimEnd('/'), 'updates'
        $Updates = (Invoke-RestMethod -Uri $RequestUri).updates.date

        # Determine latest update and parse unix timestamp
```

[1] M. Fowler. (2007, Jan. 02). *Mocks Aren't Stubs*. martinFowler.com. [Online]. Available: https://martinfowler.com/articles/mocksArentStubs.html. [Accessed: Jun. 13, 2022].

```
        [int64]$Latest = 0
        $Updates.ForEach{ if ($_ -gt $Latest) { $Latest = $_ } }
        $UnixEpoch = [datetime]::new(1970, 1, 1, 0, 0, 0, 0, 1)
        $Update = $UnixEpoch.AddMilliseconds($Latest)

        # Get local data store
        $Store = Get-DataStore -Name $Name

        # Compare updates, and stop if local store is up-to-date
        if ($Update -le $Store.Update) { return }

        # Get data of latest update from API
        $RequestUri = '{0}/{1}/{2}' -f $Source.TrimEnd('/'), 'data', $Latest
        [psobject]$NewData = Invoke-RestMethod $RequestUri

        # Save new data to local store
        Set-DataStore -Name $Name -Data $NewData.data -Update $Update
    }
}
```

The real functionality of `Get-DataStore`, `Set-DataStore`, and the API doesn't matter for this example, but you can view a functional demonstration[2] in the Extras[3] repository for this book on GitHub.

Unit tests on this function should focus on the state of the data store after execution, verifying that the function updates the correct one based on the `-Name` parameter and that updates only occur if the local store is outdated. You can, of course, use mocking for these tests to eliminate API calls with `Invoke-RestMethod` and ensure that the test operates on a dummy data store.

Mock tests, on the other hand, should focus on the calls to `Get-DataStore`, `Set-DataStore`, and `Invoke-RestMethod` to verify that the function calls them the correct number of times and with the correct parameters.

4.1.1 Stubs, Fakes, and Mocks

There are many terms to describe the simulation of dependencies for the purposes of testing. How these terms define specific approaches and implementations of them also varies wildly. Three words that consistently appear in testing literature, however, are *stubs*, *fakes*, and *mocks*. This section briefly explains these and a few other terms, using the definitions given in *The Art of Unit Testing*:[4]

- **Fake**: Any object, function, or API that replaces a real dependency in tests. Under some definitions, a fake refers to a fully comprehensive simulation of a dependency, operationally compatible with the real behavior.
- **Stub**: A simple fake of a dependency that provides predetermined responses or values and can't fail a test. A validation stub might always return `$true`, for example.

[2] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/tree/main/Edition-01/Mocking/DataStoreDemo/

[3] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/tree/main/Edition-01/

[4] R. Osherove. (2013, Nov.). *The Art of Unit Testing*. 2nd ed. Germany: Manning. ISBN: 9781617290893.

- **Mock**: A more complex fake of a dependency that provides dynamic responses and can fail a test. A mocked database API might fail a test if it receives an invalid query, for example.
- **Seam**: A place in the code you're testing where a dependency or functionality is interchangeable. An example of this from Example 1 is the call to `Get-DataStore`. You could redirect this to a stub that always returns a valid store object regardless of the `-Name` parameter.

The Pester documentation uses the term *mock* universally,[5] so the rest of the chapter assumes that *mock* can refer to any kind of fake.

4.2 Mocking in Pester with `Mock`

The fundamental method for mocking dependencies in Pester is `Mock <Command> {...}`. This creates an alias in the current scope that bypasses the underlying command.[6] The standard syntax for mocking is:

```
Mock [-CommandName] <Command> [-MockWith] <Mock Script>
```

The command must exist in the scope of the current block where you're defining the mock. For example, the `Get-DataStore` function from Example 1 might look like this:

Example 2: A function that retrieves a local data store object

```
function Get-DataStore {
    [CmdletBinding()]
    param (
        [Parameter(
            Mandatory,
            ValueFromPipeline,
            ValueFromPipelineByPropertyName
        )]
        [string]$Name
    )
    process {
        # Determine valid store filename and check it exists
        $StoreFile = Get-DataStoreFile -Name $Name -MustExist

        # If it doesn't, stop
        if (-not $? -or $null -eq $StoreFile) { return }

        # Import CLIXML data from store
        $Data = Import-Clixml -LiteralPath $StoreFile.FullName

        # Return rich store object with name and file time
        [pscustomobject]@{
            Update = $StoreFile.LastWriteTimeUtc
            Data   = $Data
            Name   = $Name
        }
    }
}
```

[5] Pester Team. (2022, Jun. 25). *Mocking with Pester*. Pester Docs. [Online]. Available: https://pester.dev/docs/usage/mocking. [Accessed: Sep. 04, 2022].

[6] Pester Team. (2022, Mar. 06). *Pester - Mock.ps1*. L228-L261. Pester/Pester on GitHub. [Online]. Available: https://github.com/pester/Pester/blob/main/src/functions/Mock.ps1. [Accessed: Jun. 14, 2022].

To create a simple mock (stub) for this function, you can return a similar object without accessing any real data.

> All examples in this chapter are consistent with *Pester 5.3*.

Example 3: Mocking the Get-DataStore function

```
Describe 'Data store test' {

    BeforeAll {
        # Mock must be placed in a BeforeAll/BeforeEach block
        # unless it's directly inside an It block
        Mock Get-DataStore {
            [pscustomobject]@{
                Name   = $Name # Copy name from $Name parameter
                Update = (Get-Date).AddDays(-1) # Yesterday
                Data   = [pscustomobject]@{} # Empty store data
            }
        }
    }

}
```

From Pester 5.0 and upwards, you don't need to include a `param()` block in your mock script. Pester copies the parameters from the real function. This is also the case when you use the `-TestCases` parameter with `It`.[7]

If you work with earlier versions of Pester, you need to add `param ($Name)` to the top of the mock script.

The mock script returns the same kind of object as the real function, but doesn't check to see if a real store exists with the passed name, nor tries to access any real data. Instead, it returns the name passed with the `$Name` parameter as if it was a valid data store name, along with an empty `Data` property and `Update` value of exactly one day ago. Running a few tests with random names demonstrates that the mock is now receiving the function calls.

Example 4: Tests making use of the mock defined in Example 3

```
# Same Describe block as Example 3
It "Returns valid fake store object for '<RandomName>'" -TestCases @(
    @{ RandomName = '1234' }
    @{ RandomName = 'AnotherRandomName' }
) {
    $Store = Get-DataStore -Name $RandomName
    $Store.Name | Should -BeExactly $RandomName
    $Store.Update.DayOfYear | Should -Be ((Get-Date).DayOfYear - 1)
    $Store.Data | Should -BeOfType [pscustomobject]
    $Store.Data.PSObject.Properties | Should -BeNullOrEmpty
}
```

[7] Pester Team. (2021, May. 14). *Migrating from Pester v4 to v5*. Pester Docs. [Online]. Available: https://pester.dev/docs/migrations/v4-to-v5. [Accessed: Jun. 16, 2022].

```
Invoke-Pester .\Examples3and4.Tests.ps1 -Output Detailed
```

```
Pester v5.3.0

Starting discovery in 1 files.
Discovery found 2 tests in 24ms.
Running tests.

Running tests from 'Examples3and4.Tests.ps1'
Describing Data store test
  [+] Returns valid fake store object for '1234' 17ms (16ms|1ms)
  [+] Returns valid fake store object for 'AnotherRandomName' 3ms (3ms|0ms)
Tests completed in 106ms
Tests Passed: 2, Failed: 0, Skipped: 0 NotRun: 0
```

Find the Examples3and4.Tests.ps1[8] file used in this example on the Extras[9] repository on GitHub.

It's important that you define the mock in the same scope or a parent scope of the tests that use it. It must also be inside a `BeforeEach`, `BeforeAll`, or `It` block. The chapter covers mock scoping in more detail later.

4.3 Mock Testing and Verifiable Mocks

So how do you use Pester mocks for mock testing? The `Should` command has two assertions for this, and both rely on mocks.

4.3.1 Should -Invoke

The first, `Should -Invoke`, counts how many times a test called a mock and fails the test if the number doesn't match. Assume for the following examples that the mock shown in Example 3 is present in the current block.

[8] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/blob/main/Edition-01/Mocking/DataStoreDemo/Examples3and4.Tests.ps1

[9] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/tree/main/Edition-01/

Example 5: Using Should -Invoke to count mock calls

```
Describe 'Should -Invoke tests' {
    # Same mock as Example 3

    It 'Calls Get-DataStore five times' {
        for ($i = 0; $i -lt 5; $i++) {
            Get-DataStore -Name 'SomeName'
        }
        # At least once
        Should -Invoke 'Get-DataStore'

        # At least twice
        Should -Invoke 'Get-DataStore' -Times 2

        # Exactly 5 times
        Should -Invoke 'Get-DataStore' -Exactly 5

        # No more than 5
        Should -Not -Invoke 'Get-DataStore' -Times 6

        # Not exactly 4 times
        Should -Not -Invoke 'Get-DataStore' -Exactly 4
    }

}
```

The `-Times` and `-Exactly` parameters coupled with `-Not` provide full control over the number of times that your tests can call a mock. `Should -Not -Invoke ...` without `-Times` or `-Exactly` means the tests shouldn't call the mock at all in this scope. By default, Pester assumes you mean calls to the mock made **only** in the current scope. In Example 5, this means the `It` block where the `Should` assertions are.

To change the target scope of the assertion, use the `-Scope` parameter. This accepts either `It`, `Context`, `Describe`, or a positive integer.

- *It* means the current `It` block containing the assertion
- *Context* and *Describe* mean the innermost block of that type that contains the current `It` block
- A positive integer means the *nth* parent of the current `It` block

Since you can nest `Context` and `Describe` blocks in many ways, the *nth* parent mode is useful for selecting an exact parent scope of the current `It` block. If you have one `Context` block inside another, for example, you can't target the outer one with `-Scope Context`.

The next example creates some more `It` tests, but inside nested `Context` blocks within the `Describe` block.

Example 6: Changing the target scope of Should -Invoke

```
# Same Describe block as Example 5
Context 'Scope tests' {

    # Example 6a:
    It 'Calls Get-DataStore exactly once in the It block' {
        Get-DataStore -Name 'SomeOtherName'
        Should -Invoke 'Get-DataStore' -Exactly 1 -Scope It
        # Same as above since It is the 0th parent (this block)
        Should -Invoke 'Get-DataStore' -Exactly 1 -Scope 0
    }

    Context 'An inner scope' {

        # Example 6b:
        It 'Calls Get-DataStore once in the current Context block' {
            Get-DataStore -Name 'SomeOtherName'
            Should -Invoke 'Get-DataStore' -Exactly 1 -Scope Context
            # Same as above since Context is the 1st parent
            Should -Invoke 'Get-DataStore' -Exactly 1 -Scope 1
        }

        # Example 6c:
        It 'Fails to target outer Context block' {
            # THIS TEST FAILS since the innermost Context block is selected
            Should -Invoke 'Get-DataStore' -Exactly 2 -Scope Context
        }
        It 'Calls Get-DataStore twice in outer Context block' {
            # The 2nd parent of this It block is the outer Context block
            Should -Invoke 'Get-DataStore' -Exactly 2 -Scope 2
        }
        It 'Calls Get-DataStore seven times in outer Describe block' {
            Should -Invoke 'Get-DataStore' -Exactly 7 -Scope Describe
            # The 3rd parent of this It block is the Describe block
            Should -Invoke 'Get-DataStore' -Exactly 7 -Scope 3
        }

    }

}
```

The two assertions in the first `It` block (Example 6a) target only the scope of the `It` block itself. This is the same as the default behavior when `-Scope` isn't present. The assertions in the second `It` block (Example 6b) target the inner `Context` block, called '*An inner scope*'. This demonstrates that `-Scope Context` selects the **innermost** parent of that type, and the same is true with `Describe`.

Example 6c is where things get a little trickier. Since the `It` block is inside nested `Context` blocks, the first assertion fails because the target scope is the inner one. Instead of the expected two calls, it only sees one. To target the outer `Context` block, called '*Scope tests*', you must use a numerical scope. In the case of Example 6c, `-Scope 1` is the inner `Context` block, `-Scope 2` is the outer `Context` block, and `-Scope 3` is the `Describe` block (not shown).

If you pass a scope number of four, you've now reached the *root* scope of the test file, which includes other `Describe` and `Contexts` blocks that run before this one. Therefore, calls to a mock of the same name in those separate blocks will now count towards `Should -Invoke`, even if defined with a different mock inside a different `Before*` block. You can't go further than the root scope, so scope numbers larger than this still target the root scope.

4.3.2 Should -InvokeVerifiable

`Should -InvokeVerifiable` is much simpler and determines whether the tests have called all mocks marked as verifiable at least once. To mark a mock as verifiable, pass the `-Verifiable` switch parameter to the `Mock` statement.

Example 7: Marking a mock as verifiable

```
Mock Get-DataStore {...} -Verifiable
```

When Pester comes across the assertion, the test only passes if tests called all verifiable mocks **before** that point.

Example 8: Checking that the tests called all verifiable mocks

```
It 'Calls all verifiable mocks at least once' {
    Should -InvokeVerifiable
}
```

Since Pester evaluates the assertions at the time it comes across them, take care with your placement of `-Invoke` and `-InvokeVerifiable`. Place them **after** any mock calls that you want Pester to count.

4.3.3 Running the Mock Assertion Tests

The code from Examples 5 to 8 contains seven tests, of which six should pass and one should fail.

Example 9: Running the mock assertion tests

```
Invoke-Pester .\Examples5to8.Tests.ps1 -Output Detailed
```

```
Pester v5.3.0

Starting discovery in 1 files.
Discovery found 7 tests in 10ms.
Running tests.

Running tests from 'Examples5to8.Tests.ps1'
Describing Should -Invoke tests
  [+] Calls Get-DataStore five times 14ms (13ms|1ms)
 Context Scope tests
   [+] Calls Get-DataStore exactly once in the It block 4ms (3ms|1ms)
  Context An inner scope
    [+] Calls Get-DataStore once in the current Context block 5ms (3ms|1ms)
    [-] Fails to target outer Context block 2ms (2ms|0ms)
     Expected Get-DataStore to be called 2 times exactly but was called 1 times
     at Should -Invoke 'Get-DataStore' -Exactly 2 -Scope Context,
     Examples5to8.Tests.ps1:63 at <ScriptBlock>, Examples5to8.Tests.ps1:63
    [+] Calls Get-DataStore twice in outer Context block 2ms (2ms|0ms)
    [+] Calls Get-DataStore seven times in outer Describe block 5ms (5ms|1ms)
    [+] Calls all verifiable mocks at least once 1ms (1ms|0ms)
Tests completed in 119ms
Tests Passed: 6, Failed: 1, Skipped: 0 NotRun: 0
```

Find the Examples5to8.Tests.ps1[10] file used in this example on the Extras[11] repository on GitHub.

The test results confirm the assertions are scoped correctly. Notice also that the indentation of the test result output varies based on the nesting of the blocks in the test file. The final (verifiable mock assertion) test is inside the '*Scope tests*' block, but not inside the '*An inner scope*' block, for example. This output can prove useful when diagnosing scoping errors for mock-related assertions.

Assert-MockCalled and Assert-VerifiableMock

Pester versions before 5.0 tested mock calls with `Assert-MockCalled` and `Assert-VerifiableMock`. While these are still available in Pester 5.0 and later, they're now deprecated and you should use the assertions `Should -Invoke` and `Should -InvokeVerifiable` instead.

4.4 Mock Scoping

The chapter has so far described checking for mock calls in various scopes, but you can also apply your mocks to specific scopes. By default, Pester scopes your mock based on the block you place it in.[12]

- If you place the `Mock...` declaration in an `It` or `BeforeEach` block, it applies only to `It` blocks.
  - Placement in an `It` block applies the mock only within that `It`; others in the same `Context` or `Describe` aren't affected.
  - Placement in a `BeforeEach` block applies the mock to every `It` block in the current `Context` or `Describe`, as if the declaration was made inside each.
- If you place the `Mock...` declaration in a `BeforeAll` block, it applies to the whole of the current `Context` or `Describe`.
- You shouldn't place a `Mock...` declaration directly inside a `Context` or `Describe` block, in line with Pester 5.0's new discovery and run best practices.[13]
- You shouldn't place a `Mock...` declaration inside `BeforeDiscovery` blocks—mocks don't function in the *discovery* phase since it's only for inspecting the test file structure and generating tests.
- Placing a `Mock...` declaration in `AfterEach` or `AfterAll` blocks is ineffective since Pester resets the scope between tests. The section discusses this later.

You can see this in action with a few simple tests.

[10] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/blob/main/Edition-01/Mocking/DataStoreDemo/Examples5to8.Tests.ps1

[11] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/tree/main/Edition-01/

[12] Pester Team. (2022, Jun. 25). *Mocking with Pester.* Pester Docs. [Online]. Available: https://pester.dev/docs/usage/mocking. [Accessed: Sep. 04, 2022].

[13] Pester Team. (2022, Jun. 19). *Discovery and Run.* Pester Docs. [Online]. Available: https://pester.dev/docs/usage/discovery-and-run. [Accessed: Jun. 21, 2022].

Example 10: Mock scoping is based on placement

```
Describe 'Mock scoping tests' {

    # Example 10a:
    It 'BeforeAll sets up mocks before all tests' {
        Get-DataStore -Name RandomName | Should -Be 'Mocked'
    }

    BeforeAll {
        # Applies to whole Describe
        Mock Get-DataStore { 'Mocked' }
    }

    # Example 10b:
    It 'BeforeAll applies mock to entire parent block' {
        Get-DataStore -Name RandomName | Should -Be 'Mocked'
    }

    # Example 10c:
    It 'Mocks applied inside It take precedence' {
        # Override with local mock
        Mock Get-DataStore { 'Re-mocked' }
        $Result = Get-DataStore -Name RandomName
        $Result | Should -Not -Be 'Mocked'
        $Result | Should -Be 'Re-mocked'
    }

}
```

Example 10a places an `It` block before the `BeforeAll`, but the mock is still available as `BeforeAll` runs before any tests. Example 10b returns the same result and confirms that mocks placed in `BeforeAll` apply to every `It` in the current `Context` or `Describe`. Example 10c demonstrates that more locally defined mocks override existing ones. This means any mock of the same dependency defined in an `It` block replaces the one that applies to the `Context` or `Describe`. Likewise, any mocks defined in the `BeforeAll` or `BeforeEach` block of a nested (child) block replace those from the parent block.

If not overridden, mocks are inherited by child blocks:

Example 11: Mocks are inherited in child scopes unless overridden

```
# Same Describe block as Example 10

# Example 11a:
Context 'Inner context' {

    It 'Mocks apply to child scopes' {
        # Inherits the mock from the Describe block
        Get-DataStore -Name RandomName | Should -Be 'Mocked'
    }

}

# Example 11b:
Context 'Another inner context' {

    BeforeAll {
        Mock Get-DataStore { 'Re-mocked again' }
    }

    It 'Unless a more local mock takes precedence' {
        # Inherits the overridden mock from the Context block
        $Result = Get-DataStore -Name RandomName
        $Result | Should -Not -Be 'Mocked'
        $Result | Should -Not -Be 'Re-mocked'
        $Result | Should -Be 'Re-mocked again'
    }

}
```

Overriding a mock does so at the current scope and in all child scopes. The `Context` block in Example 11a inherits the mock from the parent `Describe` and, in turn, the `It` block inherits it from the `Context`. In Example 11b, a new mock replaces the inherited one, so the `It` block inherits this replacement from the `Context`.

It's important to remember that the `BeforeAll` block runs once before all the tests in the current scope, whereas `BeforeEach` runs before **every** test in the same scope. Because of this, a mock defined in a `BeforeEach` overrides one defined in a `BeforeAll` of the same scope.

Example 12: The execution order of BeforeAll and BeforeEach has consequences for mocks

```
# Same Describe block as Example 10
Context 'BeforeEach and BeforeAll execution order' {

    BeforeEach {
        if ($Alt) {
            Mock Get-DataStore { 'BeforeEach Mock' }
        }
    }
    BeforeAll {
        Mock Get-DataStore { 'BeforeAll Mock' }
    }

    It 'Test <Test> should use the Before<Mock> Mock' -TestCases @(
        @{ Test = 1; Alt = $false; Mock = 'All' }
        @{ Test = 2; Alt = $true; Mock = 'Each' }
        @{ Test = 3; Alt = $false; Mock = 'All' }
    ) {
        Get-DataStore -Name RandomName | Should -Be "Before$Mock Mock"
    }

}
```

Example 12 runs three tests, and the mock they each use changes between them. Why is this? Looking at the execution order, it becomes clear how and when the original mock gets overridden.

1. The `BeforeAll` of the `Context` block runs and overrides the original mock from the `Describe` block
2. The `BeforeEach` of the `Context` block runs before *Test 1*, but the `$Alt` variable is `False` so nothing happens
3. *Test 1* runs and inherits the mock from the `BeforeAll`.
4. The `BeforeEach` runs before *Test 2*, and the `$Alt` variable is `True` this time, so it defines a new mock

5. *Test 2* runs and inherits the replaced mock from the `BeforeEach` that just ran
6. The `BeforeEach` runs before *Test 3*, and the `$Alt` variable is again `False`, so nothing happens
7. *Test 3* runs and inherits the mock from the `BeforeAll`.

So, why does *Test 3* not inherit the new mock, since the `BeforeEach` defines it earlier, before *Test 2*? The answer is that Pester 5.0 runs each test in a new scope inherited directly from the parent scope.[14] Therefore, each generated test can't communicate with others, and each run of `BeforeEach` is specific to that test. The effects of the `BeforeEach` that runs for *Test 2* end when the test does, and the scope 'resets' to the parent `Context` one. When *Test 3* runs, the environment is exactly as it was before *Test 2* and *Test 1*, so when `BeforeEach` does nothing, the mock created in the `BeforeAll` is what gets inherited.

Considering the isolated nature of individual test scopes, it becomes apparent that defining mocks in `AfterAll` or `AfterEach` blocks is useless for their intended purpose. Pester would create the mock *after* all or each of the tests and immediately remove it as the test or block scope was torn down.

Running the tests from Examples 10 to 12:

Example 13: Running the mock scoping tests

```
Invoke-Pester .\Examples10to12.Tests.ps1 -Output Detailed
```

```
Pester v5.3.0

Starting discovery in 1 files.
Discovery found 8 tests in 54ms.
Running tests.

Running tests from 'Examples10to12.Tests.ps1'
Describing Mock scoping tests
  [+] BeforeAll sets up mocks before all tests 14ms (12ms|1ms)
  [+] BeforeAll applies mock to entire parent block 8ms (7ms|0ms)
  [+] Mocks applied inside It take precedence 16ms (15ms|0ms)
 Context Inner context
   [+] Mocks apply to child scopes 9ms (8ms|1ms)
 Context Another inner context
   [+] Unless a more local mock takes precedence 16ms (14ms|1ms)
 Context BeforeEach and BeforeAll execution order
   [+] Test 1 should use the BeforeAll Mock 33ms (31ms|1ms)
   [+] Test 2 should use the BeforeEach Mock 8ms (8ms|0ms)
   [+] Test 3 should use the BeforeAll Mock 2ms (2ms|0ms)
Tests completed in 270ms
Tests Passed: 8, Failed: 0, Skipped: 0 NotRun: 0
```

Find the Examples10to12.Tests.ps1[15] file used in this example on the Extras[16] repository on GitHub.

[14] Pester Team. (2022, Jun. 19). *Discovery and Run.* Pester Docs. [Online]. Available: https://pester.dev/docs/usage/discovery-and-run. [Accessed: Jun. 21, 2022].

[15] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/blob/main/Edition-01/Mocking/DataStoreDemo/Examples10to12.Tests.ps1

[16] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/tree/main/Edition-01/

PesterBoundParameters

When inside a mock script, the `$PSBoundParameters` variable doesn't work because of how Pester uses proxy functions for mocking.[17] Pester version 5.2 introduces a functionally equivalent stand-in for this, `$PesterBoundParameters`. Use this variable in the same way you would `$PSBoundParameters`.

4.5 Mocking in the Module Scope with -ModuleName

It's often necessary to modify the behavior of dependencies, but what if the dependency is a private function required by other functions inside a module? Mocking the dependency in the test scope won't help, since the functions in the module have their own scope. The answer is to create the mock in the module's scope.

There are two approaches to this:

- Use the `-ModuleName` parameter with `Mock`.
- Use `InModuleScope { ... }` to run the mock code or the entire test in the module scope.[18]

Consider the data store example once again. Imagine that the real `Get-DataStore` function relies on an internal function, `Get-DataStoreFile`, to get valid store files from the disk. By mocking the internal function in the module's scope, you've changed the behavior of the module without rewriting it.

The following snippet creates a module on-the-fly from the DataStoreFunctions.ps1[19] file, which the tests can interact with.

Example 14: Creating a module from a script file for use in tests

```
BeforeDiscovery {
    $GetModuleParams = @{
        Name        = 'DataStoreFunctions'
        ErrorAction = 'SilentlyContinue'
    }
    Get-Module @GetModuleParams | Remove-Module
    New-Module -Name DataStoreFunctions -ScriptBlock {
        # Load functions
        . (Join-Path (Split-Path $PSCommandPath) 'DataStoreFunctions.ps1')
        # Export public functions
        $Exports = @{
            Function = @(
                'New-DataStore',
                'Remove-DataStore',
                'Get-DataStore',
                'Get-DataStoreDate',
                'Set-DataStore',
                'Set-DataStoreDate',
```

[17] Pester Team. (2022, Jun. 25). *Mocking with Pester.* Pester Docs. [Online]. Available: https://pester.dev/docs/usage/mocking. [Accessed: Sep. 04, 2022].

[18] Pester Team. (2021, May. 13). *InModuleScope.* Pester Docs. [Online]. Available: https://pester.dev/docs/commands/InModuleScope. [Accessed: Jun. 22, 2022].

[19] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/blob/main/Edition-01/Mocking/DataStoreDemo/DataStoreFunctions.ps1

```
                'Update-DataStore'
            )
        }
        Export-ModuleMember @Exports
    } | Import-Module -Force
}
```

If you try to use `Get-DataStore` with an invalid name, it throws an error because it's currently using the real `Get-DataStoreFile` internal function.

Example 15: The module uses its real internal functions without mocking

```
Describe 'Mocking in modules' {

    It 'Throws an error because real private function called' {
        { Get-DataStore -Name SomeName } | Should -Throw
    }

}
```

You can't use `Mock` for `Get-DataStoreFile` because it's not exported as a public module member.

Example 16: Attempting to mock a private function in the test scope

```
# Same Describe block as Example 15
It "Can't mock a private function in the test scope" {
    Mock Get-DataStoreFile {}
}
```

```
[-] Can't mock a private function in the test scope 59ms (58ms|0ms)
 CommandNotFoundException: Could not find Command Get-DataStoreFile
```

Using the `-ModuleName` parameter of `Mock`, you can create a mock of the private function, and return a valid dummy store file. To make this work, you need to create the dummy data store in the `BeforeAll` section. You can use Pester's test drive for this, which is a temporary `PSDrive` specifically for tests.[20]

[20] Pester Team. (2022, Jun. 19). *Isolating File Operations using the TestDrive*. Pester Docs. [Online]. Available: https://pester.dev/docs/usage/testdrive. [Accessed: Jun. 21, 2022].

Example 17: Creating the dummy data store using Pester's test drive

```
BeforeAll {
    $DummyXMLParams = @{
        Path        = 'TestDrive:\DummyDataStore.xml'
        InputObject = [pscustomobject]@{}
    }
    Export-Clixml @DummyXMLParams
}
```

The module-scoped mock can now retrieve your dummy file:

Example 18: Mocking an internal function using a dummy file and Mock -ModuleName

```
# Same Describe block as Example 15
It 'Returns the dummy data when private function mocked' {
    Mock Get-DataStoreFile {
        Get-Item 'TestDrive:\DummyDataStore.xml'
    } -ModuleName DataStoreFunctions
    $Store = Get-DataStore -Name SomeName
    $Store.Name | Should -Be 'SomeName'
    $Store.Data | Should -BeNullOrEmpty
}
```

```
[+] Returns the dummy data when private function mocked 15ms (15ms|0ms)
```

You can achieve the same result using `InModuleScope`, either just for the creation of the mock or by running the entire test in the module scope.

Example 19: Mocking an internal function with InModuleScope

```
# Same Describe block as Example 15
It 'Alternative method for mocking module functions' {
    InModuleScope DataStoreFunctions {
        Mock Get-DataStoreFile {
            Get-Item 'TestDrive:\DummyDataStore.xml'
        }
    }
    $Store = Get-DataStore -Name SomeName
    $Store.Name | Should -Be 'SomeName'
    $Store.Data | Should -BeNullOrEmpty
}
```

You can use `InModuleScope` anywhere that you'd usually place code in a Pester test file. You can also use it to wrap entire `Describe` or `Context` blocks, in order to run entire test programs in the scope of the module:

Example 20: Running entire tests in the module scope

```
InModuleScope DataStoreFunctions {

    Describe 'Running tests directly in the module scope' {

        It 'Running the entire test in the module scope' {
            Mock Get-DataStoreFile {
                Get-Item 'TestDrive:\DummyDataStore.xml'
            }
            $Store = Get-DataStore -Name SomeName
            $Store.Name | Should -Be 'SomeName'
            $Store.Data | Should -BeNullOrEmpty
        }

    }

}
```

4.5.1 Mock Testing in the Module Scope

If you've created mocks in a module scope, you can still count calls to these with the -ModuleName parameter of `Should -Invoke`. With this parameter, you can use `Should` in the same way you would for globally scoped dependencies. Like with creating mocks, you can use mock assertions inside an `InModuleScope` block as if the tests were running inside the module.

Pester counts verifiable mocks regardless of scope, making the assertions much simpler. `Should -InvokeVerifiable` will succeed if the tests call all verifiable mocks, regardless of whether they're in a module scope or the global one.

Example 21: Mock assertions in the module scope

```
# Same Describe block as Example 15
It 'Mock assertions should work in the correct scope' {
    Mock Get-DataStoreFile {} -ModuleName DataStoreFunctions -Verifiable
    Get-DataStore -Name SomeName

    Should -Invoke Get-DataStoreFile -ModuleName DataStoreFunctions -Exactly 1

    InModuleScope DataStoreFunctions {
        Should -Invoke Get-DataStoreFile -Exactly 1
    }

    Should -InvokeVerifiable
}
```

4.5.2 Running the Module Scope Tests

Running the tests from Examples 14 to 21 should yield six tests, one of which fails with an error.

Example 22: Running the module scope tests

```
Invoke-Pester .\Examples14to21.Tests.ps1 -Output Detailed
```

```
Pester v5.3.0

Starting discovery in 1 files.
Discovery found 6 tests in 15ms.
Running tests.

Running tests from 'Examples14to21.Tests.ps1'
Describing Mocking in modules
  [+] Throws an error because real private function called 4ms (3ms|2ms)
  [-] Can't mock a private function in the test scope 50ms (49ms|1ms)

   CommandNotFoundException: Could not find Command Get-DataStoreFile
  [+] Returns the dummy data when private function mocked 7ms (7ms|0ms)
  [+] Alternative method for mocking module functions 5ms (4ms|0ms)
  [+] Mock assertions should work in the correct scope 5ms (5ms|0ms)

Describing Running tests directly in the module scope
  [+] Running the entire test in the module scope 5ms (4ms|2ms)
Tests completed in 163ms
Tests Passed: 5, Failed: 1, Skipped: 0 NotRun: 0
```

Find the Examples14to21.Tests.ps1[21] file used in this example on the Extras[22] repository on GitHub.

4.6 Dynamic Mock Behavior with -ParameterFilter

So far, the chapter has only covered simple mock behavior. Creating dynamic responses to input data or conditions is a little more complex, but Pester includes a feature that helps to ease this.

The `-ParameterFilter` parameter of `Mock` restricts the conditions under which Pester will redirect a dependency call to the mock. You can use this to create multiple mocks that respond to different sets of inputs. If no parameter filter matches the parameters, Pester passes the call to the underlying (real) dependency. Alternatively, you can create an additional mock with no parameter filters. This will receive all calls whose parameters don't match the filters of the other mocks.

The standard syntax of `Mock` with a parameter filter is:

```
Mock <Command Name> <Mock Script> -ParameterFilter <Filter Script>
```

[21] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/blob/main/Edition-01/Mocking/DataStoreDemo/Examples14to21.Tests.ps1

[22] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/tree/main/Edition-01/

The *filter script* is a script block that must return `True` in order for the mock to accept and handle the call. Think of this as similar to the script block you would pass to `Where-Object`.

Consider the `Update-DataStore` function from Example 1. It makes two calls to `Invoke-RestMethod`. One of these gets the list of updates to a remote data store, and the other gets the contents of a single update. Any unit tests of the `Update-DataStore` function need the dependency to behave consistently across tests, but differently according to the *URI* parameter.

Example 23: Mocking Invoke-RestMethod specifically for update checks

```
Describe 'Update-DataStore Tests' {

    BeforeAll {

        [int64]$LaterUnix = Get-Random -Minimum 16.1e+11 -Maximum 16.2e+11
        [int64]$EarlierUnix = Get-Random -Minimum 16e+11 -Maximum 16.1e+11

        Mock Invoke-RestMethod {
            @{ class = 'updates'; updates = @(
                    @{ guid = [guid]::Empty; date = $EarlierUnix }
                    @{ guid = [guid]::Empty; date = $LaterUnix }
                )
            }
        } -ParameterFilter { $Uri -like '*/updates' } -Verifiable

    }

}
```

The real API returns an object containing an array of update objects, each made of a GUID and a Unix time stamp in milliseconds. The mock in the example replicates this with two updates containing empty (zero) GUIDs and random time stamps in a predefined range. Returning two fake updates with differing time stamps means that tests can ensure `Update-DataStore` is selecting the latest updates correctly.

The new feature of this example is, of course, the `-ParameterFilter` parameter. Recall that Pester copies the parameters of the real dependency in mocks, and the same is true for filter scripts. The filter script here checks whether the `-Uri` parameter ends with '*/updates*'. Take a look back at Example 1 and you'll be able to see how this will 'catch' the right calls.

To mock the calls to `Invoke-RestMethod` that receive the new data store contents, a second mock is necessary.

Example 24: Mocking Invoke-RestMethod for data store content requests

```
# Same Describe/BeforeAll block as Example 22
Mock Invoke-RestMethod {
    $null = $Uri -match '/data/(\d+)$'
    [int64]$Date = $Matches[1]

    @{
        class = 'data'
        date  = $Date
        name  = "Update $Date"
        data  = @{ Property1 = 'Value 1'; Property2 = 'Value 2' }
    }
} -ParameterFilter { $Uri -match '/data/\d+$' } -Verifiable
```

The filter script in this example matches any calls where the URI ends with '*/data/*' and only numbers. Since the real API supplies data stores based on the Unix time stamps, this mock will intercept any attempts to retrieve new data store contents. The mock script itself simply provides a fake data store object with fixed data, and a date that's taken from the `-Uri` parameter passed to it.

This just leaves the behavior when the URI doesn't resemble an update or data store request at all. With no additions to the test script, Pester passes these calls on to the real dependency. This is undesirable given the purpose of unit tests, so you can use a third mock without a parameter filter to catch any other calls.

Example 25: Catching unmatched mock calls with a filterless mock

```
# Same Describe/BeforeAll block as Example 22
Mock Invoke-RestMethod {
    throw (
        'Invoke-RestMethod called with unexpected parameters: {0}' -f (
            $PesterBoundParameters.Keys.ForEach{
                "$_ = $($PesterBoundParameters.$_)"
            } -join ', '
        )
    )
}
```

This mock immediately throws an error in order to fail the test, since any URI passed that doesn't match the first two mocks is invalid for the real API. The extra code uses `$PesterBoundParameters` to list any parameter keys and values passed in the invalid scenario, which helps to diagnose and correct a test failure.

You're almost ready to use these mocks in unit tests. An important thing to remember is full dependency coverage. There aren't any mocks for `Get-DataStore` or `Set-DataStore` in this file, currently, and `Update-DataStore` calls both. Focusing on fully covering one dependency and forgetting to account for others is a common unit testing mistake.

The next example creates a dummy data store that the other mocks access.

Example 26: Additional mocks for testing Update-DataStore

```
# Same Describe/BeforeAll block as Example 22
$DummyName = 'RandomName', (Get-Random) -join '_'
$DummyStore = [pscustomobject]@{
    Name   = 'Not Set'
    Update = [datetime]::MinValue
    Data   = [pscustomobject]@{}
}

Mock Get-DataStore { $DummyStore } -Verifiable

Mock Set-DataStore {
    $DummyStore.Name = $Name
    $DummyStore.Update = $Update
    $DummyStore.Data = [pscustomobject]$Data
} -Verifiable
```

Notice how the values of the dummy store at definition differ from the fake values that the `Invoke-RestMethod` mocks return. The following tests use these to verify that `Update-DataStore` writes the correct values to the dummy store.

Example 27: Unit tests for Update-DataStore using the filtered mocks

```
# Same Describe block as Example 22
Context 'Unit tests' {

    BeforeEach {
        Update-DataStore -Name $DummyName -Source 'https://example.com'
    }

    It 'Accesses the right data store' {
        $DummyStore.Name | Should -BeExactly $DummyName
    }
    It 'Chooses the latest update' {
        $UnixEpoch = [datetime]::new(1970, 1, 1, 0, 0, 0, 0, 1)
        $Later = $UnixEpoch.AddMilliseconds($LaterUnix)
        $DummyStore.Update | Should -BeExactly $Later
    }
    It 'Adds new values to the data store' {
        $DummyStore.Data.Property1 | Should -Be 'Value 1'
        $DummyStore.Data.Property2 | Should -Be 'Value 2'
    }

}
```

The tests in this example run `Update-DataStore` three times, once for each `It` block. The first call should update the dummy data store with the values from the second `Invoke-RestMethod` mock. The name stored in `$DummyStore` therefore changes from '*Not Set*' to the randomly generated name from Example 26. The two subsequent tests shouldn't attempt to update the dummy store again, since the date it contains now matches that from the mock. Later tests can confirm this with mock assertions.

The second test causes `Update-DataStore` to run again, but, as discussed in the previous paragraph, no values should change. This test checks that the dummy store's date has changed from the earliest that `DateTime` can represent, to the date represented by `$LaterUnix`. The test uses the same method to convert the Unix time stamp from `$LaterUnix` to a `DateTime`, so these dates should match if `Update-DataStore` is accessing and converting this value correctly. The third test causes a third run, and this time checks that `Update-DataStore` has added the two properties and values from the mock to the dummy data store.

Note that the three assertions in the example will work just as well if placed in the first `It` block. All the data these tests look for should have been set in the first call to `Update-DataStore`. Using three `It` blocks provides more granular results in the event of a single assertion failing and has the added benefit of calling `Update-DataStore` multiple times, since the call is in a `BeforeEach` block. The multiple calls support the following mock tests, which check that the second two calls to `Update-DataStore` result in no further changes to the dummy store, since it's *up-to-date* after the first one.

4.6.1 Filtered Mock Assertions

The next set of tests needs to verify that `Update-DataStore` only wrote to the dummy store when it was out of date. Since there are multiple mocks for `Invoke-RestMethod`, you need to filter these assertions, too. You can use the same parameter name, `-ParameterFilter`, with `Should -Invoke` to restrict what Pester counts.

Example 28: Mock tests that check how Update-DataStore responds to an updated dummy store

```
# Same Describe block as Example 22
Context 'Mock tests' {

    It 'Accesses data stores and API at least once' {
        Should -InvokeVerifiable
    }
    It 'Calls Invoke-RestMethod 4 times overall' {
        Should -Invoke Invoke-RestMethod -Exactly 4 -Scope Describe
    }
    It 'Calls Invoke-RestMethod 3 times for update URL' {
        Should -Invoke Invoke-RestMethod -ParameterFilter {
            $Uri -match '.+/updates$'
        } -Exactly 3 -Scope Describe
    }
    It 'Calls Invoke-RestMethod once for data URL' {
        Should -Invoke Invoke-RestMethod -ParameterFilter {
            $Uri -match '/data/\d+$'
        } -Exactly 1 -Scope Describe
    }

}
```

Note first that `-InvokeVerifiable` isn't affected at all by parameter filters, only by the `-Verifiable` switch of `Mock`. If a test calls all mocks marked as verifiable, the assertion succeeds. You could mark only some of your filtered mocks as verifiable, in which case those mocks **must** be called for the assertion to succeed. Additional mocks with the same name, but without the `-Verifiable` switch, don't count.

The second test checks `Invoke-Restmethod` with no filtering, so the assertion counts calls to **all** mocks with that name. There should be four calls—three with the update URI and one with the data URI.

The third and fourth tests check this explicitly with the `-ParameterFilter` parameter of `Should -Invoke`. Note how the third test uses a different parameter filter script than the one for the mock itself. The filter script for the assertion uses a regex match, whereas the one for the mock definition uses a wildcard match. This demonstrates that the filters you use for mock *assertions* don't have to be identical to the ones you use for mock *definitions*. Of course, if no mock exists that would handle the parameters defined by your assertion filter, the assertion will always fail. This freedom means you can cover overlapping parameter scenarios for multiple filtered mocks (of the same dependency) in your assertions. Or, you can test more granular scenarios for the same mock with individual assertions that have stricter parameter filters.

Finally, note that the `Should -Invoke` assertions all specify `-Scope Describe`. Since the calls to `Update-DataStore` came from a different `Context` block (‘*Unit tests*’), they won't count in this one (‘*Mock tests*’) due to mock scoping. By specifying the parent `Describe` of both `Context` blocks, the assertions here will count them.

This just leaves a couple of mock tests for `Get-DataStore` and `Set-DataStore`. As with the previous example, there should be three calls to `Get-DataStore` since three update checks should occur. Only one call to `Set-DataStore` confirms that repeated calls to `Update-DataStore` resulted in no action on the dummy store.

Example 29: Mock tests for the calls to Get-DataStore and Set-DataStore

```
# Same Describe/Context block as Example 28
It 'Calls Get-DataStore three times' {
    Should -Invoke Get-DataStore -Exactly 3 -Scope Describe
}
It 'Calls Set-DataStore once' {
    Should -Invoke Set-DataStore -Exactly 1 -Scope Describe
}
```

4.6.2 Running the Filtered Mock Tests

The code from Examples 23 to 29 yields nine tests, all of which should pass.

Example 30: Running the parameter filter tests

```
Invoke-Pester .\Examples23to29.Tests.ps1 -Output Detailed
```

```
Pester v5.3.0

Starting discovery in 1 files.
Discovery found 9 tests in 15ms.
Running tests.

Running tests from 'Examples23to29.Tests.ps1'
Describing Update-DataStore Tests
 Context Unit tests
   [+] Accesses the right data store 12ms (8ms|3ms)
   [+] Chooses the latest update 9ms (8ms|1ms)
   [+] Adds new values to the data store 7ms (6ms|1ms)
 Context Mock tests
   [+] Accesses data stores and API at least once 2ms (1ms|1ms)
   [+] Calls Invoke-RestMethod 4 times overall 3ms (3ms|1ms)
   [+] Calls Invoke-RestMethod 3 times for update URL 11ms (10ms|1ms)
   [+] Calls Invoke-RestMethod once for data URL 4ms (4ms|1ms)
   [+] Calls Get-DataStore three times 3ms (2ms|0ms)
   [+] Calls Set-DataStore once 3ms (3ms|1ms)
Tests completed in 185ms
Tests Passed: 9, Failed: 0, Skipped: 0 NotRun: 0
```

Find the Examples23to29.Tests.ps1[23] file used in this example on the Extras[24] repository on GitHub.

[23] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/blob/main/Edition-01/Mocking/DataStoreDemo/Examples23to29.Tests.ps1

[24] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/tree/main/Edition-01/

4.6.3 Restricting Mock Calls Further with -ExclusiveFilter

The `Should -Invoke` assertion has a second filter parameter, `-ExclusiveFilter`. This works the same as assertions that use `-ParameterFilter`, but only succeeds if:

- The tests call the mock the number of times specified with `-Times` or `-Exactly`
- No other calls to the mock that **don't** match the filter occur

This means that you can only use `-ExclusiveFilter` once for each mock in the current scope. Any other assertions for the same mock and scope that use `-ParameterFilter` become useless. Any mock calls that would cause additional assertions to pass, will cause the -ExclusiveFilter pass to fail. This makes the filter useful when it's imperative that a dependency call occurs only under a single set of conditions. Examples include where the function under test transmits sensitive data or makes important changes.

Example 31: Using -ExclusiveFilter to assert mock calls strictly

```
Describe 'Set-DataStore tests' {

    Context 'Correct file' {

        BeforeAll {
            Mock Export-Clixml {}
            Mock Set-ItemProperty {}
            Mock Get-DataStoreFile { Join-Path 'TestDrive:' "$Name.xml" }

            $Rand = 'Random', (Get-Random) -join ''
            $ExpectPath = Join-Path 'TestDrive:' "$Rand.xml"
            $ExpectDate = Get-Date -AsUTC
            Set-DataStore -Name $Rand -Data @{} -Update $ExpectDate
        }

        It 'Calls Export-Clixml with correct path once' {
            Should -Invoke Export-Clixml -Exactly 1 -ExclusiveFilter {
                $LiteralPath -eq $ExpectPath
            } -Scope Context
        }

    }

}
```

The exclusive filter ensures that no other calls to `Export-Clixml` occur, and also that the matching call uses the correct file path. Note once again that the assertion specifies an explicit scope (*Context*). The call to `Set-DataStore` happens in the `BeforeAll` block of `Context`, so the mock call also happens in the `Context` scope, not the `It` scope.

You can use additional exclusive filters in assertions for other mocks.

Example 32: ExclusiveFilter excludes nonmatching calls to the same mock and in the same scope

```
# Same Context block as Example 31
It 'Calls Set-ItemProperty with correct path, name, and value once' {
    Should -Invoke Set-ItemProperty -Exactly 1 -ExclusiveFilter {
        $LiteralPath -eq $ExpectPath -and
        $Name -eq 'LastWriteTimeUtc' -and
        $Value -eq $ExpectDate
    } -Scope Context
}
```

The test script declares the `$ExpectDate` and `$ExpectPath` variables in the `BeforeAll` block, so these are available to the entire `Context` block and all `It` blocks it contains. Run the tests to confirm this:

Example 33: Running the ExclusiveFilter tests

```
Invoke-Pester .\Examples31and32.Tests.ps1 -Output Detailed
```

```
Pester v5.3.0

Starting discovery in 1 files.
Discovery found 2 tests in 31ms.
Running tests.

Running tests from 'Examples31and32.Tests.ps1'
Describing Set-DataStore mock tests
 Context Correct file
   [+] Calls Export-Clixml with correct path once 12ms (11ms|2ms)
   [+] Calls Set-ItemProperty with correct path, name,
       and value once 12ms (11ms|1ms)
Tests completed in 147ms
Tests Passed: 2, Failed: 0, Skipped: 0 NotRun: 0
```

Find the Examples31and32.Tests.ps1[25] file used in this example on the Extras[26] repository on GitHub.

4.7 Calling Real Dependencies While They're Mocked

Since Pester 4.0, mocks create an alias that redirects to your mock script.[27] [28] This means you can use `Get-Command` to retrieve the real dependency and call it, both in your tests and from within your mock scripts.

[25] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/blob/main/Edition-01/Mocking/DataStoreDemo/Examples31and32.Tests.ps1

[26] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/tree/main/Edition-01/

[27] Pester Team. (2022, Mar. 06). *Pester - Mock.ps1*. L228-L261. Pester/Pester on GitHub. [Online]. Available: https://github.com/pester/Pester/blob/main/src/functions/Mock.ps1. [Accessed: Jun. 14, 2022].

[28] Pester Team. (2022, Jun. 22). *Migrating from Pester v3 to v4*. Pester Docs. [Online]. Available: https://pester.dev/docs/migrations/v3-to-v4. [Accessed: Jul. 02, 2022].

Example 34: Accessing real dependencies when they're mocked

```
Describe 'Set-DataStore unit tests' {

    Context 'Correct data' {

        BeforeAll {

            Mock Get-DataStoreFile {
                return (Join-Path 'TestDrive:' "$Name.xml")
            }

            Mock Export-Clixml {
                $RealCmd = Get-Command Export-Clixml -CommandType Cmdlet
                $Filename = Split-Path $LiteralPath -Leaf
                $SafePath = Join-Path 'TestDrive:' $Filename
                & $RealCmd -InputObject $InputObject -LiteralPath $SafePath
            }

            Mock Set-ItemProperty {
                $RealCmd = Get-Command Set-ItemProperty -CommandType Cmdlet
                $Filename = Split-Path $LiteralPath -Leaf
                $SafePath = Join-Path 'TestDrive:' $Filename
                & $RealCmd -LiteralPath $SafePath -Name $Name -Value $Value
            }

            $Rand = 'Random', (Get-Random) -join ''
            $ExpectDate = Get-Date -AsUTC
            $ExpectPath = Join-Path 'TestDrive:' "$Rand.xml"
        }

        It 'Writes CLIXML to data store file' {
            $ExpectPath | Should -Not -Exist
            Set-DataStore -Name $Rand -Data @{ a = 1 } -Update $ExpectDate
            $ExpectPath | Should -Exist
        }

    }

}
```

In this example, the mocks modify incoming file paths to ensure data is only written to Pester's test drive. They then pass on the safe values to the underlying dependencies that `Get-Command` returned. Additional tests confirm that `Set-DataStore` has written the correct data to the test drive as if it was a real data store location.

Example 35: More tests to confirm the contents of the new test drive file

```
# Same Context block as Example 33
It 'Writes correct date to data store attributes' {
    $File = Get-Item -LiteralPath $ExpectPath
    $File.LastWriteTimeUtc | Should -Be $ExpectDate
}

It 'Stores PSCustomObject with property a = 1' {
    [xml]$Data = Get-Content -LiteralPath $ExpectPath
    $Types = $Data.Objs.Obj.TN.ChildNodes
    $Types.Count | Should -Be 2
    $Types[0].'#text' |
        Should -Be 'System.Management.Automation.PSCustomObject'
    $Values = $Data.Objs.Obj.MS.ChildNodes
    $Values.Count | Should -Be 1
    $Values[0].Name | Should -Be 'I32'
    $Values[0].N | Should -Be 'a'
    $Values[0].'#text' | Should -Be '1'
}
```

Run the tests to confirm that you can access underlying dependencies in Pester.

Example 36: Running the dependency access tests

```
Invoke-Pester .\Examples34and35.Tests.ps1 -Output Detailed
```

```
Pester v5.3.0

Starting discovery in 1 files.
Discovery found 3 tests in 13ms.
Running tests.

Running tests from 'Examples34and35.Tests.ps1'
Describing Set-DataStore unit tests
 Context Correct data
   [+] Writes CLIXML to data store file 10ms (8ms|2ms)
   [+] Writes correct date to data store attributes 1ms (1ms|0ms)
   [+] Stores PSCustomObject with property a = 1 14ms (13ms|0ms)
Tests completed in 99ms
Tests Passed: 3, Failed: 0, Skipped: 0 NotRun: 0
```

Find the Examples34and35.Tests.ps1[29] file used in this example on the Extras[30] repository on GitHub.

4.8 Removing Parameter Typecasting and Validation

As the chapter has already mentioned, Pester 5.0 copies the parameters of the dependencies that you mock. This includes the validation and parameter attributes, and means that tests must pass valid parameter sets to mocks, even if the mock doesn't need them. This becomes problematic in scenarios where you want simple mocks for complex dependencies.

Imagine you are mocking the `Set-DataStore` function. This function accepts three parameters, all of which are mandatory; `$Name` (a string), `$Data` (a hashtable), and `$Update` (a DateTime object).

The `$Update` parameter also has a validation script that checks if the date is on or after January 1st, 2000:

[29] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/blob/main/Edition-01/Mocking/DataStoreDemo/Examples34and35.Tests.ps1

[30] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/tree/main/Edition-01/

```
[ValidateScript({
    $_ -ge [DateTime]::new(2000, 1, 1, 0, 0, 0, 0, 1)
})]
[datetime]$Update
```

If you mock `Set-DataStore` and pass invalid parameters to it, the call throws an error.

Example 37: Pester copies parameter typecasting and validation from the real dependency for mocks

```
Describe 'Parameter validation and typecasting in mocks' {

    It 'Mock calls with invalid parameters throw errors' {

        Mock Set-DataStore {
            # Does nothing
        }

        # Bad types for -Data and -Update
        { Set-DataStore -Name 'AName' -Data '' -Update '' } |
            Should -Throw

        # The value of -Update is before 2000
        { Set-DataStore -Name 'AName' -Data @{} -Update ([datetime]0) } |
            Should -Throw

    }

}
```

The `Mock` statement includes two parameters that can make things a little simpler in these scenarios.

The `-RemoveParameterType` parameter sets the required type of the parameter value to `Object`, from which all PowerShell objects derive. Any parameters you pass to this in your mock definition will now accept any type. In other words, you've disabled typecasting for those parameters.

The `-RemoveParameterValidation` parameter, on the other hand, removes the `Validate*` attributes from the parameters you pass. This includes the `[ValidateScript({...})]` for the `-Update` parameter of `Set-DataStore`.

Example 38: Removing typecasting and parameter validation from mocks

```
Describe 'Removing parameter validation and typecasting' {

    BeforeAll {
        Mock Set-DataStore {
            # Does nothing
        } -RemoveParameterValidation Update -RemoveParameterType Update, Data
    }

    It 'Succeeds when validation and typecasting are removed' {

        Set-DataStore -Name 'AName' -Data '' -Update ''
        Set-DataStore -Name 'AName' -Data @{} -Update ([datetime]0)

    }

}
```

The `-Name` parameter still retains its *string* type, as the mock doesn't specify this one for typecasting or validation removal. Note also that neither of these parameters makes mandatory parameters optional. You must still pass non-null values for them, even with validation and typecasting disabled.

Example 39: The Mandatory attribute isn't affected by validation or typecasting removal

```
# Same Describe block as Example 38
It 'Null values always fail for mandatory parameters' {

    # Name wasn't included in -RemoveParameterValidation
    # or -RemoveParameterType
    { Set-DataStore -Name '' -Data '' -Update '' } |
        Should -Throw

    # Update was included in both, but still can't accept $null
    { Set-DataStore -Name 'AName' -Data '' -Update $null } |
        Should -Throw

}
```

Run the tests to check that the statements do indeed throw errors as expected:

Example 40: Running the parameter validation and typecasting tests

```
Invoke-Pester .\Examples37to39.Tests.ps1 -Output Detailed
```

```
Pester v5.3.0

Starting discovery in 1 files.
Discovery found 3 tests in 9ms.
Running tests.

Running tests from 'Examples37to39.Tests.ps1'
Describing Parameter validation and typecasting in mocks
  [+] Mock calls with invalid parameters throw errors 6ms (4ms|2ms)

Describing Removing parameter validation and typecasting
  [+] Succeeds when validation and typecasting are removed 4ms (2ms|2ms)
  [+] Null values always fail for mandatory parameters 3ms (2ms|0ms)
Tests completed in 83ms
Tests Passed: 3, Failed: 0, Skipped: 0 NotRun: 0
```

Find the Examples37to39.Tests.ps1[31] file used in this example on the Extras[32] repository on GitHub.

[31] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/blob/main/Edition-01/Mocking/DataStoreDemo/Examples37to39.Tests.ps1

[32] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/tree/main/Edition-01/

4.9 Mocking Native Applications

You can mock native applications with Pester in almost the same way as you can PowerShell functions and cmdlets. The difference arises with parameters and parameter filters. Native applications don't have a standardized parameter handling mechanism, and nor does PowerShell when you make calls to those applications. You still have full access to command line arguments, however, in the form of the `$args` variable.[33] This is the standard approach for accessing unbound arguments in PowerShell, and the same is true for Pester tests. The `$args` variable is an array of arguments, roughly equivalent to those you would separate with spaces on the command line.

If some command line arguments were `-a 1 -b 2`, the `$args` variable would be:

```
@('-a', '1', '-b', '2')
```

Note how the *keys* and *values* common to many applications and also seen in PowerShell parameters are individual items in the `$args` array. When mocking native applications, you need to apply your own argument/parameter processing logic.

Example 41: Mocking a native application (tar) in Pester

```
Describe 'Mocking native applications' {

    BeforeAll {
        $MockFile = 'HelloGround.txt'
        Mock tar {
            $fileArg = -1
            for ($i = 0; $i -lt $args.Count; $i++) {
                if ($args[$i] -match '-\w*f') {
                    $fileArg = $i + 1
                    break
                }
            }
            if ($fileArg -lt 0) { throw 'No archive file passed' }
            $FilePath = (Join-Path 'TestDrive:' $MockFile)
            Set-Content $FilePath 'Oh no, not again'
            if (($args -match '-\w*v').Count -gt 0) {
                return "x $MockFile"
            }
        } -ParameterFilter {
            ($args -like '-x*').Count -gt 0
        }

        Mock tar {
            return $MockFile
        } -ParameterFilter {
            ($args -like '-t*').Count -gt 0
        }
    }

}
```

[33] Microsoft. (2022, Jan. 07). *About Automatic Variables (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_automatic_variables#args. [Accessed: Jul. 04, 2022].

`tar` is an application used to create and extract from archive files. It's available in *nix and in current Windows 10/11 environments.

The two mocks have parameter filters that check the array of arguments for their respective flags (`-x` and `-t`). The mock for `-x` also checks for the `v` flag, which would print the names of extracted files with the real dependency.

Example 42: Using mocks of native applications in tests

```
# Same Describe block as Example 40
It 'Lists mock archive' {
    tar -tf test.zip | Should -Be $MockFile
}

It 'Extracts mock file' {
    tar -xvf test.zip | Should -Be "x $MockFile"
    $File = Join-Path 'TestDrive:' $MockFile
    $File | Should -Exist
    $File | Should -FileContentMatch 'Oh no, not again'
}

It 'Only calls tar in list or extract mode' {
    Should -Invoke tar -Exactly 2 -ExclusiveFilter {
        ($args -like '-t*').Count -gt 0 -or
        ($args -like '-x*').Count -gt 0
    } -Scope Describe
}
```

As Example 42 shows, you can also use the `Should -Invoke` assertions with mocks of native applications. Any filters must again handle arguments through the `$args` variable.

Example 43: Running the native applications tests

```
Invoke-Pester .\Examples41and42.Tests.ps1 -Output Detailed
```

```
Pester v5.3.0

Starting discovery in 1 files.
Discovery found 3 tests in 8ms.
Running tests.

Running tests from 'Examples41and42.Tests.ps1'
Describing Mocking native applications
  [+] Lists mock archive 7ms (6ms|2ms)
  [+] Extracts mock file 7ms (6ms|1ms)
  [+] Only calls tar in list or extract mode 2ms (2ms|0ms)
Tests completed in 84ms
Tests Passed: 3, Failed: 0, Skipped: 0 NotRun: 0
```

Find the Examples41and42.Tests.ps1[34] file used in this example on the Extras[35] repository on GitHub.

[34] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/blob/main/Edition-01/Mocking/DataStoreDemo/Examples41and42.Tests.ps1

[35] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/tree/main/Edition-01/

4.10 Mocking .NET Objects with New-MockObject

One aspect that the chapter hasn't covered at all yet is .NET objects. PowerShell and .NET are tightly interconnected and, as such, you'll probably come across .NET objects that you need to mock for the purposes of testing.

Pester can create mock objects from any .NET type that exists in the current scope. The standard syntax for this is:

```
New-MockObject -Type 'System.Type.Name' [-Properties ...] [-Methods ...]
```

- `-Properties` accepts a hashtable of property names and values.
- `-Methods` accepts a hashtable of method names and script blocks that act as the methods themselves.

Example 44: Mocking a .NET object with New-MockObject

```
$MockParams = @{
    Type = 'System.Globalization.CultureInfo'
    Properties = @{
        Parent = 'en'
        LCID   = 1931
        Name   = 'en-mm'
        DisplayName = 'Mocked Culture'
    }
    Methods = @{
        GetConsoleFallbackUICulture = {
            return Get-Culture -Name 'en-US'
        }
    }
}
$Culture = New-MockObject @MockParams

# Example 44a: A real CultureInfo
Get-Culture

# Example 44b: The mocked one
$Culture

# Example 44c: A mocked method
$Culture.GetConsoleFallbackUICulture()
```

```
# Example 44a:
LCID             Name             DisplayName
----             ----             -----------
2057             en-GB            English (United Kingdom)

# Example 44b:
LCID             Name             DisplayName
----             ----             -----------
1931             en-mm            Mocked Culture

# Example 44c:
LCID             Name             DisplayName
----             ----             -----------
1033             en-US            English (United States)
```

Another feature of mocked objects is that you can retrieve the method call history for all methods. By default, Pester stores the history as a property of the mocked object, with the same name as the method but prefixed with an underscore _.

Example 45: Retrieving the call history of a mocked .NET object

```
# Run it a second time, this time with a parameter
$null = $Culture.GetConsoleFallbackUICulture($true)
# Retrieve the call history
$Culture._GetConsoleFallbackUICulture
```

```
Call Arguments
---- ---------
   1 {}
   2 {True}
```

The two method calls (the first made in Example 44) now show up in the `_GetConsoleFallbackUICulture` member of the mocked object. Each call has a number starting from *1* and an array of arguments that the call passed to the mocked method.

To change the prefix that Pester uses for this method call history, use the `-MethodHistoryPrefix` parameter.

Example 46: Using an alternate history prefix for mocked .NET object methods

```
$MockSpan = New-MockObject -Type timespan -Methods @{
    Test = {
        param ($Value)
        $Value
    }
} -MethodHistoryPrefix '##'

$MockSpan.Test('Allons-y, Alonzo!')

$MockSpan.'##Test'
```

```
Allons-y, Alonzo!

Call Arguments
---- ---------
   1 {Allons-y, Alonzo!}
```

As is apparent in Example 46, you can use type accelerators with `New-MockObject`. In fact, you can pass almost all the type names you would to `New-Object`.

There are some exceptions. The .NET type must support uninitialized object creation by the serialization library.[36] However, since mock objects are most useful for complex data types, you shouldn't need to mock primitives.

For objects that can't be created without calling a constructor, you can instantiate the object normally and pass it to `New-MockObject` using the `-InputObject` parameter. This doesn't work with all .NET types, so checking outside of a Pester test might be necessary.

4.11 Next Steps

Check out the Modern IT Automation with PowerShell Extras[37] repository on GitHub. The extras repository contains chapter examples, extra code, and additional materials used in the chapters of this book.

Additional material for this chapter includes the Data Store Demo Functions[38] that the tests are based on.

You can also use the Static Dummy API[39] with the `Update-DataStore` function. Change *001* in the URL to access alternate stores of various sizes, all with randomly generated dummy data (*002–008*).

4.12 Further Reading

- Official Pester Mocking Documentation[40]
- `Mock` Documentation[41]
- `Should` Documentation[42]
- `New-MockObject` Documentation[43]
- `InModuleScope` Documentation[44]
- Pester Migration Guide — v3 to v4[45]

[36] Pester Team. (2022, Jun. 30). *Pester - New-MockObject.ps1*. L90-L95. Pester/Pester on GitHub. [Online]. Available: https://github.com/pester/Pester/blob/main/src/functions/New-MockObject.ps1. [Accessed: Jul. 04, 2022].

[37] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/tree/main/Edition-01/

[38] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/tree/main/Edition-01/Mocking/DataStoreDemo/

[39] https://github.com/TheFreeman193/mita-dummyapi/blob/main/README.md

[40] https://pester.dev/docs/usage/mocking

[41] https://pester.dev/docs/commands/Mock

[42] https://pester.dev/docs/commands/Should

[43] https://pester.dev/docs/commands/New-MockObject

[44] https://pester.dev/docs/commands/InModuleScope

[45] https://pester.dev/docs/migrations/v3-to-v4

- Pester Migration Guide — v4 to v5[46]
- Pester Migration Guide — Breaking Changes in v5[47]

[46] https://pester.dev/docs/migrations/v4-to-v5
[47] https://pester.dev/docs/migrations/breaking-changes-in-v5

5. Unit Testing

5.1 Why Unit Testing?

Whether you're writing PowerShell scripts or modules for an enterprise or simply for fun, you should use a Git repository (repo) to store your PowerShell files. Even if it's only you accessing the repo, this habit is worth forming. Storing files in Git ensures that there's a traceable history of all changes, and it doubles as a backup strategy, giving you the power to revert any code changes.

While people usually store their scripts in Git repos, they often overlook the need to support unit testing. Usually, this is due to time or budget constraints in enterprise environments, or developers don't consider adding tests a worthwhile use of their time for personal projects.

So, why should you do this? Why should you care? Spending the time and effort to create unit tests may seem overkill and unnecessary. For small repos, this may be true. But often, over time, scripts in repos grow in size and complexity. Adding support for unit testing early on makes creating new tests easier; you can simply update existing ones as repo complexity increases. If you delay the addition of unit tests until your repo is more complex, adding them is more complicated and time-consuming. A bonus is the warm and fuzzy feeling you'll get when your repo tests all pass!

5.2 What Is Unit Testing?

Unit tests should test the code in the Git repository and shouldn't rely on external interfaces. This is so the unit tests can be run via the testing framework anytime without requiring external systems to be up and running. However, if external endpoints are used, and developers wish to test how scripts handle the returned data, they can be mocked out,[1] so there's no need to use actual endpoints when executing the unit tests. One advantage of mocking is that it speeds up unit test response times, as there's no latency of external systems to consider.

For more information on mocking, refer to Mocking.

Running unit tests with a limited scope should be quick, allowing for frequent execution so developers can gain more confidence that their script changes have broken nothing.

[1] Pester Team. (2022, Jun. 25). *Mocking with Pester*. Pester Docs. [Online]. Available: https://pester.dev/docs/usage/mocking. [Accessed: Sep. 04, 2022].

5.3 Testing Frameworks

Developers with experience in programming languages like C# are probably familiar with the many testing frameworks currently available. These include xUnit[2], NUnit[3], MSTest[4], Specflow[5], and others. So what testing frameworks are there for PowerShell?

Pester[6] is the de facto testing framework to use with PowerShell. It has been around for many years and is currently up to version 5.3 at the time of writing. Version 5.0 of Pester was a significant rewrite of much of the codebase.[7] This new version has simplified Pester syntax, making it easier to learn.

This chapter will focus only on Pester v5 features and examples.

5.3.1 Black Box vs. White Box Testing

Black Box testing is when the tester lacks visibility into the inner workings of the code. They can call the component and view the output or results, but that's the limit of the interaction.

White Box testing is when the tester has full access to the code of the tested component. Test cases can be structured to pass through different execution flows to ensure that many scenarios are executed. Another definition of white box testing is unit testing, since these tests can check portions of internal code and code flows. Therefore, unit testing or white box testing is the focus of the content for the rest of this chapter.

5.3.2 The AAA Approach

A common approach for structuring tests is the AAA approach (Arrange, Act, and Assert). It has proven popular because of its simplicity of structuring tests into three easy-to-follow sections.

- **Arrange**: Setup any data or conditions for tests.
- **Act**: Run the commands for the test itself.
- **Assert**: Performs checks on the results to determine if they conform to the pass/fail criteria.

For more information on AAA, refer to The AAA Approach.

[2] https://xunit.net/

[3] https://nunit.org/

[4] https://learn.microsoft.com/en-us/visualstudio/test/using-microsoft-visualstudio-testtools-unittesting-members-in-unit-tests?view=vs-2022

[5] https://specflow.org/

[6] https://pester.dev/

[7] Pester Team. (2021, May. 14). *Breaking Changes in v5.* Pester Docs. [Online]. Available: https://pester.dev/docs/migrations/breaking-changes-in-v5. [Accessed: Apr. 26, 2022].

5.4 Pester

5.4.1 Getting Started

Given the scenario of a developer or scripter creating a new Git repo with some added functions and modules, start creating unit tests before the scripts get too complex. So where do you start? If you haven't already installed Pester, open a PowerShell prompt and run the following command.

Example 1: Installing the latest version of Pester

```
Install-Module -Name Pester -Force -SkipPublisherCheck
```

The `-Force` and `-SkipPublisherCheck` switches are usually required to install the latest version of Pester on Windows. This is because version 3.4.0 is installed by default and is the latest version that Microsoft signed. Later versions are signed with a different certificate, so adding these switches is necessary.[8]

If you're struggling to install the latest version of Pester, refer to the installation documentation[9] for alternative installation approaches and troubleshooting.

Run `Get-InstalledModule` to confirm a successful installation, and `Get-Module` to check the version of Pester loaded into the PowerShell session.

Example 2: Checking the Pester module version

```
# Example 2a: Checking Pester is installed from the PowerShell Gallery
Get-InstalledModule -Name Pester

# Example 2b: Checking the Pester version in the session
Import-Module -Name Pester
Get-Module -Name Pester
```

```
# Example 2a:
Version Name   Repository Description
------- ----   ---------- -----------
5.3.3   Pester PSGallery  Pester provides a framework for running…

# Example 2b:
ModuleType Version PreRelease Name   ExportedCommands
---------- ------- ---------- ----   ----------------
Script     5.3.3              Pester {Add-ShouldOperator, AfterAll, …
```

[8] Pester Team. (2022, Jun. 22). *Installation and Update.* Pester Docs. [Online]. Available: https://pester.dev/docs/introduction/installation. [Accessed: Sep. 09, 2022].

[9] https://pester.dev/docs/introduction/installation

5.4.2 Defining Pester Test Files

The standard approach for the location of test files is to store them in the same folder as the source code.[10] The file names are the same as those of the source code files, but with a `.Tests.ps1` extension. For example, if the script `Calculator.ps1` contains the code to test, the tests would be in a `Calculator.Tests.ps1` file.

This file placement is suitable when a relatively small number of files require testing.

An alternative approach is to define tests in a `tests` folder that follows the same directory structure as the source files directory:

```
/src/functions/Calculator.ps1
/tests/functions/Calculator.Tests.ps1
```

This second approach is more useful when you have dozens of files to test.

You can use your own approach to store Pester test files, but not following one of the standard approaches adds complexity for other maintainers who are familiar with the standard. There can also be issues with code editors or plugins that expect particular filenames and locations for determining how to handle and display files. For example, the PowerShell plugin within Visual Studio Code won't recognize Pester files correctly if the filename doesn't end with `.Tests.ps1`.

5.4.3 Pester Demo Code

The rest of this chapter uses the code below for all examples. Assume this code is in a file `Calculator.ps1` and the tests for it are in a file `Calculator.Tests.ps1` in the same folder.

Example 3: Arithmetic functions that add or subtract an array of integers

```
function Invoke-Addition {
    param (
        [Parameter(Mandatory)]
        [ValidateCount(1, 20)]
        [int[]]$Numbers
    )

    $Result = 0
    $Numbers.ForEach{ $Result += $_ }

    $Result
}

function Invoke-Subtraction {
    param (
        [Parameter(Mandatory)]
        [ValidateCount(2, 20)]
        [int[]]$Numbers
    )

    $Result = $Numbers[0]
    $Numbers | Select-Object -Skip 1 | ForEach-Object { $Result -= $_ }

    $Result
}
```

[10] Pester Team. (2022, Jun. 19). *File placement and naming*. Pester Docs. [Online]. Available: https://pester.dev/docs/usage/file-placement-and-naming. [Accessed: Aug. 18, 2022].

5.4.4 Pester Test Structure

If you're new to Pester, refer to the earlier chapters on The AAA Approach and Mocking to gain an understanding of the Pester test file structure.

5.4.4.1 Basic Tests

Below is a sample Pester test file demonstrating simple, clear, and clean test cases. Documentation is barely required, as the code itself reads almost like English.

Example 4: A simple test of the Invoke-Addition function from Calculator.ps1

```
BeforeAll {
    . $PSCommandPath.Replace('.Tests.ps1', '.ps1')
}

Describe 'Invoke-Addition' {

    It 'Correctly adds the numbers passed' {
        # Arrange
        $TestNums = @(1, 2, 3, 4, 5)

        # Act
        $Sum = Invoke-Addition -Numbers $TestNums

        #Assert
        $Sum | Should -Be 15
    }

}
```

```
Starting discovery in 1 files.
Discovery found 1 tests in 7ms.
Running tests.

Running tests from 'Calculator.Tests.ps1'
Describing Invoke-Addition
  [+] Correctly adds the numbers passed 16ms (1ms|15ms)
Tests completed in 83ms
Tests Passed: 1, Failed: 0, Skipped: 0 NotRun: 0
```

5.4.4.2 Detailed Tests

From here, you can define more complex tests. This chapter touches on some more common enhancements that you can add. However, if you require more detail, refer to the Pester documentation[11].

5.4.4.2.1 Tags

You can add tags to your tests as a parameter to the `Describe`, `Context`, and `It` blocks to provide grouping.[12] A good example of this is when you only want to run tests tagged with "Unit" locally and leave "Integration" tagged tests for automated deployments.

[11] https://pester.dev/docs/quick-start

[12] Pester Team. (2022, Jun. 19). *Tags*. Pester Docs. [Online]. Available: https://pester.dev/docs/usage/tags. [Accessed: Sep. 04, 2022].

Example 5: Adding a tag to a Describe block

```
Describe 'Invoke-Addition' -Tag 'Unit' {
    It 'Correctly adds the numbers passed' {
        # Test code
    }
}
```

5.4.4.3 -TestCases

If you need to test many scenarios, the `-TestCases` parameter is more suitable than creating a separate `It` block for each one.[13] Defining the `-TestCases` parameter with rows of data in a single `It` block allows multiple tests to be executed with one code block. You can use an additional field in the `-TestCases` data to display accurate test information.

Example 6: Using test cases to reuse test code with various inputs

```
Describe 'Invoke-Addition' -Tag 'Unit' {

    It 'Returns a result of <Sum> given <Desc> (<Values>)' -TestCases @(
        @{ Desc = 'numbers 1 to 5'; Values = @(1, 2, 3, 4, 5); Sum = 15 }
        @{ Desc = 'negative numbers'; Values = @(-2, -4, -9); Sum = -15 }
        @{ Desc = 'only two numbers'; Values = @(10, 5); Sum = 15 }
        @{ Desc = 'only one number'; Values = @(64); Sum = 64 }
    ) {
        # Act
        $Result = Invoke-Addition -Numbers $Values

        # Assert
        $Result | Should -Not -BeNullOrEmpty
        $Result | Should -Be $Sum
    }

}
```

```
Starting discovery in 1 files.
Discovery found 4 tests in 5ms.
Running tests.

Running tests from 'Calculator.Tests.ps1'
Describing Invoke-Addition
  [+] Returns a result of 15 given numbers 1 to 5 (1 2 3 4 5) 3ms (1ms|2ms)
  [+] Returns a result of -15 given negative numbers (-2 -4 -9) 2ms (1ms|0ms)
  [+] Returns a result of 15 given only two numbers (10 5) 2ms (2ms|0ms)
  [+] Returns a result of 64 given only one number (64) 2ms (2ms|0ms)
Tests completed in 67ms
Tests Passed: 4, Failed: 0, Skipped: 0 NotRun: 0
```

[13] Pester Team. (2019, Jan. 09). *It - TestCases*. Pester Docs. [Online]. Available: https://pester.dev/docs/commands/It#-testcases. [Accessed: Apr. 26, 2022].

5.4.5 Mocking

Code that you're unit testing can sometimes call other external methods or servers. These external dependencies may not always be available or could have access limitations; for example, rate limits or long processing times. Pester can *mock out* calls to these dependencies to test the internal code without worrying about any external impact.

For more detail on how to mock external interfaces and even internal cmdlets, refer to the chapters on Mocking and The AAA Approach.

5.4.6 Running Pester Tests

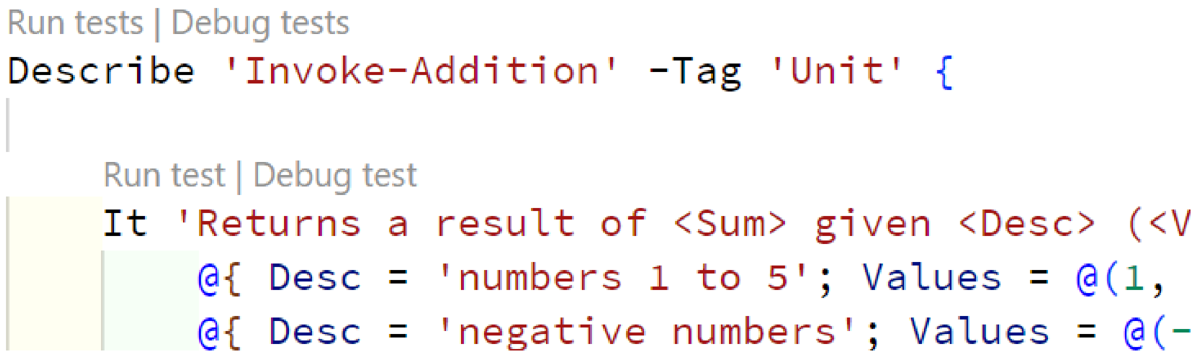
You've now defined your tests within the `*.Tests.ps1` files, but how do you run them? There are two recommended approaches.

5.4.6.1 Visual Studio Code

Several text editors are suitable for PowerShell code, but you're likely developing on Windows, Linux, or macOS. The most used (and supported) PowerShell editor is Visual Studio Code[14]. It's free and open-source,[15] cross-platform, flexible, and robust, thanks to a vast library of extensions. With the free PowerShell plugin[16], Pester support is built into the editor itself. You can see it by opening test files in VS Code: `Run Tests | Debug Tests` should appear above any `Describe`, `Context` or `It` blocks. These two links will execute the tests defined in the related block when clicked.

A different level of verbosity is defined between the `Run Tests` and `Debug Tests` links. `Debug Tests` sets the `-Output` parameter to **Diagnostic** to provide additional logging details to assist with debugging the tests.

The output of the tests will appear in the Terminal output pane, which is usually below the editor.



```
Describe 'Invoke-Addition' -Tag 'Unit' {

    It 'Returns a result of <Sum> given <Desc> (<V
        @{ Desc = 'numbers 1 to 5'; Values = @(1,
        @{ Desc = 'negative numbers'; Values = @(-
```

Pester Test Buttons in VS Code

[14] https://code.visualstudio.com/

[15] While the codebase for Visual Studio Code is open-source, some elements of the binary releases and some plugins aren't. You can learn more at https://github.com/microsoft/vscode/.

[16] https://marketplace.visualstudio.com/items?itemName=ms-vscode.PowerShell

5.4.6.2 Command Line

The Visual Studio Code approach is ideal for testing a single `Describe` block. To run all the tests or define advanced settings like filtering options, use the command line approach.

The command to run Pester tests is `Invoke-Pester`. Using the example functions defined above, you can run tests for them by passing the file path of a test file or a wildcard pattern.

Example 7: Running all test files in the current working directory

```
Invoke-Pester -Path *.Tests.ps1
```

```
Starting discovery in 1 files.
Discovery found 4 tests in 7ms.
Running tests.
[+] Calculator.Tests.ps1 55ms (7ms|41ms)
Tests completed in 57ms
Tests Passed: 4, Failed: 0, Skipped: 0 NotRun: 0
```

This is suitable for running all the tests, but it doesn’t show much detail in the output. To get more detail, add the `-Output Detailed` parameter.

Example 8: Running tests with the -Output parameter to get detailed results

```
Invoke-Pester -Path *.Tests.ps1 -Output Detailed
```

```
Pester v5.3.3

Starting discovery in 1 files.
Discovery found 4 tests in 7ms.
Running tests.

Running tests from 'Calculator.Tests.ps1'
Describing Invoke-Addition
  [+] Returns a result of 15 given numbers 1 to 5 (1 2 3 4 5) 4ms (1ms|2ms)
  [+] Returns a result of -15 given negative numbers (-2 -4 -9) 2ms (1ms|0ms)
  [+] Returns a result of 15 given only two numbers (10 5) 2ms (1ms|0ms)
  [+] Returns a result of 64 given only one number (64) 3ms (2ms|0ms)
Tests completed in 64ms
Tests Passed: 4, Failed: 0, Skipped: 0 NotRun: 0
```

The output provides more details for each test in the `Describe` block.

To filter which tests are executed by tag, use the `-TagFilter` and `-ExcludeTagFilter` parameters. Since all tests in this chapter are tagged with ‘Unit’, adding another value to `-TagFilter` means no tests run.

Example 9: Running only the tests in blocks with the 'Integration' tag

```
Invoke-Pester -Path *.Tests.ps1 -TagFilter Integration -Output Detailed
```

```
Pester v5.3.3

Starting discovery in 1 files.
Discovery found 4 tests in 6ms.
Filter 'Tag' set to ('Integration').
Filters selected 0 tests to run.
Running tests.
Tests completed in 4ms
Tests Passed: 0, Failed: 0, Skipped: 0 NotRun: 4
```

5.4.7 Pester Configuration

Pester 5.0 introduces a `PesterConfiguration` object, which contains all configuration settings and includes descriptions and default values.[17] Use it with the `-Configuration` parameter of `Invoke-Pester` to replace all other parameters in the command. It's recommended for use with more complex calls to wrap up all the settings into a single object. There are many parameters to `Invoke-Pester` marked as deprecated,[18] so it's worthwhile being familiar with the configuration object in case the deprecated parameters are removed in future releases.

This is a replacement for running the tests using splatting.

Example 10: Running Invoke-Pester using splatting

```
$Params = @{
    Path      = './scripts/build'
    TagFilter = 'Unit', 'Integration'
    Output    = 'Detailed'
    PassThru  = $true
}

$Results = Invoke-Pester @Params
```

Use the `PesterConfiguration` object with the `-Configuration` parameter of `Invoke-Pester`.

[17] Pester Team. (2022, Jun. 19). *Configuration*. Pester Docs. [Online]. Available: https://pester.dev/docs/usage/configuration. [Accessed: Sep. 04, 2022].

[18] Pester Team. (2022, Apr. 29). *Invoke-Pester*. Pester Docs. [Online]. Available: https://pester.dev/docs/commands/Invoke-Pester. [Accessed: Sep. 11, 2022].

Example 11: Running Invoke-Pester with a Pester configuration object

```
$Config = New-PesterConfiguration
$Config.Run.PassThru = $true
$Config.Run.Path = './scripts/build'
$Config.Filter.Tag = 'Unit', 'Integration'
$Config.Output.Verbosity = 'Detailed'

$Results = Invoke-Pester -Configuration $Config
```

To learn more about Pester configuration, refer to the Configuration[19] help article on Pester Docs. For details on other parameters of `Invoke-Pester`, refer to the Invoke-Pester[20] command article.

5.4.8 Pester Automation

While it's helpful to run Pester tests locally to confirm everything works as expected, it's crucial to implement them as part of automated checks during a *pull request* for script changes.

A pull request is when a developer wishes to merge the Git branch containing their changes with the main branch of the Git repository. Other developers can review the changes and offer comments and feedback. At the same time, automation checks can run on the branch to ensure that code compiles and unit tests execute successfully.

These automated tests can now be so effective that there are suggestions from industry leaders to merge automatically any pull requests that pass automated builds and tests without errors.[21] [22] If a developer needs a code review, they can request it when creating the pull request, but it would no longer be mandatory.

The thinking here is that this approach speeds up pull requests for simple changes that don't need manual review and approval. When reviewers must approve every pull request manually, there's a risk of a code review bottleneck. While there's nothing to compile with PowerShell scripts, there are other tests to carry out like syntax and linting checks with tools like PSScriptAnalyzer[23] or running unit tests with Pester.

The two primary DevOps systems in the Microsoft ecosystem are GitHub[24] and Azure DevOps[25]. Both platforms contain hosted agents that already have Pester preinstalled, so no additional setup is required to run Pester tests as part of pull request checks.

You can run the Pester tests in a single pipeline step and process the output. However, there are some differences in the process between Azure DevOps and GitHub.

[19] https://pester.dev/docs/usage/configuration

[20] https://pester.dev/docs/commands/Invoke-Pester

[21] Alex Chan. (2019, Mar. 25). *Creating a GitHub Action to auto-merge pull requests*. alexwlchan. [Online]. Available: https://alexwlchan.net/2019/03/creating-a-github-action-to-auto-merge-pull-requests/. [Accessed: Sep. 11, 2022].

[22] GitHub. (2022, Sep. 01). *Automatically merging a pull request*. GitHub Docs. [Online]. Available: https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/incorporating-changes-from-a-pull-request/automatically-merging-a-pull-request. [Accessed: Sep. 11, 2022].

[23] https://github.com/PowerShell/PSScriptAnalyzer

[24] https://resources.github.com/devops/

[25] https://azure.microsoft.com/products/devops/

5.4.8.1 Azure DevOps

Azure DevOps defaults to using YAML pipelines for CI/CD automation.[26]

YAML is a human-readable data-serialization language.[27]

The following YAML script can be added to a repo containing the PowerShell scripts used in the example above. The code below assumes `Calculator.ps1` and `Calculator.Tests.ps1` files in a `scripts` folder in the root folder of a repo stored in Azure DevOps.

Example 12: YAML definition of an Azure DevOps pipeline

```
name: PesterTests
trigger: none
jobs:
  # Runs Pester tests on the PowerShell Script in the scripts folder
  - job: RunPesterTests
    displayName: "Run Pester Tests"
    pool:
      vmImage: "ubuntu-latest"
    steps:
      - pwsh: |
          $Config = New-PesterConfiguration
          $Config.Run.Path = 'scripts/*.Tests.ps1'
          $Config.Filter.Tag = 'Unit'
          $Config.TestResult.OutputPath = 'Test-Pester.xml'
          $Config.TestResult.Enabled = $true
          Invoke-Pester -Configuration $Config
        displayName: "Invoke Pester"
        failOnStderr: false

      - task: PublishTestResults@2
        displayName: "Publish Test Results"
        inputs:
          testResultsFormat: "NUnit"
          testResultsFiles: "**/Test-Pester.xml"
          mergeTestResults: true
          failTaskOnFailedTests: true
```

There are two steps in this workflow. The first is a PowerShell step that runs the `Invoke-Pester` command similar to how you would run it locally. The main difference to the locally run command is the addition of the `-OutputFile './Test-Pester.xml'`, which will store the output of the test into the current folder in the default *NUnit XML* format.[28]

The `failOnStderr` argument set to `false` means the PowerShell step won't fail if there's an error. The reason for this is so the second step can execute and store the failed test report as a build artifact so it can be viewed for assessment.

[26] Microsoft. (2022, Apr. 27). *Template types & usage.* Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/azure/devops/pipelines/process/templates?view=azure-devops. [Accessed: Sep. 16, 2022].

[27] YAML Website Contributors. (2021, Nov. 17). *The Official YAML Web Site.* YAML.org. [Online]. Available: https://yaml.org/. [Accessed: Sep. 11, 2022].

[28] The NUnit Project. (2022). *XML Formats.* NUnit Docs. [Online]. Available: https://docs.nunit.org/articles/nunit/technical-notes/usage/XML-Formats.html. [Accessed: Sep. 11, 2022].

The second step uses a preconfigured task and contains properties, including the test output type (*NUnit*) and the location of the first step's output file to use for the test data. This step will fail the build if there's a test failure in the results (`failTaskOnFailedTests`), as it's not ideal to continue with the workflow defined in the pipeline.

> If you prefer not to run a raw PowerShell command in the Azure pipeline, there's a preconfigured task[29] available from the Pester project.

The pipeline doesn't have a trigger, because a Git branch policy will trigger it. To make it selectable in a Git branch policy, you must add it to Azure DevOps first.

5.4.8.1.1 Add as a Pipeline

- In Azure DevOps, go to **Pipelines** → **Pipelines** and click on the **New Pipeline** button.
- Select **Azure Repos Git**.
- Select the name of the repo.
- Select **Existing Azure pipelines YAML file**.
- Select the branch and the path and click on **Continue**.
- You can review the resulting YAML code before clicking **Run** to test the pipeline.

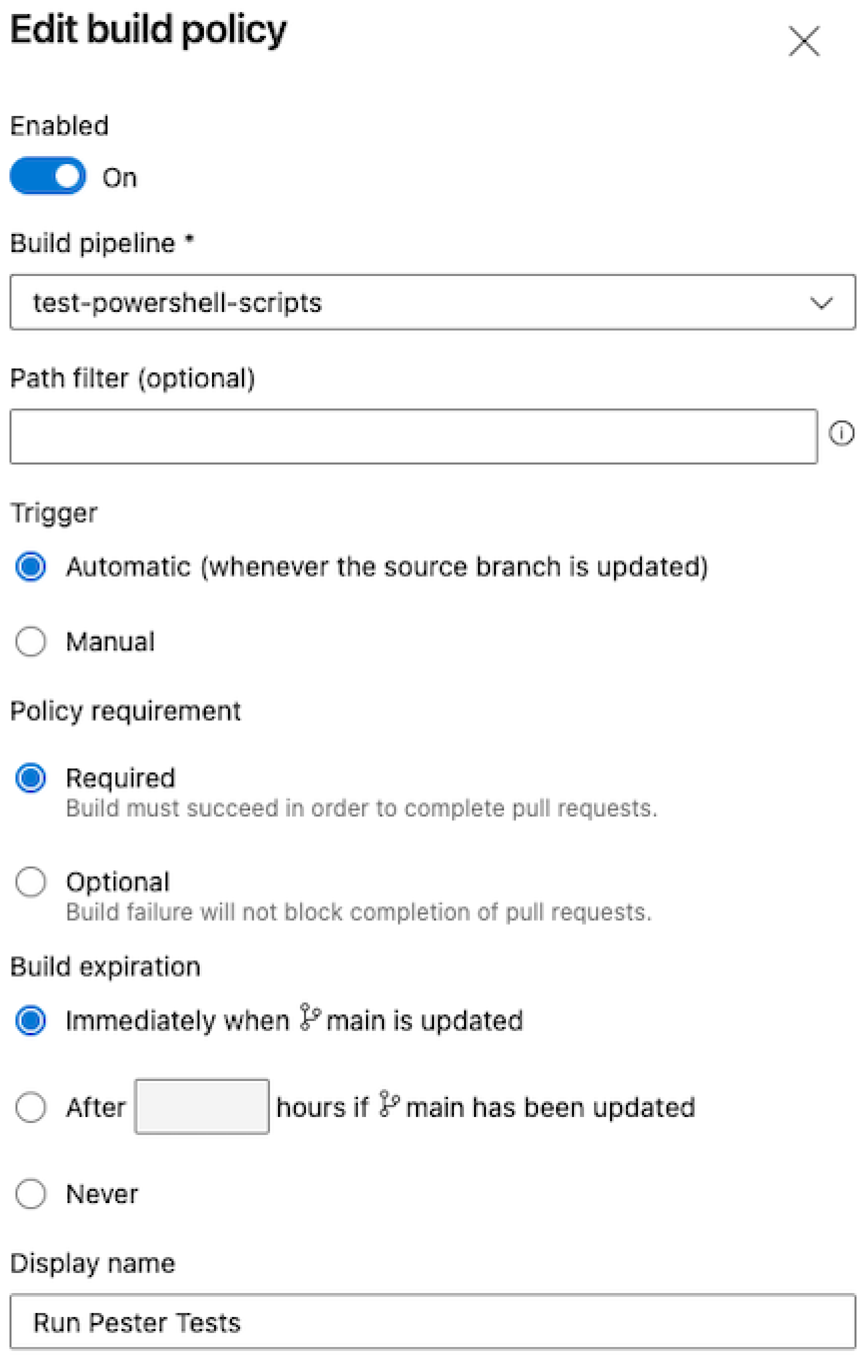
5.4.8.1.2 Add as a Git Branch Policy

To configure this in Azure DevOps, navigate to: **Azure DevOps Repo** → *Repo Name* → **Branches** → **main** → **More options** *(on right end of row)* → **Branch Policies**.

Under **Build Validation**:

- Click the **Plus (+) Sign** to get to the **Add build policy** screen and set the following options.
  - **Trigger**: Automatic
  - **Policy requirement**: Required
  - **Build Expiration**: Immediately
  - **Display Name**: Run Pester Tests
- Click **Save** at the bottom of the screen.

[29] https://github.com/pester/AzureDevOpsExtension



Edit build policy

Enabled

On

Build pipeline *

test-powershell-scripts

Path filter (optional)

Trigger

● Automatic (whenever the source branch is updated)

○ Manual

Policy requirement

● Required
Build must succeed in order to complete pull requests.

○ Optional
Build failure will not block completion of pull requests.

Build expiration

● Immediately when main is updated

○ After hours if main has been updated

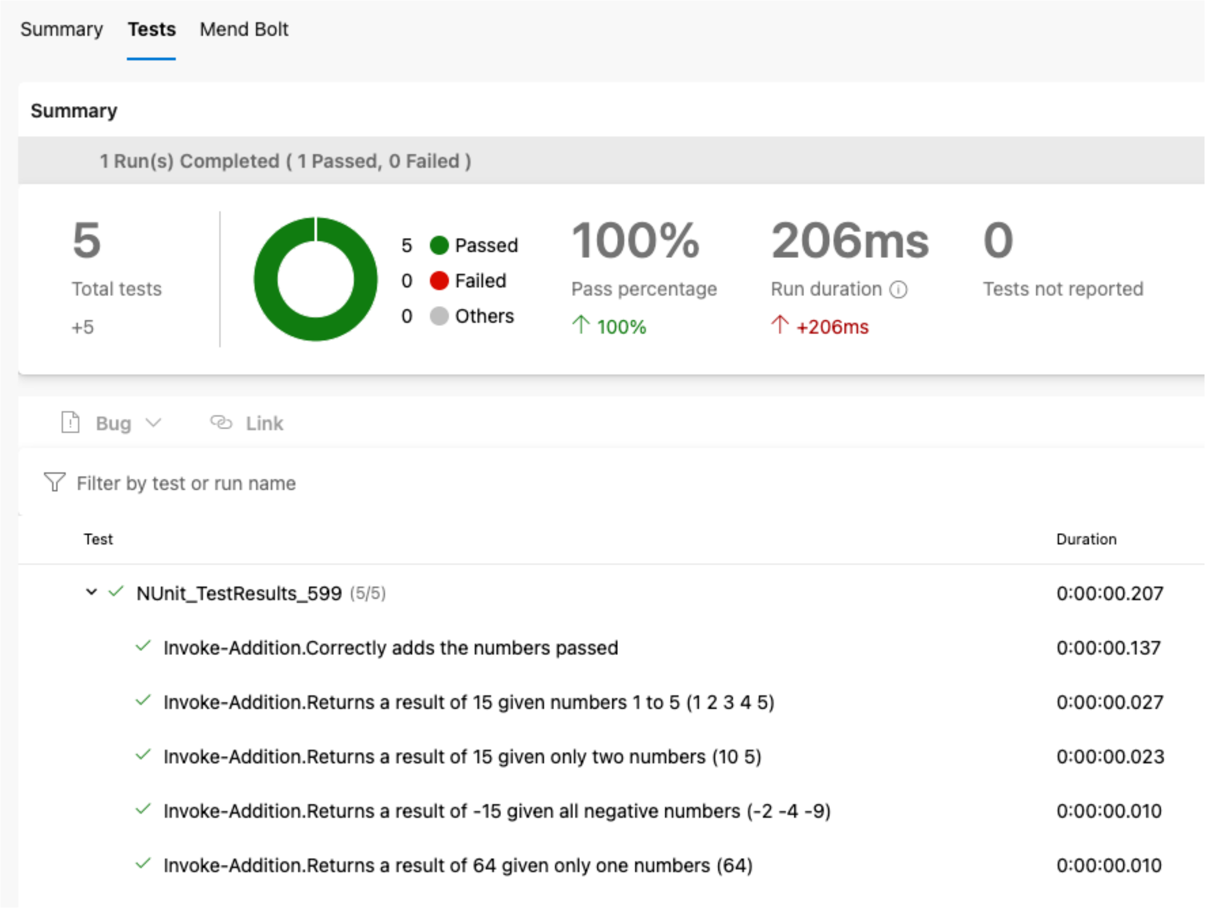
○ Never

Display name

Run Pester Tests

Azure DevOps Git Branch Build Policy Configuration

When there's a new pull request in the repo, it'll automatically run Pester tests and store the results in the **Tests** tab of the build summary screen. The pull request can't merge unless all tests pass.



Summary

1 Run(s) Completed (1 Passed, 0 Failed)

5 Total tests +5

5 ● Passed
0 ● Failed
0 ● Others

100% Pass percentage ↑ 100%

206ms Run duration ↑ +206ms

0 Tests not reported

Test	Duration
✓ NUnit_TestResults_599 (5/5)	0:00:00.207
✓ Invoke-Addition.Correctly adds the numbers passed	0:00:00.137
✓ Invoke-Addition.Returns a result of 15 given numbers 1 to 5 (1 2 3 4 5)	0:00:00.027
✓ Invoke-Addition.Returns a result of 15 given only two numbers (10 5)	0:00:00.023
✓ Invoke-Addition.Returns a result of -15 given all negative numbers (-2 -4 -9)	0:00:00.010
✓ Invoke-Addition.Returns a result of 64 given only one numbers (64)	0:00:00.010

Azure DevOps Test Summary Screen

5.4.8.2 GitHub

GitHub uses GitHub Actions for automation,[30] which stores configuration in YAML files in the `.github/workflows` folder in a Git repository. GitHub Actions is more of an automation platform, meaning the list of event triggers is extensive. CI/CD events are a small subsection of what's available.

You can place the following YAML script in the `workflows` folder of a Git repo to add a GitHub Action that runs Pester tests.

[30] GitHub. (2022, Aug. 25). *Understanding GitHub Actions*. GitHub Docs. [Online]. Available: https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions. [Accessed: Sep. 11, 2022].

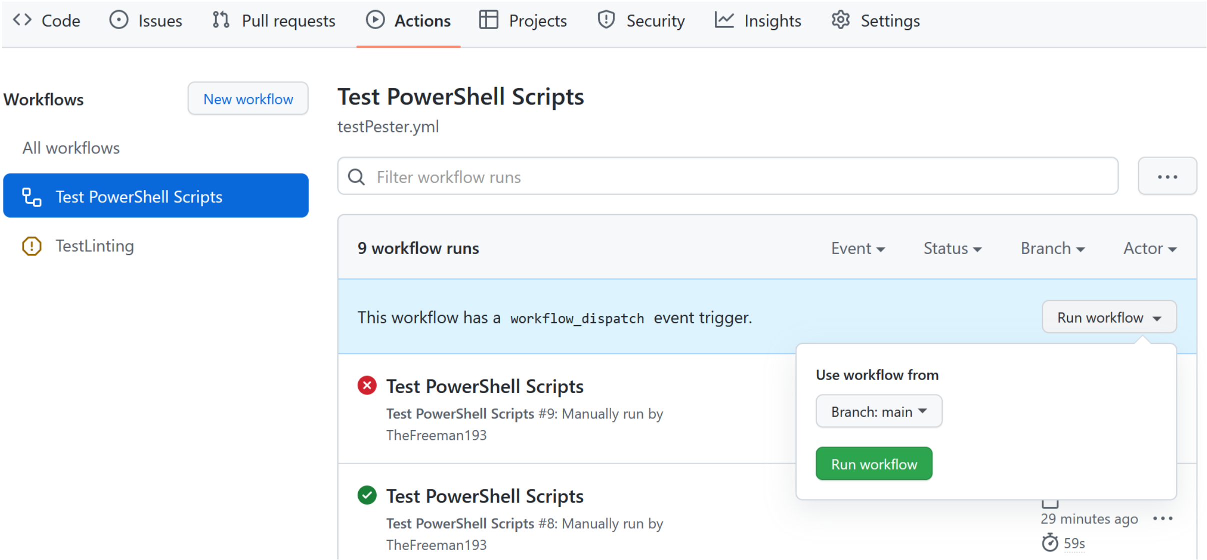
Example 13: YAML definition of a GitHub Actions workflow

```
name: Test PowerShell Scripts
on:
  workflow_dispatch:
  pull_request:
jobs:
  run-pester-tests:
    name: Run Pester tests
    runs-on: ubuntu-latest
    steps:
      - name: Check out code
        uses: actions/checkout@v3
      - name: Perform all Pester Unit Tests from the Scripts folder
        shell: pwsh
        run: |
          $Config = New-PesterConfiguration
          $Config.Run.Path = 'scripts/*.Tests.ps1'
          $Config.Filter.Tag = 'Unit'
          $Config.TestResult.OutputPath = 'Test-Pester.xml'
          $Config.TestResult.Enabled = $true
          $Config.Run.PassThru = $true
          $Result = Invoke-Pester -Configuration $Config
          if ($null -eq $Result) { exit 1 }
          '::notice title=Results::',
          "Total: $($Result.TotalCount), ",
          "Passed: $($Result.PassedCount), ",
          "Failed: $($Result.FailedCount), ",
          "Skipped: $($Result.SkippedCount)" -join ''
      - name: Publish Test Results
        uses: actions/upload-artifact@v3
        if: always()
        with:
          name: test-results
          path: "**/Test-Pester.xml"
```

There are two triggers for this action. The action runs when a pull request is created or updated, and you can start it manually via the GitHub web interface or API. It contains a PowerShell step that calls the `Invoke-Pester` command and displays the result. A second step exports the test result file.

5.4.8.2.1 Run the Action Manually

- Visit the repository webpage and select the **Actions** tab.
- In the left pane, under **All workflows**, click the action you want to run. In this example, the action is labeled **Test PowerShell Scripts**.
- The page now displays a list of historical runs for the action. Click the **Run workflow** button to the top-right of this list.
- In the menu that pops up, select any settings the action requires, including the Git branch that it should run on.
- Click the **Run workflow** button to start the action.



This workflow has a `workflow_dispatch` event trigger.

Test PowerShell Scripts
Test PowerShell Scripts #9: Manually run by TheFreeman193

Test PowerShell Scripts
Test PowerShell Scripts #8: Manually run by TheFreeman193
29 minutes ago
59s

Running a GitHub Action Manually

GitHub doesn't have native support for viewing test results, but you can publish the test file as an artifact.[31] You can then use a third-party or custom action to process the *NUnit XML* test results file. You can also use the `-PassThru` parameter of `Invoke-Pester` to display the results as an annotation in the workflow run log using the `::notice` text command in the standard output,[32] Example 13 uses this approach to display the test results.

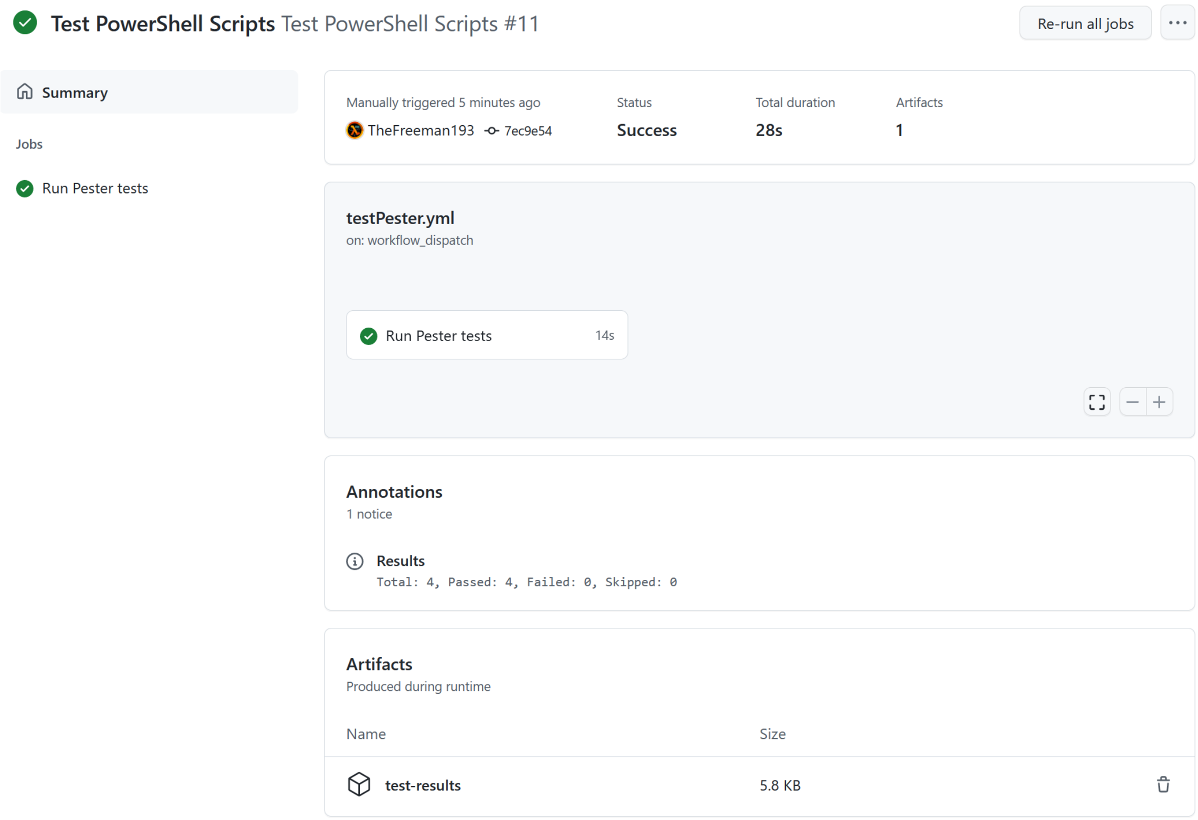
5.4.8.2.2 View the Test Results File

- On the page for your action, click the workflow run you want to view the job summary from. Each run is given a unique number.
- On the job summary page, scroll to the **Artifacts** list and click the artifact you want to download. In this example, the name is **test-results**.

The downloaded file is a compressed archive containing the `Test-Pester.xml` file generated by Pester.

[31] GitHub. (2022, Jun. 06). *Storing workflow data as artifacts*. GitHub Docs. [Online]. Available: https://docs.github.com/en/actions/using-workflows/storing-workflow-data-as-artifacts. [Accessed: Sep. 12, 2022].

[32] GitHub. (2022, Sep. 02). *Workflow commands for GitHub Actions*. GitHub Docs. [Online]. Available: https://docs.github.com/en/actions/using-workflows/workflow-commands-for-github-actions. [Accessed: Sep. 12, 2022].



Test PowerShell Scripts Test PowerShell Scripts #11

Manually triggered 5 minutes ago	Status	Total duration	Artifacts
TheFreeman193 -o- 7ec9e54	Success	28s	1

testPester.yml
on: workflow_dispatch

Run Pester tests 14s

Annotations
1 notice

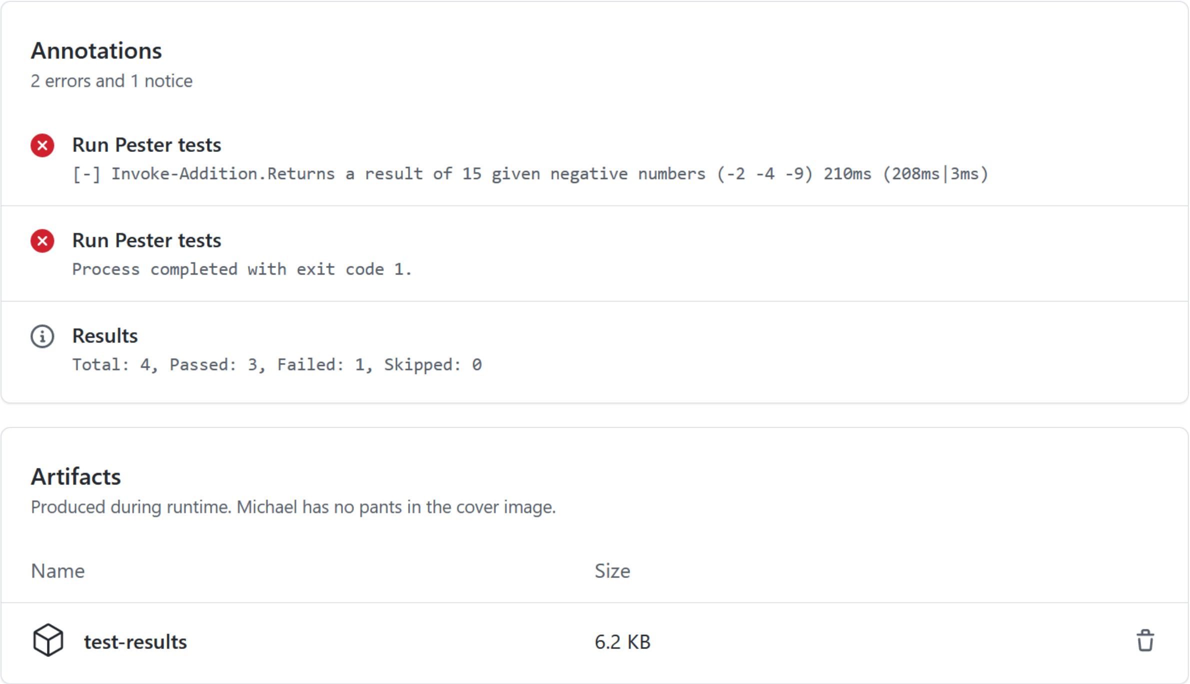
Results
```
Total: 4, Passed: 4, Failed: 0, Skipped: 0
```

Artifacts
Produced during runtime

Name	Size
test-results	5.8 KB

Viewing Test Results as Run Artifacts

The `if:` conditional with `always()` in the second step in Example 13 causes the step to run regardless of any earlier ones failing. This means the test results are always published, even on test failure. The errors that Pester writes to the error stream also show up as error annotations.



Annotations

2 errors and 1 notice

Run Pester tests

```
[-] Invoke-Addition.Returns a result of 15 given negative numbers (-2 -4 -9) 210ms (208ms|3ms)
```

Run Pester tests

```
Process completed with exit code 1.
```

Results

```
Total: 4, Passed: 3, Failed: 1, Skipped: 0
```

Artifacts

Produced during runtime. Michael has no pants in the cover image.

Name	Size
test-results	6.2 KB

Run Failures Still Produce Artifacts with always()

That’s it. GitHub requires nothing else to create the action or link it to a repo’s build process. From this point of view, it’s easier to create automated workflows on GitHub than on Azure DevOps.

5.4.8.2.3 GitHub Job Summaries

Recently, GitHub introduced a new feature called Job Summaries[33].[34]

With this feature, you can create a custom Markdown document and write it to a temporary file defined in the environment variable `GITHUB_STEP_SUMMARY`. Markdown supports rich text and numerous display elements like tables. The following example produces two tables. The first contains the total counts, and the second shows each test result in detail.

Example 14: Using the job summaries feature in a GitHub Actions workflow step

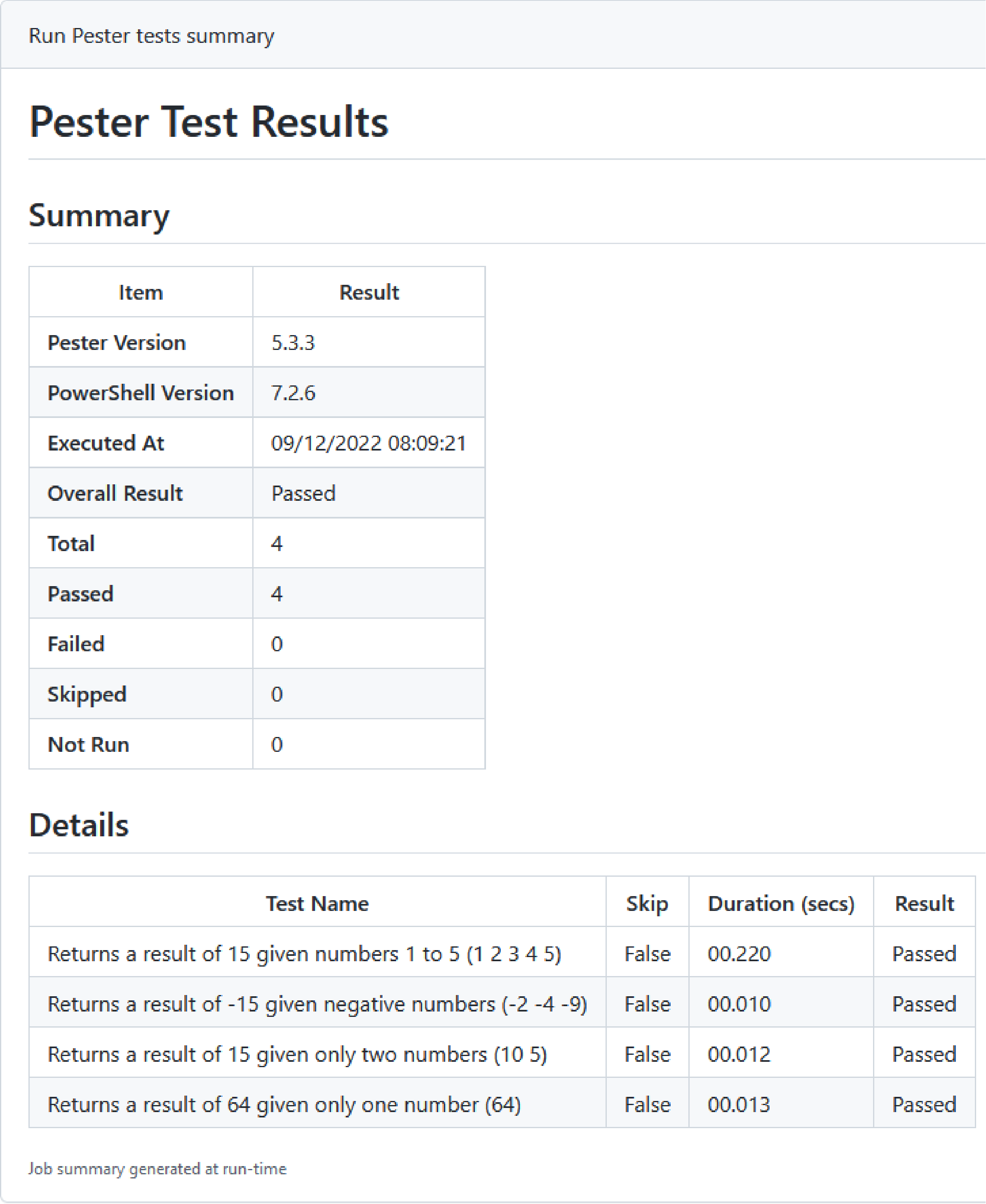
```yaml
- name: Perform all Pester Unit Tests from the Scripts folder
  shell: pwsh
  run: |
    $Config = New-PesterConfiguration
    $Config.Run.Path = 'scripts/*.Tests.ps1'
    $Config.Filter.Tag = 'Unit'
    $Config.Output.Verbosity = 'Detailed'
    $Config.Run.PassThru = $true
    $Result = Invoke-Pester -Configuration $Config
    if ($null -eq $Result) { exit 1 }
    $sb = [System.Text.StringBuilder]::new()
    $sb.AppendLine('# Pester Test results')
    $sb.AppendLine('## Summary')
    $sb.AppendLine('|Item|Result|')
```

[33] https://docs.github.com/en/actions/using-workflows/workflow-commands-for-github-actions#adding-a-job-summary

[34] GitHub. (2022, May. 09). *Supercharging GitHub Actions with Job Summaries*. GitHub Docs. [Online]. Available: https://github.blog/2022-05-09-supercharging-github-actions-with-job-summaries/. [Accessed: Sep. 12, 2022].

```
    $sb.AppendLine('|----|----|')
    $sb.AppendLine("|**Pester Version**|$($Result.Version)|")
    $sb.AppendLine("|**PowerShell Version**|$($Result.PSVersion)|")
    $sb.AppendLine("|**Executed At**|$($Result.ExecutedAt)|")
    $sb.AppendLine("|**Overall Result**|$($Result.Result)|")
    $sb.AppendLine("|**Total**|$($Result.TotalCount)|")
    $sb.AppendLine("|**Passed**|$($Result.PassedCount)|")
    $sb.AppendLine("|**Failed**|$($Result.FailedCount)|")
    $sb.AppendLine("|**Skipped**|$($Result.SkippedCount)|")
    $sb.AppendLine("|**Not Run**|$($Result.NotRunCount)|")
    $sb.AppendLine()
    $sb.AppendLine('## Details')
    $sb.AppendLine('|Test Name|Skip|Duration (secs)|Result|')
    $sb.AppendLine('----|----|----|----|')
    $Result.Tests | ForEach-Object {
      $sb.AppendLine(
        "|$($_.ExpandedName)|$($_.Skip)|" +
        "$($_.Duration.ToString('ss\.fff'))|$($_.Result)|"
      )
    }
    $sb.ToString() | Out-File -Path $ENV:GITHUB_STEP_SUMMARY
```

This results in the following output on the Job Summary screen.



Run Pester tests summary

Pester Test Results

Summary

Item	Result
Pester Version	5.3.3
PowerShell Version	7.2.6
Executed At	09/12/2022 08:09:21
Overall Result	Passed
Total	4
Passed	4
Failed	0
Skipped	0
Not Run	0

Details

Test Name	Skip	Duration (secs)	Result
Returns a result of 15 given numbers 1 to 5 (1 2 3 4 5)	False	00.220	Passed
Returns a result of -15 given negative numbers (-2 -4 -9)	False	00.010	Passed
Returns a result of 15 given only two numbers (10 5)	False	00.012	Passed
Returns a result of 64 given only one number (64)	False	00.013	Passed

Job summary generated at run-time

Sample GitHub Job Summary output

The output in this example is relatively simple, but with the advanced formatting features in Markdown and with Unicode support, you can create comprehensive reports this way.

This completes the comparison between Azure DevOps and GitHub for automating how Pester tests can be run as part of the Pull Request approval process.

5.5 Conclusion

In this chapter, you've seen why Pester is the recommended tool of choice for creating tests for PowerShell scripts and modules. By exploring what Pester offers, you can create complex and flexible test code quickly. You can execute tests locally either via the command line or from within popular text editors. You've also learned how to automate build pipelines with Pester for Azure DevOps and GitHub to ensure high code quality.

5.6 Further Reading

- Pester Quick Start—Pester Docs[35]
- Pester Configuration—Pester Docs[36]
- GitHub Actions Documentation—GitHub Docs[37]
- Azure DevOps YAML Pipeline Reference—Microsoft Docs[38]
- What is DevOps—GitHub Resources[39]
- Azure DevOps—Microsoft Azure[40]
- Difference Between Black Box and White Box Testing—Guru99[41]
- GitHub Actions Feature Overview—GitHub[42]
- Install Visual Studio Code[43]
- Install VS Code PowerShell Plugin[44]

[35] https://pester.dev/docs/quick-start
[36] https://pester.dev/docs/usage/configuration
[37] https://docs.github.com/en/actions
[38] https://learn.microsoft.com/en-us/azure/devops/pipelines/yaml-schema/?view=azure-pipelines
[39] https://resources.github.com/devops/
[40] https://azure.microsoft.com/products/devops/
[41] https://www.guru99.com/back-box-vs-white-box-testing.html
[42] https://github.com/features/actions
[43] https://code.visualstudio.com/
[44] https://marketplace.visualstudio.com/items?itemName=ms-vscode.PowerShell

6. Parameterized Testing

This chapter describes Parameterized Pester Tests to help you on your PowerShell journey. In PowerShell, you can use parameters or `param` in script blocks, functions, and scripts. Parameters define, transfer, limit, or validate inputs for your script. You can learn more about advanced function parameters[1] and cmdlet parameters[2] at Microsoft Docs. A parameterized test is a test that accepts external data as input. Use parameterized tests to avoid rewriting similar tests. If the only difference between each of your tests is its inputs, using a parameterized test reduces code length, improves readability, increases code coverage, and allows for faster changes.

The Pester module has great documentation and everything in this chapter has been inspired by the original Data driven tests[3] article. If you aren't familiar with the module, there is an excellent quick start guide[4].

6.1 Pester Versions and Parameterized Tests

There are significant differences in how parameterized tests are written in Pester v3/v4 and Pester v5.[5] It is important to know which version of Pester is executing your tests. This chapter primarily focuses on parameterized tests in Pester 5.1.0 and later with a brief section on parameterized tests in Pester v4.

We recommend you use the latest version of Pester when possible, and only use earlier versions when your situation requires them. For Windows 10 and Server 2016 systems, PowerShell 5.1 ships with Pester version 3.4.0.[6] You can use `Install-Module` to install more recent versions of Pester from the PowerShell Gallery.

Use this command to check your Pester version:

```
$(Get-Command Invoke-Pester).Version
```

Use `Install-Module` with the `-Force` parameter to install the latest version of Pester side-by-side already installed versions:

```
Install-Module -Name Pester -Force -SkipPublisherCheck
```

Use `Import-Module` with the `-MinimumVersion` or `-MaximumVersion` parameters to import specific versions of the Pester module:

[1] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_functions_advanced_parameters

[2] https://learn.microsoft.com/en-us/powershell/scripting/developer/cmdlet/cmdlet-parameters

[3] https://pester.dev/docs/usage/data-driven-tests

[4] https://pester.dev/docs/quick-start

[5] Pester Team. (2021, May. 14). *Breaking Changes in v5*. Pester Docs. [Online]. Available: https://pester.dev/docs/migrations/breaking-changes-in-v5. [Accessed: Apr. 26, 2022].

[6] Pester Team. (2021, May. 15). *Installation and Update - Compatibility*. Pester Docs. [Online]. Available: https://pester.dev/docs/introduction/installation#compatibility. [Accessed: Apr. 26, 2022].

```
Import-Module -Name Pester -MaximumVersion 4.99.99
```

```
Import-Module -Name Pester -MinimumVersion 5.1.1
```

The Pester documentation also has a guide on how to uninstall the built-in version[7] of Pester.

6.2 Your First Test

You need a file to test. Create a folder named `PowerShellDate` and change your working directory to that folder:

```
New-Item 'PowerShellDate' -Type Directory
Set-Location .\PowerShellDate\
```

Create a file named `Get-PowerShellDate.ps1` and add the following function to the file:

Example 1: A simple function that returns the release dates of various PowerShell versions

```
function Get-PowerShellDate {
    $VersionList = @{
        'Windows PowerShell 1.0' = 'Nov 2006'
        'Windows PowerShell 2.0' = 'Jul 2009'
        'Windows PowerShell 3.0' = 'Oct 2012'
        'Windows PowerShell 4.0' = 'Oct 2013'
        'Windows PowerShell 5.0' = 'Feb 2016'
        'Windows PowerShell 5.1' = 'Aug 2016'
        'PowerShell Core 6.0'  = 'Jan 2018'
        'PowerShell Core 6.1'  = 'Sep 2018'
        'PowerShell Core 6.2'  = 'Mar 2019'
        'PowerShell Core 7.0'  = 'Mar 2020'
        'PowerShell Core 7.1'  = 'Nov 2020'
    }
    return $VersionList[$args]
}
```

This function contains a hash table[8] that will return the release date of each PowerShell version. The automatic variable[9] `$args` is an array of undeclared positional parameters passed to the function.

Example 2: Retrieving the release date of Windows PowerShell 1.0

```
Get-PowerShellDate 'Windows PowerShell 1.0'
```

[7] https://pester.dev/docs/introduction/installation#removing-the-built-in-version-of-pester
[8] https://learn.microsoft.com/en-us/powershell/scripting/learn/deep-dives/everything-about-hashtable
[9] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_automatic_variables#args

```
Nov 2006
```

Example 3: Retrieving the release date of multiple versions

```
Get-PowerShellDate 'Windows PowerShell 1.0' 'Windows PowerShell 2.0'
```

```
Nov 2006
Jul 2009
```

To test your new function, create a file named `Get-PowerShellDate.Tests.ps1` file and add the following code to the file:

Example 4: A Pester test file for Get-PowerShellDate with 1 test

```
BeforeAll {
    . $PSCommandPath.Replace('.Tests.ps1', '.ps1')
}
Describe 'Get-PowerShellDate' {
    It 'Returns Nov 2006' {
        Get-PowerShellDate 'Windows PowerShell 1.0' | Should -Be 'Nov 2006'
    }
}
```

This is the Pester test. The first part is a `BeforeAll` block where you set prerequisites for your tests.[10] In this case, it dot sources the *ps1* file containing your function under test. The second part is a `Describe` block that contains one `It` block. The third part is an `It` block where your test is run. The part of the test to the left of the pipeline is your *action*, and the part of the test to the right of the pipeline is your *assertion*. The result of your action is compared to your assertion, and the test passes if they match. In this case, `Get-PowerShellDate 'Windows PowerShell 1.0'` is expected to return `Nov 2006`. Because `Nov 2006` matches what your assertion expects, the test should pass.

From the `PowerShellDate` folder, run the test with `Invoke-Pester`:

Example 5: Running the Pester test for Get-PowerShellDate

```
Invoke-Pester -Path .\Get-PowerShellDate.Tests.ps1
```

[10] Pester Team. (2019, Feb. 05). *BeforeAll (v4)*. Pester Docs. [Online]. Available: https://pester.dev/docs/v4/commands/BeforeAll. [Accessed: Apr. 26, 2022].

```
Starting discovery in 1 files.
Discovery finished in 39ms.
[+] D:\PowershellDate\Get-PowerShellDate.Tests.ps1 355ms (16ms|306ms)
Tests completed in 368ms
Tests Passed: 1, Failed: 0, Skipped: 0 NotRun: 0
```

Add the parameter `-Output` with the value `Detailed`, and rerun the test to see more detailed output::

Example 6: Displaying detailed Pester test output

```
Invoke-Pester -Path .\Get-PowerShellDate.Tests.ps1 -Output Detailed
```

```
Starting discovery in 1 files.
Discovering in D:\PowerShellDate\Get-PowerShellDate.Tests.ps1.
Found 1 tests. 121ms
Discovery finished in 139ms.

Running tests from 'D:\PowerShellDate\Get-PowerShellDate.Tests.ps1'
Describing Get-PowerShellDate
  [+] Returns Nov 2006 20ms (11ms|9ms)
Tests completed in 547ms
Tests Passed: 1, Failed: 0, Skipped: 0 NotRun: 0
```

The detailed output includes the result, name, and execution time of each of your `It` blocks.

6.2.1 -ForEach

With the `-ForEach` parameter (an alias of `-TestCases`) you are able to decouple your test data from your test actions and assertions.[11] Your actions and assertions can be written once, and reused for each set of data you need to test.

To demonstrate how this improves the readability and maintainability of your tests, first add an `It` block for each key/value pair in your function's hash table. Your `Get-PowerShellDate.Tests.ps1` file should look like this:

Example 7: A set of comprehensive tests for Get-PowerShellDate

```
BeforeAll {
    . $PSCommandPath.Replace('.Tests.ps1', '.ps1')
}
Describe 'Get-PowerShellDate' {
    It 'Returns Nov 2006' {
        Get-PowerShellDate 'Windows PowerShell 1.0' | Should -Be 'Nov 2006'
    }
    It 'Returns Jul 2009' {
        Get-PowerShellDate 'Windows PowerShell 2.0' | Should -Be 'Jul 2009'
    }
    It 'Returns Oct 2012' {
        Get-PowerShellDate 'Windows PowerShell 3.0' | Should -Be 'Oct 2012'
    }
```

[11] Pester Team. (2019, Jan. 09). *It - TestCases*. Pester Docs. [Online]. Available: https://pester.dev/docs/commands/It#-testcases. [Accessed: Apr. 26, 2022].

```
    It 'Returns Oct 2013' {
        Get-PowerShellDate 'Windows PowerShell 4.0' | Should -Be 'Oct 2013'
    }
    It 'Returns Feb 2016' {
        Get-PowerShellDate 'Windows PowerShell 5.0' | Should -Be 'Feb 2016'
    }
    It 'Returns Aug 2016' {
        Get-PowerShellDate 'Windows PowerShell 5.1' | Should -Be 'Aug 2016'
    }
    It 'Returns Jan 2018' {
        Get-PowerShellDate 'PowerShell Core 6.0' | Should -Be 'Jan 2018'
    }
    It 'Returns Sep 2018' {
        Get-PowerShellDate 'PowerShell Core 6.1' | Should -Be 'Sep 2018'
    }
    It 'Returns Mar 2019' {
        Get-PowerShellDate 'PowerShell Core 6.2' | Should -Be 'Mar 2019'
    }
    It 'Returns Mar 2020' {
        Get-PowerShellDate 'PowerShell Core 7.0' | Should -Be 'Mar 2020'
    }
    It 'Returns Nov 2020' {
        Get-PowerShellDate 'PowerShell Core 7.1' | Should -Be 'Nov 2020'
    }
}
```

You now have an action for all valid input being tested against an assertion for the action's expected output. However, your data are tightly coupled with your actions and assertions. If you want to change your action or assertion, each `It` block will have to be updated with your change.

Update your tests to use a single `It` block with the `-ForEach` parameter and hash tables containing your data:

Example 8: An equivalent set of tests using parameters

```
BeforeAll {
    . $PSCommandPath.Replace('.Tests.ps1', '.ps1')
}
Describe 'Get-PowerShellDate' {
    It 'Returns <Date> for <Name>' -ForEach @(
        @{Name = 'Windows PowerShell 1.0'; Date = 'Nov 2006'}
        @{Name = 'Windows PowerShell 2.0'; Date = 'Jul 2009'}
        @{Name = 'Windows PowerShell 3.0'; Date = 'Oct 2012'}
        @{Name = 'Windows PowerShell 4.0'; Date = 'Oct 2013'}
        @{Name = 'Windows PowerShell 5.0'; Date = 'Feb 2016'}
        @{Name = 'Windows PowerShell 5.1'; Date = 'Aug 2016'}
        @{Name = 'PowerShell Core 6.0';    Date = 'Jan 2018'}
        @{Name = 'PowerShell Core 6.1';    Date = 'Sep 2018'}
        @{Name = 'PowerShell Core 6.2';    Date = 'Mar 2019'}
        @{Name = 'PowerShell Core 7.0';    Date = 'Mar 2020'}
        @{Name = 'PowerShell Core 7.1';    Date = 'Nov 2020'}
    ) {
        Get-PowerShellDate $Name | Should -Be $Date
    }
}
```

The `-ForEach` parameter accepts an array of hash tables[12]. The hash tables contain a key/value pair for each variable in your test. In this case, the `$Name` and `$Date` variables in the test will be

[12] https://devblogs.microsoft.com/scripting/combine-arrays-and-hash-tables-in-powershell-for-fun-and-profit/

populated with values of the ‘Name’ and ‘Date’ keys in the hash tables. The `<Name>` and `<Date>` templates in the `It` block’s description will also be populated with these values.

Run your tests again with detailed output:

Example 9: Running the parameterized tests

```
Invoke-Pester -Path .\Get-PowerShellDate.Tests.ps1 -Output Detailed
```

```
Starting discovery in 1 files.
Discovering in D:\PowerShellDate\Get-PowerShellDate.Tests.ps1.
Found 11 tests. 129ms
Discovery finished in 164ms.

Running tests from 'D:\PowerShellDate\Get-PowerShellDate.Tests.ps1'
Describing Get-PowerShellDate
  [+] Returns Nov 2006 for Windows PowerShell 1.0 122ms (94ms|27ms)
  [+] Returns Jul 2009 for Windows PowerShell 2.0 13ms (8ms|4ms)
  [+] Returns Oct 2012 for Windows PowerShell 3.0 14ms (10ms|4ms)
  [+] Returns Oct 2013 for Windows PowerShell 4.0 8ms (5ms|3ms)
  [+] Returns Feb 2016 for Windows PowerShell 5.0 12ms (8ms|4ms)
  [+] Returns Aug 2016 for Windows PowerShell 5.1 12ms (8ms|5ms)
  [+] Returns Jan 2018 for PowerShell Core 6.0 13ms (8ms|4ms)
  [+] Returns Sep 2018 for PowerShell Core 6.1 49ms (45ms|4ms)
  [+] Returns Mar 2019 for PowerShell Core 6.2 13ms (8ms|5ms)
  [+] Returns Mar 2020 for PowerShell Core 7.0 9ms (6ms|3ms)
  [+] Returns Nov 2020 for PowerShell Core 7.1 9ms (7ms|3ms)
Tests completed in 904ms
Tests Passed: 11, Failed: 0, Skipped: 0 NotRun: 0
```

Because `-ForEach` and its array of hash tables is in your `It` block, a separate test is executed for each hash table.

6.2.2 Templates ‘<>’

The values surrounded by angle brackets `<>` in your `It` block descriptions are called templates[13]. Templates apply variable values inside the `It`, `Describe`, and `Context` block descriptions. An example with an array:

Example 10: Using description templates with parameterized tests

```
Describe 'Building' {
    It '<_> should not be null or empty' -ForEach @(
        'Post', 'B2F1', 'B2F2', 'Contoso', 'Fabrikam'
    ) {
        $_ | Should -Not -BeNullorempty
    }
}
```

The `_` inside `<_>` is equivalent to `$_` or `$PSItem`.[14] It takes each value located in the array `@('Post','B2F1','B2F2','Contoso','Fabrikam')` and passes it to the respective `It` block.

Another example using templates in `Describe` and `It` blocks:

[13] https://pester.dev/docs/usage/data-driven-tests#using--templates

[14] Microsoft. (2022, Jul. 07). *About Automatic Variables (Microsoft.PowerShell.Core) - $_*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_automatic_variables#_. [Accessed: Aug. 15, 2022].

Example 11: Using description templates with multiple block types

```
BeforeAll {
    . $PSCommandPath.Replace('.Tests.ps1', '.ps1')
    $title = 'function PowerShellDate '
}
Describe '<title>' {
    It 'Returns <Date> for <Name>' -ForEach @(
        @{Name = 'Windows PowerShell 1.0'; Date = 'Nov 2006'}
    ) {
        Get-PowerShellDate $Name | Should -Be $Date
    }
}
```

```
Invoke-Pester '.\Get-PowerShellDate.Tests.ps1' -Output Detailed
```

```
Starting discovery in 1 files.
Discovering in D:\PowerShellDate\Get-PowerShellDate.Tests.ps1.
Found 1 tests. 105ms
Discovery finished in 122ms.

Running tests from 'D:\PowerShellDate\Get-PowerShellDate.Tests.ps1'
Describing function PowerShellDate
  [+] Returns Nov 2006 for Windows PowerShell 1.0 19ms (10ms|9ms)
Tests completed in 521ms
Tests Passed: 1, Failed: 0, Skipped: 0 NotRun: 0
```

The `title` variable is now showing its value and also the variables `Date` and `Name`. This is helpful when reading detailed results. In Pester version v5, templates support dot navigation[15]. You can use dot navigation to retrieve values from nested objects.

Example 12: Using dot navigation within description templates

```
Describe 'Building' {
    It '<building1> does not host <building2.floor3.Apt1>' -ForEach @(
        @{
            Building1 = 'Post'
            Building2 = @{
                Floor1 = 'B2F1'
                Floor2 = 'B2F2'
                Floor3 = @{
                    Apt1 = 'Contoso'
                    Apt2 = 'Fabrikam'
                }
            }
        }
    ) {
        $PSitem | Should -Not -BeNullOrEmpty
    }
}
```

Running the test returns:

[15] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_properties#properties-of-scalar-objects-and-collections

```
...

Describing Building
  [+] Post does not host Contoso 19ms (10ms|9ms)
Tests completed in 482ms
Tests Passed: 1, Failed: 0, Skipped: 0 NotRun: 0
```

The object defined as a hashtable after the `-ForEach` displays the third nested level correctly: this third value, 'Contoso', can be seen in the detailed output.

6.2.3 BeforeDiscovery

There are two execution phases[16] in Pester v5. These two phases are named *discovery* and *run*[17]. Place code required to setup your test code inside the `BeforeDiscovery` block.[18] This code is executed during the *discovery* phase of Pester execution. Your test code is executed during the *run* phase of Pester execution. The results from code executed during the discovery phase are available to your tests during the run phase.

Move the array of hash tables from the `-Foreach` parameter in your `It` block to the `BeforeDiscovery` block:

Example 13: Using the BeforeDiscovery section to initialize variables that must be available in both the discovery and test phases

```
BeforeAll {
    . $PSCommandPath.Replace('.Tests.ps1', '.ps1')
}
BeforeDiscovery {
    $title = 'function PowerShellDate '
    $HList = [System.Collections.Generic.List[Hashtable]]::new()
    $HList.Add(@{Name = 'Windows PowerShell 1.0'; Date = 'Nov 2006'})
    $HList.Add(@{Name = 'Windows PowerShell 2.0'; Date = 'Jul 2009'})
    $HList.Add(@{Name = 'Windows PowerShell 3.0'; Date = 'Oct 2012'})
    $HList.Add(@{Name = 'Windows PowerShell 4.0'; Date = 'Oct 2013'})
    $HList.Add(@{Name = 'Windows PowerShell 5.0'; Date = 'Feb 2016'})
    $HList.Add(@{Name = 'Windows PowerShell 5.1'; Date = 'Aug 2016'})
    $HList.Add(@{Name = 'PowerShell Core 6.0';    Date = 'Jan 2018'})
    $HList.Add(@{Name = 'PowerShell Core 6.1';    Date = 'Sep 2018'})
    $HList.Add(@{Name = 'PowerShell Core 6.2';    Date = 'Mar 2019'})
    $HList.Add(@{Name = 'PowerShell Core 7.0';    Date = 'Mar 2020'})
    $HList.Add(@{Name = 'PowerShell Core 7.1';    Date = 'Nov 2020'})
}
Describe "$title" {
    It 'Returns <Date> for <Name>' -ForEach $HList {
        Get-PowerShellDate $Name | Should -Be $Date
    }
}
```

The variable `title` is set in the `BeforeDiscovery` block and displayed in the `Describe` block. `HList` is a list of hashtables and is also available for all `Describe` and `It` blocks.

[16] https://pester.dev/docs/usage/data-driven-tests#execution-is-not-top-down

[17] https://pester.dev/docs/usage/discovery-and-run

[18] Pester Team. (2021, Sep. 12). *BeforeDiscovery*. Pester Docs. [Online]. Available: https://pester.dev/docs/commands/BeforeDiscovery. [Accessed: Aug. 18, 2022].

6.2.3.1 BeforeAll

BeforeAll[19] is the location where you import the file that contains your function and setup before running the tests.

It's usually this:

```
BeforeAll {
    . $PSCommandPath.Replace('.Tests.ps1', '.ps1')
}
```

Storing functions and tests in separate folders is a common pattern in PowerShell module development.[20] You can add another `Replace()` method to `$PSCommandPath` to handle this scenario:

```
BeforeAll {
    . $PSCommandPath.Replace('\Tests\', '\Src\').Replace('.Tests.ps1', '.ps1')
}
```

This replaces the path `\Tests\Get-PowerShellDate.Tests.ps1` with `\Src\Get-PowerShellDate.ps1`.

6.2.3.2 AfterAll

AfterAll[21] follows the same principle as `BeforeAll`, but it runs *after* the test. You usually perform clean-up activities, such as deleting temporary files, in the `AfterAll` block.

Example 14: Cleaning up temporary test files in the AfterAll block

```
Describe 'PSversion Check' {
    BeforeAll {
        $FullName = '.\psversiontable.xml'
        $PSVersionTable | Export-Clixml $FullName
    }
    It 'File PSversion Should Not Be Null Or Empty'  {
        Get-Content $FullName | Should -Not -BeNullOrEmpty
    }
    AfterAll {
        Remove-Item -Path $FullName -Force
    }
}
```

6.2.3.3 BeforeEach and AfterEach

BeforeEach[22] and AfterEach[23] are similar in function to BeforeAll, but are only for `Context` or `Describe` blocks. This block runs once for every `It` block contained within the current `Context` or `Describe` block.[24]

[19] https://pester.dev/docs/commands/BeforeAll

[20] Pester Team. (2022, Jun. 19). *File placement and naming.* Pester Docs. [Online]. Available: https://pester.dev/docs/usage/file-placement-and-naming. [Accessed: Aug. 18, 2022].

[21] https://pester.dev/docs/commands/AfterAll

[22] https://pester.dev/docs/commands/BeforeEach

[23] https://pester.dev/docs/commands/AfterEach

[24] Pester Team. (2022, Apr. 20). *Setup and teardown - BeforeEach.* Pester Docs. [Online]. Available: https://pester.dev/docs/usage/setup-and-teardown#beforeeach. [Accessed: Apr. 26, 2022].

Example 15: Running code before every It test using BeforeEach

```
Describe 'PSversion Check' {
    BeforeEach {
        $FullName = '.\psversiontable.xml'
        $PSVersionTable | Export-Clixml $FullName
    }
    It 'File PSversion Should Not Be Null Or Empty'  {
        Get-Content $FullName | Should -Not -BeNullOrEmpty
    }
    AfterEach {
        Remove-Item -Path $FullName -Force
    }
}
```

6.2.4 Param

As described earlier, you can use the full `Param` syntax in your Pester test files.[25] The `Param` block will inherit the same validation properties, parameter sets, etc. available in scripts and functions. This allows you to control the type of input and detect potential errors before you run any tests.

Create a new test file named `Get-PSVersionTable.Tests.ps1` that contains the following:

Example 16: Using a param() block in a test script

```
param (
    [Parameter(Mandatory)]
    [ValidateScript({Test-Path $_})]
    [string] $ImportPath
)
BeforeAll {
    $PSVersionTableContent = Import-Clixml $ImportPath
    $PSversion = $($PSVersionTableContent.PSVersion.ToString())
}
Describe 'Current PSVersionTable' {
    It 'Current PSversion should be <PSVersionTableContent.PSVersion>'  {
        $PSVersionTable.PSVersion | Should -Match $PSversion
    }
}
```

The `$ImportPath` parameter is `(Mandatory)`, must be of type `[string]`, and must pass the validation script `{Test-Path $_}`. In the `BeforeAll` block, the `$PSVersionTableContent` variable is populated using `Import-CliXml` and the value of `$ImportPath`. The action on the left of the `It` block gets the `Version` property of the `$PSVersionTable` automatic variable. The assertion on the right of the `It` block compares the action's result with a string from the imported `SemanticVersion` object in `$PSVersionTableContent`.

You can use parameters to reuse the same test code with different actions and assertions. To invoke test files with parameters, you'll have to create a Pester container in version v5, or use the `-Script` parameter of `Invoke-Pester` in v4.

[25] Pester Team. (2022, Apr. 20). *Data driven tests - Providing external data to tests*. Pester Docs. [Online]. Available: https://pester.dev/docs/usage/data-driven-tests#providing-external-data-to-tests. [Accessed: Apr. 26, 2022].

6.2.5 Pester Container

Pester v5 introduced many new concepts, one of them being the Pester Container.[26] These objects are used by `Invoke-Pester` when the test file has a `param` block. The `-Data` parameter accepts a hashtable of parameter names and values for the script's parameters, and `-Path` is the location of the test file. In this case, the parameter name needs to be *ImportPath*. Create a folder named Import and two reference files. First, open PowerShell 7 or later and run:

```
New-Item -ItemType Directory -Name 'Import'
```

```
$PSVersionTable | Export-Clixml .\Import\psversiontable-pwsh.xml
```

Then, open Windows PowerShell 5.1 and run:

```
$PSVersionTable | Export-Clixml .\Import\psversiontable5.1.xml
```

Create the container and invoke it with detailed output:

Example 17: Passing parameters to tests in Pester v5 using containers

```
$ContainerParams = @{
    Path = '.\Tests\Get-PSVersionTable.Tests.ps1'
    Data = @{ ImportPath = '.\Import\PSVersiontable-pwsh.xml' }
}
$container = New-PesterContainer @ContainerParams
Invoke-Pester -Container $container -Output Detailed
```

```
Starting discovery in 1 files.
Discovering in D:\PowerShellDate\Tests\Get-PSVersionTable.Tests.ps1.
Found 1 tests. 129ms
Discovery finished in 145ms.

Running tests from 'D:\PowerShellDate\Tests\Get-PSVersionTable.Tests.ps1'
Describing Current PSVersionTable
  [+] Current PSversion should be 7.1.3 41ms (32ms|9ms)
Tests completed in 621ms
Tests Passed: 1, Failed: 0, Skipped: 0 NotRun: 0
```

Now perform the same tests with the PSVersiontable5.1.xml file:

[26] Pester Team. (2021, Sep. 12). *New-PesterContainer*. Pester Docs. [Online]. Available: https://pester.dev/docs/commands/New-PesterContainer. [Accessed: Apr. 26, 2022].

Example 18: Running the test in PowerShell 7 with the XML file from PowerShell 5.1

```
$ContainerParams = @{
    Path = '.\Tests\Get-PSVersionTable.Tests.ps1'
    Data = @{ ImportPath = '.\Import\PSVersiontable5.1.xml' }
}
$container = New-PesterContainer @ContainerParams
Invoke-Pester -Container $container -Output Detailed
```

```
Starting discovery in 1 files.
Discovering in D:\PowerShellDate\Tests\Get-PSVersionTable.Tests.ps1.
Found 1 tests. 9ms
Discovery finished in 12ms.
Running tests.

Running tests from 'D:\PowerShellDate\Tests\Get-PSVersionTable.Tests.ps1'
Describing Current PSVersionTable
  [-] Current PSversion should be 5.1.19041.1645 12ms (10ms|1ms)
   Expected regular expression '5.1.19041.1645' to match 7.1.3, but it
   did not match.
   at $PSVersionTable.PSVersion | Should -Match $PSversion,
   D:\PowerShellDate\Tests\Get-PSVersionTable.Tests.ps1:12
   at <ScriptBlock>, D:\PowerShellDate\Tests\Get-PSVersionTable.Tests.ps1:12
Tests completed in 255ms
Tests Passed: 0, Failed: 1, Skipped: 0 NotRun: 0
```

This test fails because 5.1 is different from 7.1.3. It doesn't need two *ps1* files to test both versions as all work is done in the `BeforeAll` block. With `New-PesterContainer` you can leverage `-Path` and `-Data` as inputs. Then, once inside the test, use `-ForEach` to generate many tests. This concept is the core of this chapter.

6.2.6 PesterConfiguration

Finally, parameterized tests might generate a lot of output and these need to be handled via `[PesterConfiguration]`[27]. Pester configuration helps you manage all types of scenarios. Here is an example running the tests from examples 17 and 18 with diagnostic verbosity:

Example 19: Running the test with diagnostic verbosity

```
$PesterconfigDiag = [PesterConfiguration]::Default
$PesterconfigDiag.Output.Verbosity = 'Diagnostic'
$PesterconfigDiag.Run.Container = $container
Invoke-Pester -Configuration $PesterconfigDiag
```

There is also the `OutputFormat` option that can handle *NUnitXml*, *NUnit2.5* or *JUnitXml*.

[27] https://pester.dev/docs/usage/Configuration

Example 20: Running the test and outputting to JUnit XML

```
$PesterconfigDiag = [PesterConfiguration]::Default
$PesterconfigDiag.TestResult.OutputFormat = 'JUnitXml'
$PesterconfigDiag.TestResult.Enabled = $true
$PesterconfigDiag.Run.Container = $container
Invoke-Pester -Configuration $PesterconfigDiag

Get-ChildItem -Path *.xml -Name
```

```
Starting discovery in 1 files.
Discovery finished in 9ms.
Running tests.
[+] D:\PowerShellDate\Tests\Get-PSVersionTable.Tests.ps1 204ms (2ms|195ms)
Tests completed in 206ms
Tests Passed: 1, Failed: 0, Skipped: 0 NotRun: 0

testResults.xml
```

Calling New-PesterConfiguration[28] is equivalent to using `[PesterConfiguration]::Default`.[29]

6.3 Pester v4

If you are using Pester versions v3 or v4, you will need to use the `-Script` parameter.[30] It has two properties, `-Path` and `-Parameters`. Here is an example for `Get-PSVersionTable` that works with these Pester versions:

Example 21: Running Get-PSVersionTable Tests with parameters in Pester v4

```
param (
    [Parameter(Mandatory)]
    [ValidateScript({Test-Path $_})]
    [string] $ImportPath
)
$PSVersionTableContent = Import-Clixml $ImportPath
$PSversion = $($PSVersionTableContent.PSVersion.ToString())
Describe 'Current PSVersionTable' {
    It "Current PSversion should be $($PSVersionTableContent.PSVersion)" {
        $PSVersionTable.PSVersion | Should Be $PSversion
    }
}
```

[28] https://pester.dev/docs/commands/New-PesterConfiguration

[29] Pester Team. (2021, Apr. 30). *New-PesterConfiguration*. Pester Docs. [Online]. Available: https://pester.dev/docs/commands/New-PesterConfiguration. [Accessed: Apr. 26, 2022].

[30] Pester Team. (2021, Apr. 17). *Invoke-Pester - Parameters*. Pester Docs. [Online]. Available: https://pester.dev/docs/v4/commands/Invoke-Pester#-script. [Accessed: Apr. 26, 2022].

```
$InvokePesterScript = @{
    Path = 'D:\PowerShellDate\Tests\Get-PSVersionTable.Tests.ps1'
    Parameters = @{
        Importpath = 'D:\PowerShellDate\Tests\Import\PSVersiontable5.1.xml'
    }
}
Invoke-Pester -Script $InvokePesterScript
```

```
Pester v4.10.1
Executing all tests in 'D:\PowerShellDate\Tests\Get-PSVersionTable.Tests.ps1'

Executing script D:\PowerShellDate\Tests\Get-PSVersionTable.Tests.ps1

  Describing Current PSVersionTable
    [+] Current PSversion should be <PSVersionTableContent.PSVersion> 36ms
Tests completed in 215ms
Tests Passed: 1, Failed: 0, Skipped: 0, Pending: 0, Inconclusive: 0
```

You can also use splatting[31] with an advanced configuration:

Example 22: Running the tests with advanced configuration in Pester v4

```
$InvokePesterParams = @{
    Script = @{
        Path = 'D:\PowerShellDate\Tests\Get-PSVersionTable.Tests.ps1'
        Parameters = @{
            Importpath = 'D:\PowerShellDate\Tests\Import\PSVersiontable5.1.xml'
        }
    }
    Outputfile = 'test.xml'
    Outputformat = 'NUnitXml'
    PassThru = $true
}
$ResultPester = Invoke-pester @InvokePesterParams

Test-Path .\test.xml
```

```
Pester v4.10.1
Executing all tests in 'D:\PowerShellDate\Tests\Get-PSVersionTable.Tests.ps1'

Executing script D:\PowerShellDate\Tests\Get-PSVersionTable.Tests.ps1

  Describing Current PSVersionTable
    [+] Current PSversion should be <PSVersionTableContent.PSVersion> 0ms
Tests completed in 111ms
Tests Passed: 1, Failed: 0, Skipped: 0, Pending: 0, Inconclusive: 0

True
```

As you can see, the idea is the same: `-Parameters` provides the input variables and `-Path` selects the test file. You may need to adjust the syntax based on your environment.

[31] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_splatting

6.4 Outputs

With the ability to have many variables as input, your test results will also increase. Here, `Get-Service.Tests.ps1` tests if each service is currently running. Not helpful in real-life scenarios, but practical for this example.

Example 23: Collecting result data with -PassThru

```
BeforeDiscovery {
    $AllServices = Get-Service
    $HList = [System.Collections.Generic.List[hashtable]]::new()
    $AllServices.ForEach{
        $HList.add(@{ Name = $_.name })
    }
}

BeforeAll {
    $AllServices = Get-Service
}

Describe 'Services' {
    It '<Name> Should be Running' -ForEach $HList {
        $AllServices.Where{ $_.name -match $name }.Status |
            Should -Match 'Running'
    }
}
```

```
$PesterParams = @{
    Path     = '.\Tests\Get-Service.Tests.ps1'
    Output   = 'Detailed'
    PassThru = $true
}
$ResultPester = Invoke-Pester @PesterParams
```

`$ResultPester` is a `[Pester.Run]` object and contains one test result for each service on your machine.[32] You could select only the ones that passed and export them to CSV:

```
$ResultPester.Passed | Select ExpandedName, Result | Export-Csv '.\Passed.csv'
```

Or output a detailed failed test report to the host:

```
$ResultPester.Failed |
    Select-Object ExpandedName,Result,ErrorRecord,Block,ExecutedAt,Duration
```

With the ImportExcel[33] module, you can create an Excel report with conditional formatting to highlight your Pass/Fail results:

[32] Pester Team. (2021, Mar. 07). *Pester - Run.cs.* pester/Pester on GitHub. [Online]. Available: https://github.com/pester/Pester/blob/main/src/csharp/Pester/Run.cs. [Accessed: Apr. 26, 2022].

[33] https://github.com/dfinke/ImportExcel

Example 24: Exporting Pester result data to an Excel spreadsheet with ImportExcel

```
$ResultData = $ResultPester.Tests |
    Select-Object ExpandedName, Result, { $_.ErrorRecord },
    Block, ExecutedAt, Duration

$ResultData | Export-Excel -Path .\test.xlsx -ConditionalText $(
    $TextPARAMs = @{
        Text                 = 'Passed'
        Range                = 'B:B'
        BackgroundColor      = 'Green'
        ConditionalTextColor = 'White'
    }
    New-ConditionalText @TextPARAMs
    $TextPARAMs2 = @{
        Text                 = 'Failed'
        Range                = 'B:B'
        BackgroundColor      = 'Red'
        ConditionalTextColor = 'White'
    }
    New-ConditionalText @TextPARAMs2
)
```

6.5 One Last Example

Finally, consider one last example. Create a new file named `Get-FileContent.Tests.ps1`. This test will read the content of a file and check each line for two conditions. The first condition tests if the line length is less than 100 characters. The second condition tests if the line contains the letter ‘o’. The `FullName` property provides a file for the `Get-Content` command.

Example 25: Exporting Pester result data to an Excel spreadsheet with ImportExcel

```
param (
    [Parameter(Mandatory)]
    [string] $FullName
)

BeforeDiscovery {
    $FileContent = Get-Content -Path $FullName
    $HList = [System.Collections.Generic.List[hashtable]]::new()
    $L = 0
    $FileContent.ForEach{
        $L++
        $HList.Add(@{
            LineNumber  = $L
            LineLength  = $PSItem.Length
            LineContent = $PSItem
        })
    }
}

Describe 'FileContent' -ForEach $HList {
    Context 'Line <LineNumber>' {
        It 'Line <LineNumber> Should be less than 100 char' {
            $PSItem.Length | Should -BeLessThan 100
        }
        It "Line '<LineContent>' Should Match 'o'" {
            $PSItem.LineContent | Should -Match 'o'
        }
    }
}
```

Running the following will create an Excel file with the failed/passed results for each line:

```
$ContainerParams = @{
    Path = '.\Tests\Get-FileContent.Tests.ps1'
    Data = @{ FullName = '.\Import\psversiontable-pwsh.xml'}
}
$Container = New-PesterContainer @ContainerParams

$ResultPester = Invoke-Pester -Container $Container -Output None -PassThru

$ResultData = $ResultPester.Tests |
    Select-Object ExpandedName, Result, { $_.ErrorRecord },
    Block, ExecutedAt, Duration

$ResultData | Export-Excel -Path .\test.xlsx -ConditionalText $(
    $TextPARAMs = @{
        Text =  'Passed'
        Range = 'B:B'
        BackgroundColor =  'Green'
        ConditionalTextColor = 'White'
    }
    New-ConditionalText @TextPARAMs
    $TextPARAMs2 = @{
        Text =  'Failed'
        Range = 'B:B'
        BackgroundColor =  'Red'
        ConditionalTextColor = 'White'
    }
    New-ConditionalText @TextPARAMs2
)
```

Modify the test to accept multiple input files. The `Get-FileContent.Tests.ps1` file should look like this:

Example 26: A modified test script that accepts multiple files

```
param (
    [Parameter(Mandatory)]
    [string[]]$FullNameList
)

BeforeDiscovery {
    $HList = [System.Collections.Generic.List[hashtable]]::new()
    foreach ($FullName in $FullNameList) {
        $FileContent = Get-Content -Path $FullName
        $FileName = Split-Path $FullName -Leaf
        $L = 0
        $FileContent.ForEach{
            $L++
            $HList.Add(@{
                FileName    = $FileName
                LineNumber  = $L
                LineLength  = $PSItem.Length
                LineContent = $PSItem
            })
        }
    }
}

Describe 'FileContent' -ForEach $HList {
    Context 'Line <LineNumber> from <FileName>' {
        It 'Line <LineNumber> Should be less than 100 char' {
            $PSItem.Length | Should -BeLessThan 100
        }
        It "Line '<LineContent>' Should Match 'o'" {
            $PSItem.LineContent | Should -Match 'o'
        }
    }
}
```

The `$ContainerParams` variable changes as follows:

```
$ContainerParams = @{
    Path = '.\Tests\Get-FileContent.Tests.ps1'
    Data = @{ FullNameList = @(
        '.\Import\psversiontable-pwsh.xml'
        '.\Import\psversiontable5.1.xml'
    )}
}
```

The `FullName` parameter is now `FullNameList` and has the `[string[]]` type accelerator instead of `[string]`. The extra brackets in `[string[]]` changes the parameter's type from a string object to a string array. In the `BeforeDiscovery` block, a `foreach` loop iterates through each path passed to the `$FullNameList` parameter. The `$FileName` variable is populated with the file name of the current object in the loop, and added as an additional key/value in `$HList`. The FileName key is then used in the `Context` block's description to distinguish between the files under test. The rest of the test code is the same as before. You now have a test script that can accept a list of file paths and generate tests for each line in each file.

6.6 Conclusions

With parameters, you can write test code that is dynamic and reusable with external data from any source. Parameter values can be combined with code executed during test *discovery* and the `-Foreach` parameter of Pester test control blocks such as `Context` and `It` that get executed during test *run*. Your actions and assertions are written once, and Pester generates a separate test for each data set passed to the tests. The results can be output in a variety of formats for further automated or interactive processes to identity and fix errors in your code.

6.7 Further Reading

- Official Pester v5 Documentation[34]

[34] https://pester.dev/docs/quick-start

- Official Pester v4 Documentation[35]
- Data Driven Tests article[36]
- Breaking changes in Pester v5[37]
- Migrating from Pester v4 to v5[38]
- Migrating from Pester v3 to v4[39]
- Pester Articles List[40]
- Pester Courses List[41]
- Pester on GitHub[42]

[35] https://pester.dev/docs/v4/quick-start

[36] https://pester.dev/docs/usage/data-driven-tests

[37] https://pester.dev/docs/migrations/breaking-changes-in-v5

[38] https://pester.dev/docs/migrations/v4-to-v5

[39] https://pester.dev/docs/migrations/v3-to-v4

[40] https://pester.dev/docs/additional-resources/articles

[41] https://pester.dev/docs/additional-resources/courses

[42] https://github.com/pester/pester/

III PowerShell in Depth

> *“Give a person PowerShell and they will do their job. Teach a person PowerShell and they will automate their job.”* — Michael Zanatta

For most administrators, PowerShell is a simple tool used to perform simple tasks quickly. However, when used correctly, PowerShell is a powerful automation tool that can solve complex problems with efficiency. This section will cover advanced PowerShell concepts in code design and refactoring, progressive conditions, logging, and Infrastructure as Code (IaC). It features deep-dive topics such as interpolation, data management, bitwise operators, and operator precedence.

7. Refactoring PowerShell

Refactoring code is an everyday task that improves the code's design and implementation structure, improving the coder's skills as new programming concepts are introduced, learned, and understood. This chapter explores the concepts of how to refactor your code to make it more readable and maintainable, including:

- Expanding on the PowerShell pipeline to simplify code logic.
- Expanding on splatting to simplify cmdlet execution.
- Utilizing interpolation to make strings more readable and maintainable.
- A deep dive into refactoring functions, making them more readable and maintainable.
- Writing better code.
- Data management.

7.1 Expanding on the Pipeline

When developing scripts, the PowerShell pipeline is often forgotten and substituted for PowerShell's programmable syntax. Scripts are commonly written in the traditional programming syntax, preferring to have additional logic with less use of the PowerShell pipeline. The objective of writing code is to be 'simple and testable', needing minor changes to the design of the existing PowerShell code.

Introducing PowerShell Grouping, Sorting, and Filtering (GSF). GSF is a simplification of functional programming, simplifying logic through function expressions instead of sequenced steps. The PowerShell GSF approach simplifies code by using pipeline grouping, sorting, and filtering to streamline the readability and execution of code. Grouping uses the `Group-Object` cmdlet to group objects based on object properties. The `Sort-Object` and `Select-Object` cmdlets are used to sort and create object properties. The `Where-Object` cmdlet or the 'filtering left' technique is used to filter objects based on criteria.[1]

In the following example, duplicate processes are filtered, grouped, and sorted:

[1] Microsoft. (2021, Jun. 10). *PowerShell 101: Chapter 4 - One-liners and the pipeline - Filtering Left.* Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/scripting/learn/ps101/04-pipelines#filtering-left. [Accessed: May. 25, 2022].

Example 1: Filtering, grouping, and sorting process information

```
# Apply Filtering Left by selecting the processes that we need.
# Where-Object isn't required here.
$DuplicateProcesses = Get-Process node, pwsh, sh, git |
    # The object properties are selected before
    # the grouping since it's more difficult
    # after Group-Object has executed.
    Select-Object -Property Name, Id |
    Group-Object -Property Name |
    Sort-Object -Property Name
```

The GSF approach simplifies the required data or information before invoking it. In some circumstances, the `Where-Object` cmdlet increases processing time by lacking looping controls and executing a condition statement on each item within the array or list. The `Where()` method is a suitable alternative; however, it lacks pipeline support.[2] Below is an example of some PowerShell code that can be refactored using the GSF approach.

Example 2: Matching Windows process and service information using a traditional looping approach

```
#Requires -Version 5.1
#
# In this script, the Process and Windows Service
# data is formatted and joined:

$WindowsServices = Get-CimInstance -ClassName Win32_Service
$WindowsProcesses = Get-Process
$NewObj = @()

# Iterate through each of the processes and find Windows processes.
foreach ($WindowsProcess in $WindowsProcesses) {

    $matchedService = $WindowsServices | ForEach-Object {
        if ($WindowsProcess.Id -eq $_.ProcessId) { Write-Output $_ }
    }

    if ([Array]$matchedService.Count -eq 0) { continue }

    $NewObj += $WindowsProcess | Select-Object *,
        @{ Name = 'WindowsService'; Expression = { $matchedService } }

}

$NewObj | Where-Object { $_.WindowsService -ne $null } |
    Export-Clixml ProcessesWithServiceInfo.clixml
```

This can be refactored into:

[2] Microsoft. (2022, Mar. 17). *About Methods (Microsoft.PowerShell.Core) - ForEach and Where methods*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_methods#foreach-and-where-methods. [Accessed: May. 25, 2022].

Example 3: Refactoring the code from example 2 using grouping, sorting, and filtering

```
# Get the Items that are needed.
$WindowsServices = Get-CimInstance -ClassName Win32_Service

# Let's slow down here and explain each pipline step:

# Perform the Process Lookup, but since we have the Windows Services
# we can 'filter-left' and parse the Process id's directly into the cmdlet.
# This removes the Where-Object, filtering processes for only Windows Services.
# By utilizing filtering left, the number of objects have been reduced
# improving script execution performance.
Get-Process -Id $WindowsServices.ProcessId |
    # We can use select expressions to attach the matched
    # Windows Service object to the process object
    Select-Object *, @{
        Name = "WindowsService"
        Expression = {
            # We need to declare a variable here since the pipeline
            # token is lost when piped into where-object.
            $processId = $_.Id
            $WindowsServices | Where-Object { $_.ProcessId -eq $processId }
        }
    } |
    # Finally Export to CLIXML
    Export-Clixml ProcessesWithServiceInfo.clixml
```

Notice the performance difference between the two examples:

```
Seconds           : 45
Milliseconds      : 232

Seconds           : 15
Milliseconds      : 243
```

Notice how the first example is significantly slower than the second example (by 30 seconds). Why? The final filtering is done *after* the processing is completed, whereas in the second example, the filtering was applied *before* using the 'filtering left' technique.

7.2 Expanded Splatting

Splatting is a concept that's used to simplify the execution of PowerShell cmdlets or parameterized script blocks, by defining parameters within an `array` or `Hashtable`.[3] The `array` or `Hashtable` is parsed into the expression using an `@` operator. You can read more about the basics of splatting at Microsoft Docs[4].

Splatting is commonly used to simplify execution; however, it's also used to simplify your logic. Enter *expanded splatting*.

The purpose of expanded splatting is to apply logic to the `Hashtable` rather than the *cmdlet*. Hashtables are modified with:

[3] Microsoft. (2022, Mar. 19). *About Splatting (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_splatting. [Accessed: May. 25, 2022].

[4] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_splatting

- `$Hashtable.NewKey` to define a new key within the `Hashtable`. Previously, the `Add()` method was used. Today, PowerShell keys are automatically created by declaring the new key within the table (`Hashtable.NewKey`).
- `$Hashtable.Remove()` to remove a key from the `Hashtable`.
- `$Hashtable.ExistingKey = 'Value`', to update the value for an existing key.

Example 4: Working with hashtables in PowerShell

```
#
# Example 1: Basic Usage with a Hashtable
# Define the Hashtable
$Hashtable = @{
    Name = 'Michael Zanatta'
    Age = 21
    Occupation = 'PowerShell Developer'
    TreeColor = 'Green'
}

# Adding a Key
$Hashtable.Salary = '$50000'

# Removing a Key
$Hashtable.Remove('TreeColor')

# Updating a Key
$Hashtable.Age = 30

# Output the Hashtable
$Hashtable
```

```
# Example 1:
Name                           Value
----                           -----
Name                           Michael Zanatta
Occupation                     PowerShell Developer
Age                            30
Salary                         $50000
```

Using these concepts, PowerShell code can be refactored to be simpler by applying logic to the parameterized hashtable rather than the cmdlet itself. Typical parameters are defined initially within the hashtable and then transformed by the logic. In the example below is some PowerShell and some logic without splatting:

Example 5: A traditional approach to cmdlet parameter selection

```
function Do-Something {
    param($Parameter1, $Parameter2, $Parameter3, $Parameter4)
    $PSBoundParameters.Keys.ForEach{
        '{0} = {1}' -f $_, $PSBoundParameters[$_]
    }
}

$Condition2 = $true

if ($Condition) {
    Do-Something -Parameter1 'Value' -Parameter2 'Value2'
}
elseif ($Condition2) {
    Do-Something -Parameter1 'Value' -Parameter3 'Value3'
}
```

```
Parameter1 = Value
Parameter2 = Value3
```

This can be refactored with splatting to make it easier to read:

Example 6: Selecting cmdlet parameters using hashtable splatting

```
# Parameter1 is a common parameter that's used in both lines of logic.
$Params = @{
    Parameter1 = 'Value'
}

$Condition2 = $true

# The Logic can be applied now:
# Either add the key 'Parameter2' or 'Parameter3' to the Hashtable.
if ($Condition) { $Params.Parameter2 = 'Value2' }
elseif ($Condition2) { $Params.Parameter3 = 'Value3' }

# Splat the contents of the Hashtable into the cmdlet.
Do-Something @Params
```

```
Parameter1 = Value
Parameter3 = Value3
```

Let’s expand with a detailed example. Below, several conditions are defined that need refactoring:

Example 7: More complex parameter permutations using the traditional approach

```
$Condition2 = $true

if ($Condition) {
    Do-Something -Parameter1 'Value' -Parameter2 'Value2'
}
elseif ($Condition2) {
    # Splatted for line formatting in book.
    $Params = @{
        Parameter1 = 'NewValue'
        Parameter3 = 'Value3'
        Parameter4 = 'Value4'
    }
    Do-Something @Params
}
elseif ($Condition3) {
    Do-Something -Parameter3 'Value3'
}
else {
    Do-Something -Parameter1 'Value'
}
```

```
Parameter4 = Value4
Parameter3 = Value3
Parameter1 = NewValue
```

To refactor this code:

- Declare the `Hashtable` with the common parameters that occur to all the items within the cmdlet. In this instance, it's `Parameter1`, which is present in all the conditions. The `else` statement is removed since `Parameter1` is declared (implicitly) in the `Hashtable`.
- `Condition2` changes items in `Parameter1`, and adds keys `Parameter3` and `Parameter4`.
- `Condition3` removes `Parameter1`, and adds `Parameter3`.

Example 8: Complex parameter permutations using splatting

```
$Condition2 = $true

# Parameter1 is the Common Value; set it initially.
$Params = @{
    Parameter1 = 'Value'
}

# Add Parameter2
if ($Condition) { $Params.Parameter2 = 'Value2' }

if ($Condition2) {
    # Update Parameter1
    $Params.Parameter1 = 'NewValue'
    # Add Parameter3 and Parameter4
    $Params.Parameter3 = 'Value3'
    $Params.Parameter4 = 'Value4'
}

if ($Condition3) {
    # Remove Parameter1
    $Params.Remove('Parameter1')
    $Params.Parameter3 = 'Value3'
}

Do-Something @Params
```

```
Parameter4 = Value4
Parameter3 = Value3
Parameter1 = NewValue
```

Much better!

7.3 Interpolation

Before PowerShell, *VBScript* was the scripting language for Windows System Administrators. VBScript lacked modern features, which made it difficult to use. PowerShell changed that; it was derived from C#, inheriting the language structure, management, and syntax. In PowerShell, string management switched from *concatenation* to *interpolation*. Interpolation is a technique where string data is substituted into a string using different methods. This segment explores variable substitution and string formatting.

7.3.1 Variable Substitution

Variable substitution is a method of interpolation where variables are declared within the string and substituted on execution. Variable substitution uses *expandable* (non-literal) strings `""` for interpolation. When referencing object properties, use the subexpression operator `$()` to wrap the object and its property within the expandable string:[5]

Example 9: Variable substitution in expandable strings

```
#
# Example 1: Basic Substitution
$value = 'string!'
"This is a $value"
#
# Example 2: Substitution using an Object
$obj = [PSCustomObject]@{
    Property = "string inside a property!"
}
"This is a $($obj.Property)"
#
# Example 3: Multiple string substitution
$value1 = 'is a'
$value2 = 'string!'
"This $value1 $value2"
```

[5] Microsoft. (2022, Mar. 19). *About Operators (Microsoft.PowerShell.Core) - Subexpression operator*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_operators#subexpression-operator–. [Accessed: May. 25, 2022].

```
# Example 1:
This is a string!

# Example 2:
This is a string inside a property!

# Example 3:
This is a string!
```

The drawbacks of using this method are increasing complexity and lack of formatting capabilities. Three object properties are interpolated into a string in the following example. Note the complexity of the PowerShell before and after the refactorization using the `-join` operator.

Example 10: Using the -join operator in place of variable substitution

```
#
# Top Level Object used in the examples
$obj = [PSCustomObject]@{
    Property       = 'string inside a property!'
    SecondProperty = 'another string!'
    ThirdProperty  = 'And another!'
}
#
# Example 1: Completed String Interpolation.

# Note the complexity of the string.
"Initial $($obj.Property) $($obj.SecondProperty) $($obj.ThirdProperty)"
#
# Example 2: Refactor the initial interpolation.

# We can use the join operator to join the three strings together and then
# insert it into the output string. Note the complexity. It's not preferred.
$string = $($obj.Property), $($obj.SecondProperty),
          $($obj.ThirdProperty) -join ' '
"This is the initial $string"
```

```
# Example 1:
Initial string inside a property! another string! And another!

# Example 2:
This is the initial string inside a property! another string! And another!
```

Variable substitution is best used for simple string interpolation. For more complex substitution, use the `-f` (format) operator with composite formatting.

7.3.2 Using the Format (-f) Operator

Format operator syntax:[6]

[6] Microsoft. (2022, Mar. 19). *About Operators (Microsoft.PowerShell.Core) - Format operator.* Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_operators#format-operator-f. [Accessed: May. 25, 2022].

```
"String with placeholders" -f $arrayOfPlaceHolders
```

Composite formatting syntax:

```
"{ Index[,Alignment][:FormatString] }" -f $arrayOfPlaceHolders
```

The format (`-f`) operator is a relatively unknown feature used to interpolate and format literal and expandable strings within .NET languages.

On the left side, the string is defined with placeholders containing array indexes to associate them. These are called *format items*.[7] Format items are defined as curly braces within the string `{}`, which contains the composite formatting syntax:

1. The `Index` component denotes the placeholder array index, starting at `0`.
2. *(Optional)* The alignment component. An optional item used to denote the alignment position of the string.
3. *(Optional)* The format string component. An optional item used to denote what string formatting is to be applied.

Note that the delimiter to separate the index and alignment is a comma (`,`), and the secondary delimiter is a colon (`:`). On the right side is an array of items nested in a root object type (for example, `String`, `Int`, or `Char`)

The format operator performs simple interpolation without the `Alignment` and `FormatString` properties. In the following example, PowerShell performs a basic interpolation by adding `String` and `!` to the initial string. Note the format items and the index values (`{IndexNumber}`) on the left side of the string:

Example 11: Using the -f format operator

```
"This is a {0}{1}" -f 'string', '!'
```

```
This is a string!
```

If a format item's index value on the left side is out of range of the array on the right side, an error is thrown:

[7] Microsoft. (2021, Nov. 20). *Composite formatting*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/standard/base-types/composite-formatting. [Accessed: May. 25, 2022].

Example 12: Out of range format items result in errors

```
# Note the {2} index value. There is no third item in the array
"This is a {0}{1}{2}" -f 'string', '!'
```

```
InvalidOperation: Error formatting a string: Index (zero based) must be greater
than or equal to zero and less than the size of the argument list..
```

It's possible to have additional array items on the right side, but unless explicitly defined within the format item, it won't be interpolated into the string:

Example 13: Additional placeholder array items are ignored

```
# Note the extra '?' in the right side array.
"This is a {0}{1}" -f 'string', '!', '?'
```

```
This is a string!
```

Format items on the left side don't need to be in order in the string. However, it's recommended to order the left side for readability and maintainability:

Example 14: The position of format items in the composite format string doesn't matter

```
# Note that 1 and 0 are swapped around.
"This is a {1}{0}" -f 'string', '!'
```

```
This is a !string
```

Format item indexes can be used multiple times within the left side string:

Example 15: Format items are reusable

```
# Note the repeated use of the format item.
"This is a {0} {0} {0}!" -f 'string'
```

```
This is a string string string!
```

7.3.2.1 Alignment Component

The optional *Alignment* component (`{Index,Alignment:FormatString}`) is used to define the preferred field width by padding white spaces into the string. A positive number `{Index,n}` right-aligns `n` characters from the *rightmost side* of the string by applying padding to the *left side of the string*. A negative number `{Index,-n}` left-aligns `n` characters from the *leftmost side* of the string by applying padding to the *right side of the string*. In the example, left and right alignments are shown:

Example 16: Using the Alignment component in format items

```
#
# Example 1: Basic Right Alignment
$String = "{0,10}" -f 'value'
# Note that the string length is exactly 10.
$String.Length
$String
#
# Example 2: Same as Example 1, wrapped with '' quotation marks
$String = "'{0,10}'" -f 'value'
$String.Length
$String
#
# Example 3: Basic Left Alignment
$String = "{0,-10}" -f 'value'
# Note that the string length is 10.
$String.Length
$String
#
# Example 4: Same as Example 3, wrapped with '' quotation marks
$String = "'{0,-10}'" -f 'value'
$String.Length
$String
#
# Example 5: The difference between a left and right-hand alignment

# LeftHand
$LHString = "'{0,-10} {0,-10}'" -f 'value'
# Note that the string length of each format item is exactly 10.

# RightHand
$RHString = "'{0,10} {0,10}'" -f 'value'
# Note that the string length of each format item is exactly 10.

$RHString
$LHString
#
# Example 6: What happens if we left-hand and right-hand in the same string?
$String = "'{0,10}{0,-10}'" -f 'value'
$String2 = "'{0,-10}{0,10}'" -f 'value'
$String
$String2
```

```
# Example 1: Basic Right Alignment
10
     value

# Example 2: Same as Example 1, wrapped with '' quotation marks
12
'     value'

# Example 3: Basic Left Alignment
10
value

# Example 4: Same as Example 3, wrapped with '' quotation marks
12
'value     '

# Example 5: The difference between a left and right-hand alignment.
'     value      value'
```

```
'value      value     '

# Example 6: Left and right-hand alignment in the same string
'     valuevalue     '
'value          value'
```

Below are some observations to note from the examples above:

- In *Example 1*, 'value' is aligned to the right side of the 10 character string. The placeholder value (string `'value'`) is included within the alignment padding.
- *Example 2* is the same as *Example 1*; the string is wrapped with quotes to show the start and end of the string.
- In *Example 3*, 'value' is aligned to the left side of the 10 character string. Again, note that the placeholder value is **included** within the alignment padding.
- In *Example 4*, quotes are wrapped around the string from *Example 3* to show the start and end of the string.
- In *Example 5*, note the difference between the left-hand and right-hand formatting when used with multiple format items. Notice how the padding is relative to its own format item.
- In *Example 6*, note how the padding is again relative to its own format item, making it impossible to overlap text with each other.

In the example above, the placeholder string length was smaller than the alignment value, so what happens when it's larger? The padding is ignored when the placeholder value string length is equal to or larger than the alignment value.

Example 17: Placeholder values aren't truncated when they're longer than the alignment value

```
#
# Example 1
# The string length 'value' is larger then three characters when
# using the left-hand alignment.
$String = "'{0,3}'" -f 'value'
$String
#
# Example 2
# Let's swap sides and now try the right-hand
# alignment.
$String = "'{0,-3}'" -f 'value'
$String
```

```
# Example 1
'value'

# Example 2
'value'
```

7.3.2.2 Format String Component

The format string component defines what composite formatting type is to be performed on the format item. The following table lists some custom formatting types (with examples):

Key	Description	Example	Output
`:c`	Formats a number as the local currency defined in the culture	`'{0:c}' -f 15000`	`'$15,000.00'`
`:dn` *(n = Number of decimal places)*	Adds leading zeros to the beginning of a *whole number*. If `n` exceeds the length, no padding is added.	`'{0:d7}' -f 15000`	`'0015000'`
`:e`	Formats a number in exponential notation	`'{0:e}' -f 10000000000`	`'1.000000e+010'`
`:fn` *(n = Number of decimal places)*	Formats as a fixed point decimal with following zeros as defined by `n`. If the length of the decimal exceeds `n`, it's rounded. `'{0:f1}' -f 15.14` is rounded to: `'15.1'`	`'{0:f4}' -f 15.14`	`15.1400`
`:gn` *(n = Number of digits)*	Formats a number in its more compact format, either fixed or in exponential notation.	`'{0:g1}' -f 15.14`	`'2e+01'`
`:nP` *(P = Number of decimal places)*	Formats a number as the local culture representation. `:n` is similar to `:f`, where `P` is used to define the number of decimal places. If the length of the decimal exceeds `n`, it's rounded.	`'{0:n2}' -f 150000.1455`	`'150,000.15'`
`:pn` *(n = Number of decimal places)*	Formats a decimal number as a percentage. `n` is similar to `:fn`, where `n` is used to define the number of decimal places. If the length of the decimal exceeds `n`, it's rounded.	`'{0:p}' -f .1`	`'10%'`
`:x`	Formats a number to hexadecimal	`'{0:x}' -f 30`	`'1E'`
`:hh`	Formats a `Datetime` object to a 2-digit hour	`'{0:hh}' -f (Get-Date)`	`06`
`:HH`	Formats a `Datetime` object to a 2-digit 24-hour format	`'{0:hh}' -f (Get-Date)`	`20`
`:mm`	Formats a `Datetime` object to a 2-digit minute	`'{0:mm}' -f (Get-Date)`	`15`
`:ss`	Formats a `Datetime` object to a 2-digit second	`'{0:ss}' -f (Get-Date)`	`15`

Key	Description	Example	Output
`:t`	Formats a `Datetime` object to compact time	`'{0:t}' -f (Get-Date)`	`6:23 AM`
`:tt`	Formats a `Datetime` object AM or PM	`'{0:tt}' -f (Get-Date)`	`AM`
`:d`	Formats a `Datetime` object to a compact date according to the local culture.	`'{0:d}' -f (Get-Date)`	`2/2/2022`
`:dd`	Formats a `Datetime` object to a 2-digit day of the month	`'{0:dd}' -f (Get-Date)`	`15`
`:ddd`	Formats a `Datetime` object to the compact day of the week	`'{0:ddd}' -f (Get-Date)`	`'Sun'`
`:dddd`	Formats a `Datetime` object to a day of the week	`'{0:dddd}' -f (Get-Date)`	`'Sunday'`
`:M`	Formats a `Datetime` object to the full month name and day of the month.	`'{0:M}' -f (Get-Date)`	`'February 20'`
`:MM`	Formats a `Datetime` object to a 2-digit month.	`'{0:MM}' -f (Get-Date)`	`'02'`
`:MMM`	Formats a `Datetime` object to the compact month.	`'{0:MMM}' -f (Get-Date)`	`'Feb'`
`:MMMM`	Formats a `Datetime` object to the full month.	`'{0:MMMM}' -f (Get-Date)`	`'Febuary'`
`:y`	Formats a `Datetime` object to the full month name and the year.	`'{0:y}' -f (Get-Date)`	`'February 2022'`
`:yy`	Formats a `Datetime` object to the 2-digit year of the century.	`'{0:yy}' -f (Get-Date)`	`'22'`
`:yyyy`	Formats a `Datetime` object to a 4-digit year.	`'{0:yyyy}' -f (Get-Date)`	`'2022'`

You can find out more about composite formatting at Microsoft Docs[8].

When formatting dates, multiple date formats can be grouped into the same format string:

Example 18: Composite date formats in format items

```
'{0:ddddMMMMyyyy}' -f (Get-Date)
```

[8] https://learn.microsoft.com/en-us/dotnet/standard/base-types/composite-formatting

```
SundayFebruary2022
```

This can be expanded further by adding some literal formatting within the format strings component. In this example, a space between the format strings is added:

Example 19: Spaces are treated literally inside format string components

```
'{0:dddd d MMMM yyyy}' -f (Get-Date)
```

```
Sunday 20 February 2022
```

Strings can be wrapped within the format string component by using a literal or an expandable string (`''`). By wrapping the strings within a string, special characters are ignored. If using the same string type to escape, use double quotation marks (`''` or `""`). In the example below, `'Day :'`, `'Month:'` and `'Year:'` are added to the format string. However, since wrapped in a string, special characters are ignored:

Example 20: Including literal characters in format string components

```
#
# Example 1: String wrapping using single quotation marks within
# an expandable string
"{0:'Day: 'dddd d', Month: 'MMMM', Year: 'yyyy}" -f (Get-Date)
#
# Example 2: Same as Example 1, however using two single quotation
# marks to escape the literal string.
'{0:''Day: ''dddd d'', Month: ''MMMM'', Year: ''yyyy}' -f (Get-Date)
#
# Example 3: Same as Example 1, however using double quotation
# marks within a literal string.
'{0:"Day: "dddd d", Month: "MMMM", Year: "yyyy}' -f (Get-Date)
#
# Example 4: Same as Example 1, however using two double quotation
# marks to escape the expandable string.
"{0:""Day: ""dddd d"", Month: ""MMMM"", Year: ""yyyy}" -f (Get-Date)
```

```
# Example 1:
Day: Sunday 20, Month: February, Year: 2022

# Example 2:
Day: Sunday 20, Month: February, Year: 2022

# Example 3:
Day: Sunday 20, Month: February, Year: 2022

# Example 4:
Day: Sunday 20, Month: February, Year: 2022
```

The `ToString()` method also accepts the format string operators.[9]

[9] Microsoft. (2022, Mar. 11). *Overview: How to format numbers, dates, enums, and other types in .NET.* Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/standard/base-types/formatting-types. [Accessed: Jun. 08, 2022].

Example 21: Format string operators with ToString()

```
#
# Example 1: Formatting a date
(Get-Date).ToString('dddd d MMMM yyyy')

#
# Example 2: Formatting a number
(0.42).ToString('p0')
```

```
# Example 1:
Sunday 20 February 2022

# Example 2:
42%
```

If two format string operators are ambiguous and need to be separated logically, use an empty literal span:

Example 22: Separating ambiguous operators with an empty literal span

```
# Example 1: Five 'd' operators results in the long day name
'{0:ddddd}' -f (Get-Date)
#
# Example 2: Separating these with an empty literal span solves the issue
'{0:dddd""d}' -f (Get-Date)
```

```
# Example 1:
Sunday

# Example 2:
Sunday20
```

The advantages of using the format operator expand from interpolation to string formatting, simplifying scripts. Some drawbacks are that the substitution and format operators are only used for *static* strings, making it difficult to construct and interpolate many strings.

7.4 Refactoring Functions

PowerShell functions are overlooked when refactoring code; however, it's one of the essential methods of structuring code to be maintainable and testable. This segment explores several methodologies that are used to make code more maintainable and testable. These are:

- Simplifying functions to perform a singular task.
- Using typecasting on parameters.
- Using advanced function parameters.
- Using approved verbs.
- Using a singular output object type.

7.4.1 Simplify Functions to Perform a Singular Task

You should structure functions to be as simple as possible or perform a singular task. This improves the testability and readability of the code. In the example below, the function `Process-File` is used to perform several tasks; unzipping, verifying files, and inserting 'test' to the end of each of the expanded files:

Example 23: A single function performing multiple tasks

```
function Process-File {
    param($Filename, $VerifyHashes)

    #
    # Get the File object, expand it, and return an SHA256 hash of the contents

    $File = Get-Item -LiteralPath $Filename
    $ExpandPath = Join-Path ([System.IO.Path]::GetTempPath()) (New-Guid)
    $File | Expand-Archive -DestinationPath $ExpandPath

    $GetChildItemParams = @{
        LiteralPath = $ExpandPath
        File        = $true
        Recurse     = $true
    }

    $Hashes = (Get-ChildItem @GetChildItemParams | Get-FileHash).Hash

    #
    # Perform a comparision on the hashes to ensure

    $CompareObjectParams = @{
        ReferenceObject  = $Hashes
        DifferenceObject = $VerifyHashes
    }

    $Difference = Compare-Object @CompareObjectParams

    if ($Difference.InputObject.Count -ne 0) {

        $MailMessageParams = @{
            To      = 'Helpdesk'
            From    = 'NoReply'
            Subject = 'File Transfer Failed'
        }

        Send-MailMessage @MailMessageParams
        throw "Issue with the hash"

    }

    #
    # Insert 'Test' at the end of each of the files
    Get-ChildItem @GetChildItemParams | Add-Content -Value 'Test'

    return $ExpandPath
}
```

While the code is relatively succinct, it's not maintainable or testable. The function is performing several tasks, making it difficult to test. `Process-File` can be refactored further, splitting out the file extraction, the validation, and the file transformation, wrapped with an additional function to join them together:

Example 24: Refactoring the function from Example 24 yields discrete functions for each task

```
#
# The First Function to Expand the File
function Extract-File {
    param ($Filename)

    $File = Get-Item -LiteralPath $Filename
    $ExpandPath = Join-Path ([System.IO.Path]::GetTempPath()) (New-Guid)
    $File | Expand-Archive -DestinationPath $ExpandPath

    return $ExpandPath
}

#
# The Second Function to Test Exported Files
function Test-ExportedFiles {
    param($LiteralPath, $VerifyHashes)

    $GetChildItemParams = @{
        LiteralPath = $LiteralPath
        File        = $true
        Recurse     = $true
    }

    $CompareObjectParams = @{
        ReferenceObject  = $(Get-ChildItem @GetChildItemParams |
            Get-FileHash).Hash
        DifferenceObject = $VerifyHashes
    }

    $Difference = Compare-Object @CompareObjectParams

    if ($Difference.InputObject.Count -ne 0) { return $false }

    $true

}

#
# The third to Append the string to the end of the file.
function Add-StringToEndOfFile {
    param($FilePath, $Value)

    Get-ChildItem $FilePath -File -Recurse | Add-Content -Value $Value

}

#
# The Final to join them together.
function Process-File {
    param($Filename, $VerifyHashes)

    # Extract the File to C:\Windows
    $ExpandPath = Extract-File -Filename $Filename

    # Test the files
    $Params = @{
        LiteralPath = $ExpandPath
        VerifyHash  = $VerifyHashes
    }

    if (-not (Test-ExportedFiles @Params)) {

        $MailMessageParams = @{
            To      = 'Helpdesk'
            From    = 'NoReply'
            Subject = 'File Transfer Failed'
        }

        Send-MailMessage @MailMessageParams
        throw "Issue with the hash"
    }

    #
    # Add Test to the end of every file.
    Add-StringToEndOfFile -FilePath $ExpandPath -Value 'Test'

    return $ExpandPath
}
```

7.4.2 Use Typecasting on Parameters

Always typecast parameters with `[type]$parameterName` ensure that parameters are the correct type.[10] [11] In the following example, the function has two parameters: `$FileName` and `$ExtractPath`. These parameters aren't typecasted, so any object type can be parsed:

Example 25: A function with unlimited parameter types

```
function Extract-File {
    param ($FileName, $ExtractPath)

    $File = Get-Item -LiteralPath $FileName
    $File | Expand-Archive -DestinationPath $ExtractPath

}
```

To make the code more maintainable/testable, setting the type of the parameters identifies the exact allowed parameter types. In the following example, the parameters `$FileName` and `$ExtractPath` are protected by adding the `[String]` type:

[10] Microsoft. (2021, Jul. 30). *PowerShell Language Specification: Chapter 6 - Conversions*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/scripting/lang-spec/chapter-06. [Accessed: May. 26, 2022].

[11] PowerShell Team. (2007, Oct. 29). *Dynamic Casting*. Microsoft Dev Blogs. [Online]. Available: https://devblogs.microsoft.com/powershell/dynamic-casting/. [Accessed: May. 26, 2022].

Example 26: The function from Example 26, with typecast parameters

```
function Extract-File {
    param ([String]$FileName, [String]$ExtractPath)

    $File = Get-Item -LiteralPath $FileName
    $File | Expand-Archive -DestinationPath $ExtractPath

}
```

7.4.3 Use Advanced Function Parameters

Using advanced function parameters provides different mechanisms to simplify your code when the script/function is invoked, by requiring or validating the correct input.[12] To simplify PowerShell functions, *parameter sets* and *mandatory parameters* can be used. Parameter sets simplify the execution experience by removing parameters that aren't required for a particular combination. In the following example, parameter sets are used to simplify the parameters:

Example 27: Restricting parameter combinations with parameter sets

```
#
# Function Code:
function Invoke-Something {
    [CmdletBinding(DefaultParameterSetName = 'P1')]
    param(
        [Parameter(ParameterSetName = 'P1')]
        [string]$Parameter1,
        [Parameter(ParameterSetName = 'P2')]
        [string]$Parameter2,
        [Parameter(ParameterSetName = 'P1')]
        [Parameter(ParameterSetName = 'P2')]
        [string]$Parameter3
    )

    # Print out the parameter set
    $PSCmdlet.ParameterSetName

}
#
# Example 1: Invoke the Function with no parameter
Invoke-Something
#
# Example 2: Invoke the Function with 'Parameter1'
Invoke-Something -Parameter1 ''
#
# Example 3: Invoke the Function with 'Parameter2'
Invoke-Something -Parameter2 ''
#
# Example 4: Invoke the Function with 'Parameter3'
Invoke-Something -Parameter3 ''
#
# Example 5: Invoke the Function with 'Parameter1' and 'Parameter2'
Invoke-Something -Parameter1 '' -Parameter2 ''
```

[12] Microsoft. (2021, Jun. 10). *About Functions Advanced (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_functions_advanced. [Accessed: May. 26, 2022].

```
# Example 1:
P1

# Example 2:
P1

# Example 3:
P2

# Example 4:
P1

# Example 5:
Invoke-Something: Parameter set cannot be resolved using the specified
named parameters. One or more parameters issued cannot be used together
or an insufficient number of parameters were provided.
```

Let's review the examples from the previous tests:

- In *Example 1*, the test shows that the default parameter set is 'P1' as parameters weren't included.
- In *Example 2*, the test shows the parameter set 'P1' is applied when `-Parameter1` is specified.
- In *Example 3*, the test shows the parameter set 'P2' is applied when `-Parameter2` is specified.
- In *Example 4*, `-Parameter3` is parsed being present in both parameter sets 'P1' and 'P2'. Since no other parameters in the corresponding parameter set are included, PowerShell defaults to the `DefaultParameterSetName` in the `CmdletBinding` attribute, which is 'P1'.
- In *Example 5*, parameters `-Parameter1` and `-Parameter2` (from different parameter sets) are parsed into the function, raising an error. The error raised demonstrates that parameters from different parameter sets can't mix.

Adding mandatory parameters ensures that parameter inputs are met, simplifying the internal function code. An important thing to remember is that when using parameter sets, ensure you set the `DefaultParameterSetName` argument within the `CmdletBinding` attribute:

Example 28: Always define a default parameter set if using them

```
function Invoke-Something {
    # Set the CmdletBinding Attribute and Set the Default Parameter Set Name
    [CmdletBinding(DefaultParameterSetName = 'P1')]
    param(
        # Define the Parameter and the ParameterSetName
        [Parameter(ParameterSetName = 'P1')]
        [string]$Parameter1
    )
}
```

When the *Mandatory* attribute is set, and no parameter input is found, PowerShell prompts the user to supply values for the parameter, as seen in the example below:

Example 29: Omitting mandatory parameters results in a prompt for input

```
# Invoke-Something has a mandatory parameter here
Invoke-Something
```

```
cmdlet Invoke-Something at command pipeline position 1
Supply values for the following parameters:
```

In the following example, the *Mandatory* attribute is used to filter the input parameters:

Example 30: Using mandatory parameters to ensure an input is present

```
#
# Function Definition
function Invoke-Something {
    param(
        # Add the Parameter Attribute and Include
        # the Mandatory Argument
        [Parameter(Mandatory)]
        [String]$Parameter1,
        [String]$Parameter2
    )
    'Complete!'
}
#
# Example 1: Call the Function without -Parameter1
Invoke-Something -Parameter2 ''
#
# Example 2: Call the Function with a -Parameter1 value
# and -Parameter2 without a value.
Invoke-Something -Parameter1 'Value' -Parameter2 ''
```

```
# Example 1:
Supply values for the following parameters:
Parameter1:
Invoke-Something: Cannot bind argument to parameter 'Parameter1'
because it is an empty string.

# Example 2:
Complete!
```

Notice from the example above the use of the *Mandatory* attribute. The error in *Example 1* occurs when an input prompt is left empty for a mandatory parameter.

Another method for simplifying the input parameters is using the *Validate** Attributes to test the input itself. This chapter focuses on the following attributes:

- ValidateSet
- ValidatePattern
- ValidateLength
- ValidateCount
- ValidateNotNull

- ValidateNotNullOrEmpty
- ValidateScript

A notable feature of the *Validate** attributes is that they stack on each parameter value, further narrowing the input requirements. In the following example, *ValidateScript* and *ValidateNotNullOrEmpty* are added. Pay attention to the script block within the *ValidateScript* Attribute:

Example 31: Combining Validate* attributes for a single parameter

```
#
# Example Function
function Invoke-Something {
    param(
        # Add the first attribute
        [ValidateScript({
            # Note! There's PowerShell Code in here!
            Test-Path -Path $_
        })]
        # Add the second one (executed first)
        [ValidateNotNullOrEmpty()]
        [string]$Path
    )
    $Path
}
#
# Example 1: Parse $null into the Parameter
Invoke-Something -Path $null
#
# Example 2: Parse a Valid Path
Invoke-Something -Path "$PSHOME"
#
# Example 3: Parse an invalid Path
Invoke-Something -Path '\:BadPath?'
```

```
# Example 1:
Invoke-Something : Cannot validate argument on parameter 'Path'.
The argument is null or empty. Provide an argument that is not null or empty,
and then try the command again.

# Example 2:
C:\Program Files\PowerShell\7

# Example 3:
Invoke-Something: Cannot validate argument on parameter 'Path'. The "
    # Note! There's PowerShell Code in here!
    Test-Path -Path $_ -IsValid
" validation script for the argument with value "\:BadPath?" did not
return a result of True. Determine why the validation script failed,
and then try the command again.
```

Note from the previous example the different errors that were raised when the string validation failed. In the following example, the same attribute *ValidateScript* is used. However, note the order of execution:

Example 32: The execution order of multiple ValidateScript attributes is last to first

```
#
# Example Function
function Invoke-Something {
    param(
        # First Script Block
        [ValidateScript({
            Write-Host 'First'
            $_ -ne 'D:\Temp'
        })]
        # Second Script Block
        [ValidateScript({
            Write-Host 'Second'
            $_ -ne 'Z:\Temp'
        })]
        # Third Script Block
        [ValidateScript({
            Write-Host 'Third'
            Test-Path -Path $_ -IsValid
        })]
        [string]$Path
    )
    $Path
}
#
# Parse a valid path
Invoke-Something -Path $PSHOME
```

```
Third
Second
First
C:\Program Files\PowerShell\7
```

Note how the execution order is from the bottom-up, not top-down.

Other validation attributes provide different means to simplify your logic. These are:

- `[ValidateSet('Value1', 'Value2')]`. Used to test against an array of string inputs:

Example 33: Specifying a fixed set of inputs with the ValidateSet attribute

```
#
# ValidateSet Function
function Invoke-Something {
    param(
        # Add the first attribute
        [Parameter(Mandatory)]
        [ValidateSet('Value1', 'Value2')]
        [String]$Parameter
    )
    $Parameter
}

#
# Example 1: ValidateSet
Invoke-Something -Parameter 'Value1'

#
# Example 2: ValidateSet: Unknown ParameterValue
Invoke-Something -Parameter 'Unknown'
```

```
# Example 1:
Value1

# Example 2:
Invoke-Something: Cannot validate argument on parameter 'Parameter'.
The argument "Unknown" does not belong to the set "Value1,Value2"
specified by the ValidateSet attribute. Supply an argument that is
in the set and then try the command again.
```

- `[ValidatePattern('RegexPattern')]`. Validates the string input against a regex pattern:[13]

Example 34: Limiting input to strings that match a regex pattern with the ValidatePattern attribute

```
#
# ValidatePattern Function
function Invoke-Something {
    param(
        # Add the first attribute
        [Parameter(Mandatory)]
        [ValidatePattern('^(Value)[0-9]$')]
        [String]$Parameter
    )
    $Parameter
}

#
# Example 1: ValidatePattern
Invoke-Something -Parameter 'Value9'

#
# Example 2: ValidatePattern Bad Input
Invoke-Something -Parameter 'SomeOtherValue'
```

```
# Example 1:
Value9

# Example 2:
Invoke-Something: Cannot validate argument on parameter 'Parameter'.
The argument "SomeOtherValue" does not match the "^(Value)[0-9]$" pattern.
Supply an argument that matches "^(Value)[0-9]$" and try the command again.
```

- `[ValidateLength(MinIntLength, MaxIntLength)]`. Validates the length of the string input:

[13] Microsoft. (2022, Mar. 18). *About Regular Expressions (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_regular_expressions. [Accessed: May. 27, 2022].

Example 35: Controlling input string length with the ValidateLength attribute

```
#
# ValidateLength Function
function Invoke-Something {
    param(
        # Add the first attribute
        [Parameter(Mandatory)]
        # Minimum Length is 5
        # Maxiumum Length is 7
        [ValidateLength(5, 7)]
        [String]$Parameter
    )
    $Parameter
}

#
# Example 1: ValidateLength
Invoke-Something -Parameter 'Value'

#
# Example 2: ValidateLength. Too Short
Invoke-Something -Parameter 'Two'

#
# Example 3: ValiadateLength. Too Long
Invoke-Something -Parameter 'StringisTooLong'
```

```
# Example 1:
Value

# Example 2:
Invoke-Something: Cannot validate argument on parameter 'Parameter'.
The character length (3) of the argument is too short.
Specify an argument with a length that is greater than or equal to "5",
and then try the command again.

# Example 3:
Invoke-Something: Cannot validate argument on parameter 'Parameter'.
The character length of the 15 argument is too long.
Shorten the character length of the argument so it is fewer than or
equal to "7" characters, and then try the command again.
```

- `[ValidateCount(MinIntLength, MaxIntLength)]`. Validates the number of allowed arguments (array items):

Example 36: Limiting the input array size with the ValidateCount attribute

```
#
# ValidateCount Function
function Invoke-Something {
    param(
        # Add the first attribute
        [Parameter(Mandatory)]
        # Minimum Length is 5
        # Maxiumum Length is 7
        [ValidateCount(5, 7)]
        # Note the type of the string
        [String[]]$Parameter
    )
    $Parameter -join ' '
}
```

```

#
# Example 1: Count
Invoke-Something -Parameter 1, 2, 3, 4, 5

#
# Example 2: ValidateLength. Too Short
Invoke-Something -Parameter 1

#
# Example 3: ValiadateLength. Too Long
Invoke-Something -Parameter 1, 2, 3, 4, 5, 6, 7, 8
```

```
# Example 1:
1 2 3 4 5

# Example 2:
Invoke-Something: Cannot validate argument on parameter 'Parameter'.
The parameter requires at least 5 value(s) and no more than 7 value(s) -
1 value(s) were provided.

# Example 3:
Invoke-Something: Cannot validate argument on parameter 'Parameter'.
The parameter requires at least 5 value(s) and no more than 7 value(s) -
8 value(s) were provided.
```

- `[ValidateScript({ #ScriptBlock })]`. Use a PowerShell script block to test the input: The validation process is similar to the script block you use with `Where-Object`, where the `$_` token denotes the current parameter, and the script block needs to return a Boolean value. Other parameter values aren't accessible inside the script block. It's tempting to write comprehensive code but keep it as simple as possible since it can be challenging to test:

Example 37: Validating an input using a script block with the ValidateScript attribute

```
#
# ValidateScript Function

function Invoke-Something {
    param(
        # Add the first attribute
        [Parameter(Mandatory)]
        # In this example, we test the input
        # for a valid filepath
        [ValidateScript({
            # Access the current parameter by
            # using the $_ token:
            Test-Path -LiteralPath $_
        })]
        [String]$Parameter
    )
    $Parameter
}

#
# Example 1: Using a known file path.
Invoke-Something -Parameter "$PSHOME"

#
# Example 2: Using an unknown file path.
Invoke-Something -Parameter 'D:\Bad\FilePath'
```

```
#
# Example 1:
C:\Program Files\PowerShell\7

#
# Example 2:
Invoke-Something: Cannot validate argument on parameter 'Parameter'. The "
    # Access the current parameter by
    # using the $_ token:
    Test-Path -LiteralPath $_
" validation script for the argument with value "D:\Bad\FilePath"
did not return a result of True. Determine why the validation script
failed, and then try the command again.
```

7.4.4 Use Approved Verbs

Approved verbs describe the intended use of the PowerShell function.[14] For a list of approved verbs and suggestions, see the list on Microsoft Docs[15]. Another thing to consider is the expected output object type in conjunction with the verb. For instance, if the verb of the function is 'test', the output of that function should be a boolean result. In the following example, the function `Test-FileContents` returns the contents of the file if successful:

Example 38: This function shouldn't output file content since it's named with the 'Test' verb

```powershell
function Test-FileContents {
    param([String]$LiteralPath)

    $Content = Get-Content -LiteralPath $LiteralPath -Raw
    if ($Content.Length -eq 0) { return $false }
    $Content
}
```

The verb 'Test' shouldn't be returning the contents of the file. Instead, it should return a boolean object type. The code is refactored to return a boolean result in the following example:

Example 39: A more typical behavior for a Test-* function

```powershell
function Test-FileContents {
    param([String]$LiteralPath)

    $Content = Get-Content -LiteralPath $LiteralPath -Raw
    if ($Content.Length -eq 0) { return $false }
    $true
}
```

If you're returning the file content, the 'Get' verb is more appropriate.

[14] Microsoft. (2022, Jan. 02). *Approved Verbs for PowerShell Commands*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/scripting/developer/cmdlet/approved-verbs-for-windows-powershell-commands. [Accessed: May. 26, 2022].

[15] https://learn.microsoft.com/en-us/powershell/scripting/developer/cmdlet/approved-verbs-for-windows-powershell-commands

7.4.5 Use a Singular Output Object Type

Standardizing a function's output type simplifies the testing and debugging processes. If several object types are required, wrap the output within a `[PSCustomObject]` with separate properties for different types. In the example below, the function `Import-FileasXML` imports a file's contents and attempts to typecast the contents as XML. If successful, an `[XML]` object is returned; otherwise, the file contents are returned as a string array `[String[]]` of lines:

Example 40: A function that outputs different data types based on file contents

```
function Import-FileasXML {
    param([String]$LiteralPath)

    $Content = Get-Content -LiteralPath $LiteralPath

    try {
        $Content = [XML]$Content
    } catch {}

    $Content
}
```

This function can be refactored by splitting out the object functionality into two separate properties (`Content` and `XMLContent`):

Example 41: The refactored function always returns a custom object with two properties

```
function Import-FileasXML {
    param([String]$LiteralPath)

    $Content = Get-Content -LiteralPath $LiteralPath

    [PSCustomObject]@{
        Content    = $Content
        XMLContent = try { [XML]$Content } catch { $null }
    }
}
```

The previous example shows that the file content was loaded initially and then added to a `[PSCustomObject]` with two properties: `Content` and `XMLContent`. Within the property `XMLContent`, the file's contents are tentatively parsed as an XML file. If successful, the XML object is added to the `XMLContent` property; otherwise, `$null` is returned.

7.5 Writing Better Code

Whether you are a novice or a seasoned professional reading this book, the goal is always to write simple and maintainable code. Each programming language has nuances that developers use to make development easier. This segment explores PowerShell's nuances to make your code simpler and more maintainable.

7.5.1 Simplify Nested Statements

Nested statements are a fact of life for any developer, and managing them can be tricky. Nested statements also drift away from the goal of writing readable and maintainable code. So, what's the solution? It's a tricky answer that depends on the implementation, with code requirements dictating the outcome. This segment aims to provide options to manage nested statements better.

7.5.1.1 Option 1: Grouping Nested Conditions

Nested conditions use the following code structure:

```powershell
if ($Condition1) {
    if ($Condition2) {
        if ($Condition3) {
            # Do Something
        }
    }
}
```

Suppose the conditions are *singular*, meaning that there aren't other actions of dependencies on the condition. In that case, each of the conditions can be grouped by using parentheses and the `-and` operator:

```powershell
if (($Condition1) -and ($Condition2) -and ($Condition3)) {
    # Do Something
}
```

If the condition isn't *singular*, this makes it a lot trickier; however, it can be refactored with the use of the `elseif` statement. The example below demonstrates the use case where each condition isn't *singular* and contains an alternate execution path after each condition:

Example 42: A nested block of 'if' and 'else' statements

```powershell
if ($Condition1) {
    if ($Condition2) {
        if ($Condition3) {
            # Do Something
        }
        else {
            # [Third Condition] Do Something Else
        }
    }
    else {
        # [Second Condition] Do Something Else
    }
}
else {
    # [First Condition] Do Something Else
}
```

This can be refactored to:

Example 43: The same logic as Example 43, using a more linear structure

```
if (($Condition1) -and ($Condition2) -and ($Condition3)) {
    # Do Something
}
elseif (-not ($Condition1)) {
    # [First Condition] Do Something Else
}
elseif (($Condition1) -and (-not ($Condition2))) {
    # [Second Condition] Do Something Else
}
elseif (($Condition1) -and ($Condition2) -and (-not ($Condition3))) {
    # [Third Condition] Do Something Else
}
```

The nested code can be flattened by grouping the conditions, but notice (from the previous example) that it's not readable or maintainable. So how can this be taken further? First, questions need to be asked about the design of the script:

1. What requirements are needed for this code?
2. Can the conditions be solved further up the code stack?
3. Can the use of the pipeline resolve any issues?

In this example, the code has been optimized, and the conditions depend on each other. This presents two options:

1. Use the existing examples, or
2. Refactor the code by using implicit conditions.

The code can be refactored further by simplifying the conditions:

Example 44: Simplifying further by understanding the execution logic

```
if (($Condition1) -and ($Condition2) -and ($Condition3)) {
    # Do Something
}
elseif (-not ($Condition1)) {
    # [First Condition] Do Something Else
}
elseif (-not ($Condition2)) {
    # [Second Condition] Do Something Else
}
elseif (-not ($Condition3)) {
    # [Third Condition] Do Something Else
}
```

In the example, all irrelevant conditions are removed. In this case, the nested conditions validating the `$Condition1` and `$Condition2` are removed. Why? Within the `elseif` statement, each previous condition is implicitly `$true` *based on the previous condition*. The first condition is testing for success. If the first condition fails, either `$Condition1`, `$Condition2`, or `$Condition3` is `$false`. The subsequent `elseif` conditions are then *explicitly ordered* based on the nesting in the original example (`$Condition1` → `$Condition2` → `$Condition3`).

```
...
}
elseif (-not ($Condition1)) {
    # [First Condition] Do Something Else
}
elseif (($Condition1) -and (-not ($Condition2))) {
    # [Second Condition] Do Something Else
}
```

In the first `elseif` statement, the code tests if `$Condition1` is `$false`. In the second statement, the code tests if `$Condition1` is `$true` and condition2 is `$false`. However, the assumption is made:

> *In the first* `elseif` *statement, condition1 is being tested for* `$false`*, making it implicitly* `$true` *for subsequent statements.*

This removes the requirements to test the aforementioned conditions again since it's not `$true`.

There is a risk here that the outcomes are affected since they depend on the ordering within the `elseif` statement. To mitigate this, write unit tests to test for variations and document the ordering within the code.

7.5.1.2 Option 2: Simplifying Nested Loops

PowerShell loop statements can become unwieldy when multiple statements are nested, increasing the complexity of the code and associated unit tests with it. Loop statements *shouldn't be nested more than three times* without clear loop-controls (using `continue` and `break`).[16] [17] [18]

Each nested loop statement increases the number of loops, multiplying the number of possibilities.

Consider the following example:

Example 45: Nested looping constructs increase the number of operations exponentially

```
$Counter = 0
for ($i = 0; $i -ne 10; $i++) {
    for ($x = 0; $x -ne 10; $x++) {
        $Counter++
        Write-Host "$Counter"
    }
}
```

[16] Microsoft. (2014, May. 08). *PowerShell Looping: Advanced Break*. Microsoft Dev Blogs. [Online]. Available: https://devblogs.microsoft.com/scripting/powershell-looping-advanced-break/. [Accessed: May. 26, 2022].

[17] Microsoft. (2022, Mar. 03). *About Break (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_break. [Accessed: May. 26, 2022].

[18] Microsoft. (2022, Mar. 19). *About Continue (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_continue. [Accessed: May. 26, 2022].

```
1
2
3
...
98
99
100
```

The previous statement contains two nested statements that both count to 10. However, if an additional nested loop statement is added, it's now (10x10x10) 1000:

Example 46: Nested looping constructs increase the number of operations exponentially

```
$Counter = 0
for ($i = 0; $i -ne 10; $i++) {
    for ($x = 0; $x -ne 10; $x++) {
        for ($z = 0; $z -ne 10; $z++) {
            $Counter++
            Write-Host "$Counter"
        }
    }
}
```

```
1
2
3
...
998
999
1000
```

Sometimes nested looping statements are inevitable—a necessary evil; however, they can be refactored. These are the following rules to apply:

1. Cmdlet parameters. Check the cmdlet parameter object type to see if it supports arrays being parsed. For example, the following loop statement individually runs `Invoke-Command` on a number of remote computers:

Example 47: Running Invoke-Command many times inside a loop is inefficient

```
$Computers = Get-ADComputer -Filter *
foreach ($Computer in $Computers) {
    $params = @{
        ComputerName = $Computer
        ScriptBlock  = { Write-Host "Hello World" }
    }
    Invoke-Command @params
}
```

This code can be refactored since the `-ComputerName` Parameter accepts arrays and the `-AsJob` parameter is also present. The `-AsJob` parameter (in this use case) defers the execution of PowerShell for each computer into PowerShell jobs.[19] This simplifies and improves the performance at the same time since you can parse the entire array, removing the loop statement as well as deferring all the execution for each computer into a PowerShell job:

[19] Microsoft. (2022, Mar. 18). *About Jobs (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_jobs. [Accessed: May. 27, 2022].

Example 48: Running Invoke-Command once with an array of computers is more efficient

```
$Computers = Get-ADComputer -Filter *
$Params = @{
    ComputerName = $Computers
    ScriptBlock  = {
        Write-Host 'Hello-World'
    }
    AsJob        = $true
}
Invoke-Command @Params | Wait-Job
```

From the example above, you can use `Wait-Job` to pause execution until all the PowerShell jobs have been completed.

2. Use `Where-Object` to filter content. `Where-Object` can be used to perform the final condition by removing a nested statement. In the following example, a list of users is returned if one of the group's *distinguished names* (that they're a member of) contains 'test':

Example 49: Identifying group members using nested foreach constructs

```
# Define an Output Array
$AffectedUsers = @()
# Retrive all the Users from Active Directory
$Users = Get-ADUser -Filter *
# Iterate through each of the Users.
foreach ($user in $Users) {
    # Retrive the group membership for the user
    $UsersGroups = $_ | Get-ADPrincipalGroupMembership
    # Set ContainsTest to be implicitly false
    $ContainsTest = $false
    foreach ($userGroup in $UsersGroups) {
        if ($userGroup.DistinguishedName -match 'test') {
            # It contains test
            $ContainsTest = $true
            # Stop processing
            break
        }
    }
    if ($ContainsTest) { $AffectedUsers += $user }
}
```

This can be refactored by reducing the number of nested statements with the use of `Where-Object`:

Example 50: Using Where-Object in place of nested foreach statements

```
# Define an Output Array
$AffectedUsers = @()
# Retrive all the Users from Active Directory
$Users = Get-ADUser -Filter *
# Iterate through each of the Users.
foreach ($user in $Users) {
    # Retrive the group membership for the user
    $UsersGroups = $_ | Get-ADPrincipalGroupMembership
    # Use Where-Object
    [array]$ContainsTest = $UsersGroups | Where-Object {
        $_.DistinguishedName -match 'test'
    }
    if ($ContainsTest.Count -ne 0) { $AffectedUsers += $user }
}
```

Notice how the nested loop is removed? It's not, really. `Where-Object` has just simplified it.

Never nest `Where-Object` statements within `Where-Object` script blocks.

3. Consider the use case situations for a nested loop. Either remove the loop or abstract it away into a cmdlet. Nested statements can be filtered using `Where-Object`. However, when a left-hand and right-hand comparison is needed, simplify the list into an array and use the `Compare-Object` cmdlet. For example, consider the following comparison code comparing two byte arrays:

Example 51: Manually comparing two byte arrays from both directions

```
# Strings
$ByteArray1 = [System.Text.Encoding]::UTF8.GetBytes('Hello World1')
$ByteArray2 = [System.Text.Encoding]::UTF8.GetBytes('Hello World2')
# Note that $ByteArray1 and ByteArray2 are arrays.
# See: Output1 in 'Output'
#
# Output 1:
$ByteArray1 -join ' '
$ByteArray2 -join ' '
# While we can compare strings, for this demo
# we're assuming we're comparing byte strings.

# Compare the left
$LeftDifference = @()
for ($i = 0; $i -ne $ByteArray1.Count; $i++) {
    if ($ByteArray1[$i] -ne $ByteArray2[$i]) {
        $LeftDifference += [pscustomobject]@{
            Index      = $i
            Pos        = 'left'
            ValueLeft  = $ByteArray1[$i]
            ValueRight = $ByteArray2[$i]
        }
    }
}

# Compare the right
$RightDifference = @()
for ($i = 0; $i -ne $ByteArray2.Count; $i++) {
    if ($ByteArray1[$i] -ne $ByteArray2[$i]) {
        $RightDifference += [pscustomobject]@{
            Index      = $i
            Pos        = 'right'
            ValueLeft  = $ByteArray1[$i]
            ValueRight = $ByteArray2[$i]
        }
    }
}

#
# Output 2:
$LeftDifference
$RightDifference
```

```
#
# Output 1
72 101 108 108 111 32 87 111 114 108 100 49
72 101 108 108 111 32 87 111 114 108 100 50
#
# Output 2
Index Pos   ValueLeft ValueRight
----- ---   --------- ----------
   11 left         49         50
   11 right        49         50
```

This can be refactored using `Compare-Object` to simplify the code execution:

Example 52: Comparing two byte arrays using Compare-Object

```
# Strings
$ByteArray1 = [System.Text.Encoding]::UTF8.GetBytes('Hello World1')
$ByteArray2 = [System.Text.Encoding]::UTF8.GetBytes('Hello World2')
# While we can compare strings, for this demo
# we're assuming we're comparing byte strings.

Compare-Object -ReferenceObject $ByteArray1 -DifferenceObject $ByteArray2
```

```
InputObject SideIndicator
----------- -------------
         50 =>
         49 <=
```

7.5.2 Grouping Similar Code

Code that's grouped together improves the readability and maintainability of code. Consider the following PowerShell script; the code isn't grouped and is difficult to follow:

Example 53: Ungrouped code is difficult to read and maintain

```
# CSV format:
# "Source","Destination"

# Importing CSV
$CSVContent = Import-Csv -LiteralPath 'Example.csv'
# Filtering CSV with another Object
$Filtered = $CSVContent | Where-Object { $_.Destination -eq 'Temp' }
# Process some changes
$Temp = New-Item -ItemType Directory -Name (New-Guid)
$Filtered | ForEach-Object {
    Copy-Item -Path ($_.Source) -Destination $Temp
}
$CSVContent2 = Import-Csv -LiteralPath 'Example2.csv'
# Filtering CSV with another Object
$Filtered2 = $CSVContent2 | Where-Object { $_.Destination -eq 'Temp' }
# Process some changes
$Temp2 = New-Item -ItemType Directory -Name (New-Guid)
$Filtered2 | ForEach-Object {
    Copy-Item -Path ($_.Source) -Destination $Temp2
}
# Random Function
function Get-Something {
    Get-Random
}
# Send an Email Confirming the Changes
$Params = @{
    To      = 'helpdesk@contoso.com'
    From    = 'noreply@contoso.com'
    Subject = 'Report'
}
Send-MailMessage @Params
```

Below are guidelines describing the order of grouping that's used to refactor the code to be more readable and maintainable:

1. Group top-level code together. Top-level code means top-level script items such as parameters and global variables, classes, functions, pre/post execution, and the main code segment. The order of grouping is:

    1. **Parameters and Global Variables**: Defining top-level variables such as passwords or variables that cascade into functions.
    2. **Classes**: Classes need to be added before adding the functions to ensure there aren't any issues.
    3. **Functions**: Functions are added after classes and before any PowerShell statement execution.
    4. **Pre-Execution**: Any code that needs to run before the main code block. For instance: SQL connections, setup log files, setup HTTPS TLS versioning.
    5. **Main Code Block**: The main code block where the code is executed.
    6. **Post-Execution**: Any code that needs to run after the main code block. For instance: Closing SQL connections, closing log files.

2. Group related code by inserting blank (empty) lines. Adding blank lines to separate code provides 'physical' separation of code, similar to how paragraphs group common points or ideas.
3. Group common actionable items together. Everyday actionable items can be things that share the same objective or have the same/similar logic. It's essential when grouping not to break the logic flow. This approach works well with the Grouping, Sorting, and Filtering approach mentioned in the Expanding on the Pipeline section.

Let's refactor the previous example by:

- Grouping the top-level code.
- Grouping related code by adding blank lines.
- Grouping common actionable items together.

Example 54: Grouped code is much easier to read and maintain

```
# Random Function
function Get-Something {
    Get-Random
}

# (NOTE: A blank line is inserted above to separate groups)
# (NOTE: The CSV imports are the same so they can be grouped together.)
# Importing CSV
$CSVContent = Import-Csv -LiteralPath 'Example.csv'
$CSVContent2 = Import-Csv -LiteralPath 'Example2.csv'

# (NOTE: A blank line is inserted above to separate groups)
# (NOTE: The directory creations are the same so they can be grouped together.)
# Creating temporary directories
$Temp = New-Item -ItemType Directory -Name (New-Guid)
$Temp2 = New-Item -ItemType Directory -Name (New-Guid)

# (NOTE: A blank line is inserted above to separate groups)
# (NOTE: Where-Object calls are the same so they can be grouped together.)
# Filtering CSV with another Object
$Filtered = $CSVContent | Where-Object { $_.Destination -eq 'Temp' }
$Filtered2 = $CSVContent2 | Where-Object { $_.Destination -eq 'Temp' }

# (NOTE: A blank line is inserted above to separate groups)
# (NOTE: The following code is similar so it can be grouped together.)
# Process some changes
$Filtered | ForEach-Object {
    Copy-Item -Path ($_.Source) -Destination $Temp
}
# Process some other changes
$Filtered2 | ForEach-Object {
    Copy-Item -Path ($_.Source) -Destination $Temp2
}

# (NOTE: A blank line is inserted above to separate groups)
# Send an Email Confirming the Changes
$Params = @{
    To      = 'helpdesk@contoso.com'
    From    = 'noreply@contoso.com'
    Subject = 'Report'
}
Send-MailMessage @Params
```

By applying these guidelines, the code becomes more readable and maintainable.

7.5.3 Refactoring Comments and Documentation

Commenting PowerShell scripts documents your code, describing in detail to the reader what's going on and why. Issues arise when code is grouped into similar segments with no description or delimiter, making it difficult for the reader to identify the two segments.

Structure comments by adding 'headers' to the code by using the following guide:

```
#
# Header 1:
#

#
# Header 2:

# Comment:
```

The heading structure follows the same waterfall heading design in Microsoft Word. The three hashes denote top-level headers. Secondary headers are denoted by two hashes, with the blank hash above, and finally, the text body is a standard comment.

Consider the following PowerShell code:

Example 55: Code comments with a single heading level can be confusing as the importance of each comment is unknown

```
# Random Function
function Get-Something {
    Get-Random
}

# Importing CSV
$CSVContent = Import-Csv -LiteralPath 'Example.csv'
$CSVContent2 = Import-Csv -LiteralPath 'Example2.csv'

# Creating temporary directories
$Temp = New-Item -ItemType Directory -Name (New-Guid)
$Temp2 = New-Item -ItemType Directory -Name (New-Guid)

# Filtering CSV with another Object
$Filtered = $CSVContent | Where-Object { $_.Destination -eq 'Temp' }
$Filtered2 = $CSVContent2 | Where-Object { $_.Destination -eq 'Temp' }

# Process some changes
$Filtered | ForEach-Object {
    Copy-Item -Path ($_.Source) -Destination $Temp
}
# Process some other changes
$Filtered2 | ForEach-Object {
    Copy-Item -Path ($_.Source) -Destination $Temp2
}

# Send an Email Confirming the Changes
$Params = @{
    To      = 'helpdesk@contoso.com'
    From    = 'noreply@contoso.com'
    Subject = 'Report'
}
Send-MailMessage @Params
```

The PowerShell code is well-documented and grouped; however, it’s unclear. You can segment the groupings by adding multiple headers into the code:

Example 56: Using comments with three heading levels clarifies the priority of each comment

```
#
# Functions
#

# Random Function
function Get-Something {
    Get-Random
}

#
# Main: Process CSV Files and Send an Email
#

#
# Importing CSV
$CSVContent = Import-Csv -LiteralPath 'Example.csv'
$CSVContent2 = Import-Csv -LiteralPath 'Example2.csv'

#
# Creating temporary directories
$Temp = New-Item -ItemType Directory -Name (New-Guid)
$Temp2 = New-Item -ItemType Directory -Name (New-Guid)

#
# Filtering CSV with another Object
$Filtered = $CSVContent | Where-Object { $_.Destination -eq 'Temp' }
$Filtered2 = $CSVContent2 | Where-Object { $_.Destination -eq 'Temp' }

#
# Process Changes:

# Process some changes
$Filtered | ForEach-Object {
    Copy-Item -Path ($_.Source) -Destination $Temp
}
# Process some other changes
$Filtered2 | ForEach-Object {
    Copy-Item -Path ($_.Source) -Destination $Temp2
}

#
# Send an Email Confirming the Changes
$Params = @{
    To      = 'helpdesk@contoso.com'
    From    = 'noreply@contoso.com'
    Subject = 'Report'
}
Send-MailMessage @Params
```

Notice how easy the code is to follow with some headers describing the actions. Headers also apply to multiline comments as well, and body comments don't require a hash:

Example 57: Multilevel comment headings in multiline comments

```
<#
#
# Step 3: Processing Changes and Generating Report
#

#
# Import items of work.

Importing CSV
#>
Import-Csv -LiteralPath 'Example.csv'
```

It's essential to provide end user documentation of a script and/or *public* function using PowerShell's *comment-based help*. Within complex scripts, it's best to have the documentation at the top of the script or function, since it also assists code review, documentation, and code readability.

More information on PowerShell's comment-based help and `Get-Help` can be found at Microsoft Docs[20].

7.5.4 Using Code Regions

Syntax:

```
#region <name>
Get-Random
#endregion (optional)<name>
```

Code regions (region folding) are groupings of code that 'interact' (fold/collapse) within an IDE editor.[21] It provides a means of folding/collapsing groupings of code, aiding in the readability and maintainability of code. Regions are denoted by the `#region` comment, some code, and the `#endregion` comment. Regions can be nested as well, enabling finer control over groupings.

Example 58: Using code regions in PowerShell code

```
#region
Get-Random
#endregion

#region someName
Get-Random
#endregion

#region TopLevel
#region AnotherLevel
Get-Random
#endregion AnotherLevel
#endregion TopLevel
```

[20] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_comment_based_help

[21] Microsoft. (2014, Nov. 12). *Use Regions for PowerShell Comments*. Microsoft Dev Blogs. [Online]. Available: https://devblogs.microsoft.com/scripting/use-regions-for-powershell-comments/. [Accessed: May. 26, 2022].

It’s always recommended to be explicit and define region names to prevent possible IDE issues.

In the following example, regions are added to the code groupings:

Example 59: Regions work alongside comment heading levels and code groupings to aid reading

```
#
# Functions
#
#region Functions

# Random Function
#region Function-Something
function Get-Something {
    Get-Random
}
#endregion Function-Something

#endregion Functions
#
# Main: Process CSV Files and Send an Email
#
#region Main

#
# Importing CSV
$CSVContent = Import-Csv -LiteralPath 'Example.csv'
$CSVContent2 = Import-Csv -LiteralPath 'Example2.csv'

#
# Creating temporary directories
$Temp = New-Item -ItemType Directory -Name (New-Guid)
$Temp2 = New-Item -ItemType Directory -Name (New-Guid)

#
# Filtering CSV with another Object
$Filtered = $CSVContent | Where-Object { $_.Destination -eq 'Temp' }
$Filtered2 = $CSVContent2 | Where-Object { $_.Destination -eq 'Temp' }

#
# Process Changes:

# Process some changes
$Filtered | ForEach-Object {
    Copy-Item -Path ($_.Source) -Destination $Temp
}
# Process some other changes
$Filtered2 | ForEach-Object {
    Copy-Item -Path ($_.Source) -Destination $Temp2
}

#
# Send an Email Confirming the Changes
$Params = @{
    To      = 'helpdesk@contoso.com'
    From    = 'noreply@contoso.com'
    Subject = 'Report'
}
Send-MailMessage @Params
#endregion Main
```

Within a compatible IDE, these regions can be collapsed, hiding the blocks of code they contain:

Example 60: The code from Example 60 with collapsed regions as seen in an IDE

```
#
# Functions
#
#region Functions ···
#endregion Functions
#
# Main: Process CSV Files and Send an Email
#
#region Main ···
#endregion Main
```

While regions are handy, there can be too much of a good thing. As a rule of thumb, use regions to group ‘top-level’ items, such as functions, classes and large code-block groupings. Low-level regions don’t add anything and can add complexity to the code and comment structure.

7.5.5 Refactoring Logic Flow to be Implicitly `$True` or `$False`

Unstructured PowerShell function code traditionally follows a varying logic flow with no clear guidelines. Parameters are parsed into the function; the function performs a task and outputs the result. From the testability point of view, functions can behave as ‘black box’ entities that developers fear making changes to.

Junior PowerShell developers write functions based on an input/output methodology, focusing on the task and the needed output without consideration given to the logic flow.

Example 61: Code without a clear execution and logic flow is difficult to maintain

```
# Test the files to make sure that they exist
if ('SomeCSVPath', 'SomeCLIXMLPath', 'SomeOtherCLIXMLPath' | Test-Path) {
    # Import the File
    $CSVFile = Import-Csv -LiteralPath 'SomeCSVPath'
    # Import the CLIXML
    if ($someCondition) {
        if ($anotherCondition) {
            $CLIXML = Import-Clixml -LiteralPath 'SomeCLIXMLPath'
        }
        else {
            $CLIXML = Import-Clixml -LiteralPath 'SomeOtherCLIXMLPath'
        }
        Write-Host "Completed!"
        Write-Output $CLIXML
    }
    else {
        throw "Condition Failed. Stopping"
    }
}
```

The PowerShell code is functional; however, the logic flow of this code is difficult to follow. The code doesn’t follow a natural flow; the success output is inside a nested `if` statement. While this is a simple point, lots of developers make this mistake, and the effect is immediate, producing black box code that everyone (including the author) will struggle to reread in a few months. Logic should be a structured model similar to a waterfall, where inputs via parameters are parsed into the top, and the process executes and falls to the bottom. The advantage of using this design is that it's readable and maintainable three months from deployment. Why? People generally read a page from top to bottom when reading a book; that's how you should write code. It's that simple.

To do this, you need to use some techniques to refactor the code. These are:

1. Reducing nested statements
2. Inverting conditions
3. Using splatting[22]

Using these techniques, the code can be refactored to follow a waterfall design:

Example 62: The code from Example 62 with clearer execution and logic flows

```powershell
# Test if the Paths Exist. If not, return to the caller
if (-not ('SomeCSVPath', 'SomeCLIXMLPath', 'SomeOtherCLIXMLPath' | Test-Path)) {
    return
}

# Import the CSV File
$CSVFile = Import-Csv -LiteralPath 'SomeCSVPath'

# Invert $someCondition
if (-not ($someCondition)) {
    throw "Condition Failed. Stopping"
}

<#
We can use splatting to simplify the output here.
Multiple examples are used to demonstrate different
means of refactoring the code. These are:

1. Standard Execution (used in the example)
2. Subexpression (shown below)
3. Ternary operator (shown below)

#>
$Params = @{
    LiteralPath = 'SomeCLIXMLPath'
}

# Standard Execution
if (-not ($anotherCondition)) { $Params.LiteralPath = 'SomeOtherCLIXMLPath' }

# Splat it in!
Import-Clixml @Params
```

You can use subexpressions within hashtable constructs to select values dynamically:

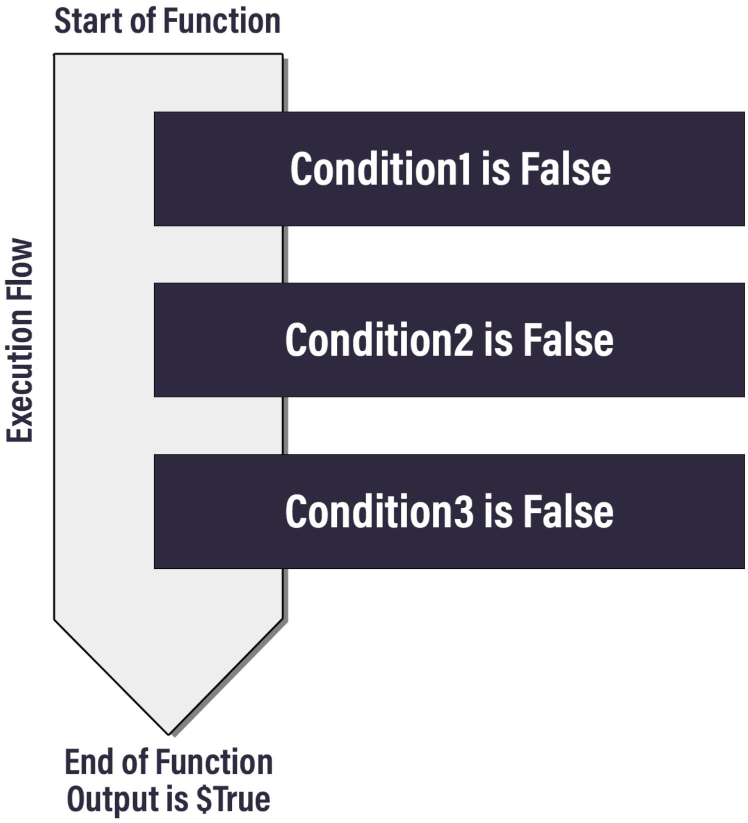
[22] Microsoft. (2022, Mar. 19). *About Splatting (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_splatting. [Accessed: May. 25, 2022].

```
$Params = @{
    LiteralPath = $(
        if ($anotherCondition) { 'SomeCLIXMLPath' }
        else { 'SomeOtherCLIXMLPath' }
    )
}
```

Other logic, such as PowerShell 7's ternary operator, is also usable:[23] [24]

```
$Params = @{
    LiteralPath = ($anotherCondition) ? 'SomeCLIXMLPath' : 'SomeOtherCLIXMLPath'
}
```

The preferred example in the demo is the ternary operator, since a singular value is returned from a condition. However, the logic is much easier to follow from the previous example. This waterfall design is *implicitly true*. By implicitly 'true', it means filtering out all the conditions that aren't required, cascading towards the result qualifying as 'true'.



Function Execution

[23] Microsoft. (2022, Mar. 19). *About Operators (Microsoft.PowerShell.Core) - Ternary operator*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_operators#ternary-operator–if-true–if-false. [Accessed: May. 26, 2022].

[24] Microsoft. (2022, Mar. 18). *About If (Microsoft.PowerShell.Core) - Using the ternary operator syntax*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_if#using-the-ternary-operator-syntax. [Accessed: Jun. 04, 2022].

The waterfall approach also applies to implicitly ‘false’ function flows. These flows invert the logic, where the bottom of the function is ‘false’, and the filtering conditions are true. An example would be to write a function to perform a number of file checks, which would qualify a ‘True’ result. After the cascade ends, the implicit result would be ‘false’.

Example 63: A function with an ‘implicitly false’ logic flow

```
function Invoke-Something {
    param($FilePath)

    #
    # If the File Extension is an '.exe'
    # Then return true.
    if ((Get-Item $FilePath).Extension -eq '.exe') { return $true }

    #
    # Load the file
    $Content = Get-Content $FilePath

    # If the first line of the file contains 'Value'
    # Then return true.
    if ($Content[0] -eq 'Value') { return $true }

    # If the last line of the file contains 'EndOfFile'
    # Then return true.
    if ($Content[-1] -eq 'EndOfFile') { return $true }

    # Return false
    return $false
}
```

Another example of using an implicitly ‘false’ approach in the wild is to compare this approach against network IP firewalls. As the packet is evaluated within the IP tables (firewall rules), it falls down through all the source/destination rules. If no rules are valid, the packet falls into a ‘global block’ (at the bottom), where packets are dropped. This approach makes it a lot easier to trace issues since they’re ‘allowlisting’ a limited set of inputs. Everything else is dropped.

7.6 Data Management

Data structure management within PowerShell is always a key consideration when dealing with scripts that store data. This segment explores the common data structures, how to use them and best practices when using them.

7.6.1 JSON

JSON (JavaScript Object Notation) is an open standard format that uses human-readable text to store and transmit object information. JSON natively encodes scalars (base object types) such as strings, integers, lists, arrays and associative arrays (dictionary or hashtable). JSON is commonly used in the HTTP body content type for *REST* APIs and also for application configuration files. To serialize and deserialize JSON, use the `ConvertTo-Json` and `ConvertFrom-Json` cmdlets. Please note these cmdlets don’t directly serialize or deserialize to a file, so they need to be piped from `Get-Content` or to `Set-Content`.

Example 64: Data can only be deserialized from or serialized to JSON strings natively

```powershell
#
# Read from a File
Get-Content -Path 'demo.json' | ConvertFrom-Json
#
# Write to a File
$PSObject | ConvertTo-Json | Set-Content -Path 'demo.json'
```

It's crucial to remember that when using `Invoke-RestMethod`, the request body needs to be serialized into JSON. The response is automatically deserialized into a `[PSCustomObject]` if the `Content-Type` header is set within the request header (by default, the content type implicitly set). `Invoke-WebRequest` doesn't deserialize the response body since it's used to interact with web pages or web services rather than explicit REST services. `ConvertTo-Json` can serialize all object types and `ConvertFrom-Json` only deserializes JSON into a `[PSCustomObject]`.

In PowerShell 6 and later, the `ConvertFrom-Json` cmdlet accepts the `-AsHashtable` parameter. This results in a hashtable structure of deserialized data instead of `[PSCustomObject]`.

Example 65: Serializing and deserializing JSON data, and data type limitations

```powershell
#
# Example 1: Serialize as JSON
$Object = [PSCustomObject]@{
    Property        = 'Value'
    AnotherProperty = 'AnotherValue'
}
# Store the output into a variable
$JSONString = $Object | ConvertTo-Json
$JSONString
#
# Example 2: Deserialize JSON
$JSONString | ConvertFrom-Json

# Example 3: Explore Object Type
$Process = Get-Process | Select-Object -First 1
$Process.GetType() | Select-Object Name, BaseType
#
# Example 4: Using the Process Variable, let's serialize it

$JSONString = $Process | ConvertTo-Json
$JSONString.Length
#
# Example 5: Deserialize $JSONString and check the Object Type
($JSONString | ConvertFrom-Json).GetType() | Select-Object Name, BaseType
```

```
# Example 1:
{
  "Property": "Value",
  "AnotherProperty": "AnotherValue"
}

# Example 2:
Property AnotherProperty
-------- ---------------
Value    AnotherValue

# Example 3:
Name    BaseType
----    --------
Process System.ComponentModel.Component

# Example 4:
5722

# Example 5:
Name           BaseType
----           ---------
PSCustomObject System.Object
```

7.6.2 YAML

YAML (Yet Another Markup Language) is a markup language that's used to store configuration for applications. It's used by configuration management solutions (Ansible and DSCDatum) to store complex machine configurations in a human-readable syntax. YAML natively encodes scalars (base object types) such as strings, integers, lists, arrays and associative arrays (dictionary or hashtable). It's simpler than JSON, using a Python-style indentation to represent nesting. It's also more compact, using square brackets to denote lists and curly braces for associative arrays. PowerShell has no native YAML serialization and deserialization (compared to JSON), so third-party modules are used. The module powershell-yaml[25] from the PowerShell Gallery is used in the following examples.

Remember: `Install-Module` is used to install PowerShell modules from the PowerShell Gallery.

YAML is serialized and deserialized by using `ConvertTo-Yaml` and `ConvertFrom-Yaml`:

[25] https://www.powershellgallery.com/packages/powershell-yaml

Example 66: Serializing and deserializing

```
#Requires -Modules "powershell-yaml"
#
# Example 1: Serialize a PowerShell Object to YAML
$Object = [PSCustomObject]@{
    Property        = 'Value'
    AnotherProperty = 'AnotherValue'
}
$Object | ConvertTo-Yaml
#
# Example 2: Deserialize a YAML string into a PowerShell Object
$Text = @'
Property: Value
AnotherProperty: AnotherValue
'@
$Deserialized = $Text | ConvertFrom-Yaml
$Deserialized
#
# Example 3: Show the PowerShell Object Type of a Deserialized YAML string.
$Deserialized.GetType() | Select-Object Name, BaseType
```

```
# Example 1:
Property: Value
AnotherProperty: AnotherValue

# Example 2:
Name                           Value
----                           -----
AnotherProperty                AnotherValue
Property                       Value

# Example 3:
Name      BaseType
----      --------
Hashtable System.Object
```

7.6.3 XML

XML (eXtensible Markup Language) is a markup language format for storing and transmitting object data. XML is designed to be ‘semi-human’ and machine-readable. XML has largely been superseded by newer markup languages (such as JSON and YAML). XML can be parsed natively within PowerShell. XML is loaded by reading the XML content as a string and then typecasting the string to the `[XML]` class.

Example 67: Deserializing XML into an XML object

```
$XMLString = @'
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">$projectname$</string>
</resources>
'@

# TypeCast the FileContents as an XML.
$XMLObject = [XML]$XMLString
$XMLObject
```

```
xml                            resources
---                            ---------
version="1.0" encoding="utf-8" resources
```

To serialize an XML document to file, use the `Save()` Method:

Example 68: Serializing PowerShell XML objects to an XML file

```
$FilePath = Join-Path $PWD 'demo.xml'
$XMLObject.Save($FilePath)

Get-Item $FilePath
```

```
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a---          20/02/2022   6:47 PM           118 demo.xml
```

You can access the raw serialized string using the `OuterXml` property:

Example 69: Retrieving the serialized string from an XML object

```
$XMLString = $XMLObject.OuterXml
$XMLString
```

```
<?xml version="1.0" encoding="utf-8"?><resources><string name="app_name">
$projectname$</string></resources>
```

XML's key advantage over newer markup languages is its native ability to use XPATH to search the XML dataset. There are two methods of searching XML, using the object methods and using `Select-Xml`:

1. Using `SelectSingleNode([string] $XPath)` or `SelectNodes([string] $XPath)` methods within the typecasted XML object.

Example 70: Searching XML objects using SelectNodes() and SelectSingleNode()

```
$XMLString = "
<note>
<to>Michael</to>
<from>George</from>
<type>Event</type>
<body>Meet you at the park.</body>
</note>
"

$XML = [XML]$XMLString

#
# Example 1: Use the SelectNodes method
$XML.SelectNodes('//note')

#
# Example 2: Use the SelectSingleNode method
$XML.SelectSingleNode('//to')
```

```
# Example 1:
to      from   type  body
--      ----   ----  ----
Michael George Event Meet you at the park.

# Example 2:
#text
-----
Michael
```

2. Using `Select-Xml` to find text in an XML string or document.

Example 71: Searching XML strings using Select-Xml

```
$XMLString = "
<note>
<to>Michael</to>
<from>George</from>
<type>Event</type>
<body>Meet you at the park.</body>
</note>
"

$Result = Select-Xml -Content $XMLString -XPath '//to'
#
# Example 1: Result object
$Result
#
# Example 2: Actual XML node
$Result.Node
```

```
# Example 1:
Node Path        Pattern
---- ----        -------
to   InputStream //to

# Example 2:
#text
-----
Michael
```

7.6.4 CSV

CSV (Comma-Separated Values) is a standard for storing simple data structures (containing rows and columns) using the comma ‘,’ as a delimiter. Other delimiter-separated values (DSV) are also common, and in PowerShell, the delimiter is adjustable. A drawback of CSV is that objects with nested objects can’t be serialized, limiting the scope of the data. For more complex data storage, CLIXML is preferred. In PowerShell, files (`Import-Csv` and `Export-Csv`) and strings (`ConvertTo-Csv` and `ConvertFrom-Csv`) are used to serialize and deserialize CSV and DSV data.

Example 72: Serializing to and deserializing from CSV and DSV

```
#
# Example 1: Serializing an Object with CSV
$ObjectList = @(
    [PSCustomObject]@{
        Property        = 'Value'
        AnotherProperty = 'Another Value'
    }
    [PSCustomObject]@{
        Property        = 'Second Value'
        AnotherProperty = 'Second Another Value'
    }
)
$ObjectList | ConvertTo-Csv
#
# Example 2: Serializing a List with CSV
$ObjectList = Get-Process | Select-Object -First 2 -Property E*, H*, Id, Name
$ObjectList | ConvertTo-Csv
#
# Example 3: Serializing a CSV file
$ObjectList | Export-Csv -Path 'List.csv'
#
# Example 4: Deserializing a CSV File
Import-Csv -Path 'List.csv' | Select-Object Id, Name, Handles
#
# Example 5: Deserializing a CSV String
$string = @'
"Property","AnotherProperty"
"Value","Another Value"
"Second Value","Second Another Value"
'@
$String | ConvertFrom-Csv
#
# Example 6: Serializing Using a Different Delimiter
$ObjectList = @(
    [PSCustomObject]@{
        Property        = 'Value'
        AnotherProperty = 'Another Value'
    }
    [PSCustomObject]@{
        Property        = 'Second Value'
        AnotherProperty = 'Second Another Value'
    }
)
$ObjectList | ConvertTo-Csv -Delimiter '>'
#
# Example 7: De-Serializing Using a Different Delimiter
$String = @'
"Property">"AnotherProperty"
"Value">"Another Value"
"Second Value">"Second Another Value"
'@
$String | ConvertFrom-Csv -Delimiter '>'
```

```
# Example 1:
"Property","AnotherProperty"
"Value","Another Value"
"Second Value","Second Another Value"

# Example 2:
"ExitCode","ExitTime","EnableRaisingEvents","Handles","Handle","HasExited", ...
        "HandleCount","Id","Name"
,,"False","88",,,"88","6996","AggregatorHost"
,,"False","550","1908","False","550","33056","ApplicationFrameHost"

# Example 3:
Id    Name                 Handles
--    ----                 -------
6996  AggregatorHost       88
33056 ApplicationFrameHost 550

# Example 4:
Property     AnotherProperty
--------     ---------------
Value        Another Value
Second Value Second Another Value

# Example 5:
"Property">"AnotherProperty"
"Value">"Another Value"
"Second Value">"Second Another Value"

# Example 6:
Property     AnotherProperty
--------     ---------------
Value        Another Value
Second Value Second Another Value
```

7.6.5 CLIXML

Did you know? CLIXML is the data type that’s used to transmit the PowerShell Session content when using PowerShell Remoting.

CLIXML (Common Language Infrastructure eXtensible Markup Language) is a subset of XML, where complex PowerShell objects are serialized into XML, attempting to preserve object types.

For complex objects that aren't a base type, PowerShell won't attempt to typecast them and they're placed in a `PSObject`.

Serializing PowerShell secure strings[26] as CLIXML can be problematic since it uses the Windows DPAPI (user-based) to encrypt the secure string. If another user or machine deserializes the CLIXML, the process fails.

To serialize and deserialize CLIXML, use `Import-Clixml` and `Export-Clixml`.

Example 73: Serializing and deserializing data to CLIXML

```
#
# Example 1: Serialize a PowerShell Object as CLIXML
$PSObject = [PSCustomObject]@{
    Property        = 'Value'
    AnotherProperty = 'AnotherValue'
}
$PSObject | Export-Clixml -Path 'SomePath.clixml'
#
# Example 2: Note the object type
$PSObject.GetType().Name
#
# Example 3: Deserialize a CLIXML file as an PowerShell Object.
$Object = Import-Clixml -Path 'SomePath.clixml'
$Object
#
# Example 4: Note the object type
$Object.GetType().Name
#
# Example 5: What does a CLIXML file look like?
Get-Content -Path 'SomePath.clixml'
```

```
# Example 1:
(no output)

# Example 2:
PSCustomObject

# Example 3:
Property AnotherProperty
-------- ---------------
Value    AnotherValue

# Example 4:
PSCustomObject
```

[26] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.security/convertto-securestring

```
# Example 5:
<Objs Version="1.1.0.1"
      xmlns="http://schemas.microsoft.com/powershell/2004/04">
  <Obj RefId="0">
    <TN RefId="0">
      <T>System.Management.Automation.PSCustomObject</T>
      <T>System.Object</T>
    </TN>
    <MS>
      <S N="Property">Value</S>
      <S N="AnotherProperty">AnotherValue</S>
    </MS>
  </Obj>
</Objs>
```

In the example listed, a `PSCustomObject` was serialized and deserialized, preserving the object type. In the following example, a complex object is used:

Example 74: Working with more complex data and CLIXML

```
#
# Example 1: Serialize an Object as CLIXML
$Processes = Get-Process | Select-Object -First 1
$Processes | Export-Clixml -Path 'SomePath.clixml'
#
# Example 2: Note the object type
$Processes.GetType().Name
#
# Example 3: Deserialize a CLIXML file as an PowerShell Object.
$CustomObject = Import-Clixml -Path 'SomePath.clixml'
$CustomObject
#
# Example 4: Note the object type
$CustomObject.GetType().Name
```

```
# Example 1:
(no output)

# Example 2:
Process

# Example 3:
 NPM(K)    PM(M)      WS(M)     CPU(s)      Id  SI ProcessName
 ------    -----      -----     ------      --  -- -----------
      0     0.00      39.09       0.29      89  78 node

# Example 4:
PSObject
```

In the example, the initial object type was `[Process]`; however, after exporting and importing from CLIXML, it's now a `[PSObject]`.

Remember, complex objects won't retain their object types after serialization/deserialization.

7.6.6 Best Practices for Data Management

The following best practices should be followed when dealing with data storage and transmission:

- Use known data structures such as XML, JSON, YAML, CSV, and CLIXML. Never create a markup language unless it's necessary.
- Try always to serialize/deserialize using out-of-the-box or native modules.
- Never create or adjust data structures by changing the serialized string.
- Use JSON and YAML to store simple human-readable data. Try not to use XML; JSON and YAML are better suited.
- Use CSV, CLIXML, and SQL for non human-readable data structures. Use CSV for simple data sheets, CLIXML for complex nested objects, and SQL for large complex object structures.
- Use PowerShell Secrets Management to store passwords and avoid exporting secure strings in CLIXML.

7.7 Further Reading

- Introduction to JSON—Microsoft Tech Community[27]
- About Quoting Rules—Microsoft Docs[28]
- Understanding Advanced Functions in PowerShell—Microsoft DevBlogs[29]
- Introduction to Advanced PowerShell Functions—Microsoft DevBlogs[30]
- About Functions Advanced Parameters—Microsoft Docs[31]
- Cmdlet Attributes—Microsoft Docs[32]
- Export-Clixml—Microsoft Docs[33]
- Import-Clixml—Microsoft Docs[34]
- Composite Formatting—Microsoft Docs[35]
- PowerShell Language Specification Chapter 6: Conversions—Microsoft Docs[36]

[27] https://techcommunity.microsoft.com/t5/microsoft-365-pnp-blog/introduction-to-json/ba-p/2049369
[28] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_quoting_rules
[29] https://devblogs.microsoft.com/scripting/understanding-advanced-functions-in-powershell/
[30] https://devblogs.microsoft.com/scripting/introduction-to-advanced-powershell-functions/
[31] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_functions_advanced_parameters
[32] https://learn.microsoft.com/en-us/powershell/scripting/developer/cmdlet/cmdlet-attributes
[33] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/export-clixml
[34] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/import-clixml
[35] https://learn.microsoft.com/en-us/dotnet/standard/base-types/composite-formatting
[36] https://learn.microsoft.com/en-us/powershell/scripting/lang-spec/chapter-06

8. Advanced Conditions

PowerShell has several advanced conditions available to you for refactoring your code. You'll use familiar conditional operators in most of your scripts, but there are more advanced features you may not have explored in depth. This chapter will explore the following topics:

1. Case Sensitive Operators
2. Switch Statement
3. Conversion Operators
4. Type Comparison using `-is` and `-isnot`
5. Typecasting using `-as`
6. Bitwise Operators
7. `-like` and `-notlike`
8. `-match` and `-notmatch`
9. `-contains` and `-in`
10. `-replace`
11. Ternary Operators
12. Null-Coalescing Operator
13. Null-Coalescing Assignment Operator
14. Null-Conditional Operators
15. Loop Labels
16. PowerShell Operator Precedence

8.1 Case Sensitive Operators

PowerShell is *case-insensitive* by default, which influences the behavior of most common comparison operators.

Example 1: PowerShell operators are case-insensive by default

```
# Example 1: Using the -eq Operator
'test' -eq 'TEST'

# Example 2: Using the -in Operator
'test' -in 'TEST','VALUE'
```

```
# Example 1:
True

# Example 2:
True
```

PowerShell also features case-sensitive operators with names having a ‘c’ prefix.

Example 2: Case-sensitive operator equivalents

```
# Example 1: Using the -ceq Operator
'test' -ceq 'TEST'

# Example 2: Using the -cin Operator
'test' -cin 'TEST','VALUE'
```

```
# Example 1:
False

# Example 2:
False
```

The following table lists each case-sensitive operator, its description, and an example:[1]

Operator	Description	Example
`-ceq`	Case Sensitive Equal	`'VALUE' -ceq 'value'`
`-cne`	Case Sensitive Not Equal	`'VALUE' -cne 'value'`
`-clike`	Case Sensitive Like	`'VALUE' -clike 'value*'`
`-cnotlike`	Case Sensitive Not Like	`'VALUE' -cnotlike 'value*'`
`-ccontains`	Case Sensitive Contains	`'VALUE','Value' -ccontains 'value'`
`-cnotcontains`	Case Sensitive Not Contains	`'VALUE','Value' -cnotcontains 'value'`
`-cin`	Case Sensitive In	`'value' -cin 'value','Value'`
`-cnotin`	Case Sensitive Not In	`'value' -cnotin 'value','Value'`
`-csplit`	Case Sensitive Split	`'valueVALUEvalue' -csplit 'VALUE'`
`-cmatch`	Case Sensitive Match	`'VALUE' -cmatch 'value'`
`-cnotmatch`	Case Sensitive Not Match	`'VALUE' -cnotmatch 'value'`
`-cgt`	Case Sensitive Greater Than	`5 -cgt 2`
`-clt`	Case Sensitive Less Than	`1 -clt 2`
`-cge`	Case Sensitive Greater Than or Equal To	`2 -ceg 2`
`-cle`	Case Sensitive Less Than or Equal To	`2 -cle 2`

[1] Microsoft. (2022, Mar. 19). *About Comparison Operators (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_comparison_operators. [Accessed: Mar. 30, 2022].

PowerShell rarely uses explicit case-insensitive operators since the language supports this mode by default. Explicit case-insensitive operators have names prefixed with an ‘i’, similar to case-sensitive operators.[2]

Example 3: Explicit case-insensitive operators are aliases of the standard ones

```
# Example 1:
# Note: These two statements are the same.
# That's because they are!

# Explicit Case-Insensitive Matching
'value' -ieq 'VALUE'
# Implicit Case-Insensitive Matching
'value' -eq 'VALUE'
```

```
# Example 1:
True
True
```

8.2 Using the Switch Statement

Syntax:

```
switch [-Regex|-Wildcard|-Exact][-CaseSensitive] (<value>)
{
    "string"|number|variable|{ expression } { statementlist }
    default { statementlist }
}
```

The `switch` statement is an operator used to test a value against multiple conditions.[3]

Example 4: Using the switch statement for multiple conditions

```
switch (3) {
    1 { 'one' }
    2 { 'two' }
    3 { 'three' }
    default { 'Something else' }
}
```

```
three
```

The `switch` statement accepts optional parameters (`-Regex`, `-Wildcard`, `-Exact` and `-CaseSensitive`) that are used with two parameter sets. The `-Regex`, `-Wildcard` and `-Exact` parameters are mutually exclusive, controlling `[string]"string"` matching behavior. The optional `-CaseSensitive` parameter enables case-sensitive matching.

[2] Microsoft. (2022, Mar. 19). *About Comparison Operators (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_comparison_operators. [Accessed: Mar. 30, 2022].

[3] Microsoft. (2022, Mar. 19). *About Switch (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_switch. [Accessed: Mar. 30, 2022].

8.2.1 Using `-Regex`

The `-Regex` parameter enables string matching against regular expressions.

Example 5: Using the switch statement in regex mode

```
# With -Regex:

$value = 'value'
switch -Regex ($value) {
    # Match strings that contain exactly four characters
    '^\w{4}$' {
        'Skipped'
    }
    # Match strings that contain numerals
    '^[0-9]*$' {
        'Also Skipped'
    }
    # Match any string that contains alphanumeric characters.
    '^\w*$' {
        'Match'
    }
}
```

```
Match
```

8.2.2 Using `-Wildcard`

The `-Wildcard` parameter enables non-literal string matching within the condition.

Example 6: Using the switch statement in wildcard mode

```
# With -Wildcard:

# Value contains white space on each side
$value = ' value '
switch -Wildcard ($value) {
    '*value*' {
        'This matches'
    }
    '*valu*' {
        'This also matches'
    }
    'value*' {
        'This does not match'
    }
}
$output
```

```
This matches
This also matches
```

8.2.3 Using `-Exact`

`-Exact` is used to enable case-insensitive string matching, character-for-character.[4] This is the default behavior when all other parameters aren't present.

Example 7: The -Exact parameter causes the default - literal comparison

```
# With -Exact:

$value = 'value'
switch -Exact ($value) {
    'vAlue' {
        'Not skipped'
    }
    'Value' {
        'Also not Skipped'
    }
    'value' {
        'Match'
    }
}
```

```
Not skipped
Also not Skipped
Match
```

8.2.4 Using `-CaseSensitive`

The `-CaseSensitive` parameter enables case-sensitive matching behavior (applies to `[String]`). If an exact string match isn't matched, it's ignored.

Example 8: Using switch in case-sensitive mode

```
# Example 1: Without -CaseSenitive:

$value = 'value'
switch ($value) {
    'vAlue' {
        'Wrong case'
    }
    'Value' {
        'Also wrong case'
    }
    'value' {
        'Match'
    }
}
```

[4] Microsoft. (2022, Mar. 19). *About Switch (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_switch. [Accessed: Mar. 30, 2022].

```

# Example 2: With -CaseSenitive:
$value = 'value'
switch -CaseSensitive ($value) {
    'vAlue' {
        'Wrong case'
    }
    'Value' {
        'Also wrong case'
    }
    'value' {
        'Match'
    }
}
```

```
# Example 1:
Wrong case
Also wrong case
Match

# Example 2:
Match
```

8.2.5 Using PowerShell Expressions for Matching

When more complex matching is required, the switch statement can match expressions with a syntax similar to the `Where-Object` cmdlet. Expressions wrapped with curly-braces (`[Scriptblock]{}`) precede the statement list and must return either `$true` or `$false` (`$null` is treated as `$false`). `$_` is used inside an expression to reference the current `$PSItem`.[5]

Example 9: Using expression cases with switch

```
# With Expressions:

# Value contains white space on each side
$value = ' value '
switch ($value) {
    # Trim both the start and ending white space and perform a match
    { $_.Trim() -eq 'value' } {
        'This matches'
    }
    # Perform string manipulation and test
    { ($value.Substring(3) -replace 'PowerShell') -eq $_ } {
        'This does not match'
    }
    # This condition is always $true
    { $true } {
        'This also matches'
    }
    # This one is always $false
    { $false } {
        'This does not match'
    }
    # This one is always $false, too
```

[5] Microsoft. (2022, Mar. 19). *About Switch (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_switch. [Accessed: Mar. 30, 2022].

```
    { $null } {
        'This does not match'
    }
}
```

```
This matches
This also matches
```

8.2.6 `Default`

The `default` statement is an optional reserved statement used when all other conditions aren't met.[6]

Example 10: Using the default statement with switch

```
# Example 1: All other conditions will fail.
$string = 'value'
switch -CaseSensitive ($string) {
    'string' {
        'Skipped'
    }
    'another string' {
        'Also Skipped'
    }
    default {
        'default'
    }
}

# Example 2: The default statement is included, however won't be called.
# One of the conditions will succeed.
$string = 'value'
switch -CaseSensitive ($string) {
    'value' {
        'success'
    }
    'another string' {
        'Also Skipped'
    }
    default {
        'default'
    }
}
```

[6] Microsoft. (2022, Mar. 19). *About Switch (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_switch. [Accessed: Mar. 30, 2022].

```
# Example 1:
default

# Example 2:
success
```

8.2.7 Parsing Lists and Arrays

Under the hood, the `switch` statement functions as a loop statement, iterating through each item within a list.

Example 11: The switch statement behaves like a loop

```
# Example 1: Parse an array
$values = 'one','two','three','four'
switch ($values) {
    'one' { 'this is one' }
    'two' { 'this is two' }
    'three' { 'this is three' }
    default { 'default' }
}

# Example 2: Parse an array with loop control 'break' in the statement
$values = 'one','two','three','four'
switch ($values) {
    'one' { 'this is one' }
    'two' { break; }
    'three' { 'this is three' }
    default { 'default' }
}

# Example 3: Parse an array with loop control 'continue' in the statement
$values = 'one','two','three','four'
switch ($values) {
    'one' { 'this is one' }
    'two' { continue; }
    'three' { 'this is three' }
    default { 'default' }
}
```

```
# Example 1:
this is one
this is two
this is three
default

# Example 2:
this is one

# Example 3:
this is one
this is three
default
```

Here we replace the `switch` statement with equivalent logic using `foreach` and `if` statements:

Example 12: An equivalent foreach loop to the switch statement in Example 11

```
# While an explicit array cast is not required,
# it's used to associate a singular array item.
$strings = @('value')
# Iterate through each of the array items
foreach ($string in $strings) {
    #
    # Switch Conditions
    if ($string -eq 'value') {
        'matched'
    }
    if ($string -eq 'value2') {
        'not matched'
    }
    #
    # Default Statement
    # Note: How the "default" statement is structured.
    # The conditional logic explicitly tests all previous conditions.
    # and if $null, then is considered true.
    if (($string -ne 'value') -and ($string -ne 'value2')) {
        'default'
    }
}
```

```
matched
```

Loop control statements are applied to control the outcome of the execution. Since `switch` functions the same as a loop statement, `break` and `continue` will change the behavior. The `break` statement exits the program loop, the `switch` statement in this case, skipping all future items parsed into the statement.[7] The `continue` statement will stop and jump to the top of the innermost loop, processing the next item in the switch statement.

Example 13: Using the break and continue statements with switch

```
# Example 1: Use of a break statement in a singular array
$string = 'value'
switch ($string) {
    'value' {
        'This matches'
        # Add the break statement
        break
    }
    # Note that the matching condition is the same as
    # the previous condition. Without the break statement,
    # this wouldn't be skipped.
    'value' {
        'This also matches'
    }
    'values' {
        'This does not match'
    }
}

# Example 2: Use of a continue statement in a singular array
```

[7] Microsoft. (2022, Mar. 19). *PowerShell 101: Chapter 6 - Flow control - Break, Continue, and Return*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/scripting/learn/ps101/06-flow-control#break-continue-and-return. [Accessed: Mar. 31, 2022].

```
#            This example is the same as the previous example, except
#            replaces 'break' with 'continue'.
#            Note how the statement executes the same with a singular array.

$string = 'value'
switch ($string) {
    'value' {
        'This matches'
        # This time add the continue statement
        continue
    }
    # Note that the matching condition is the same as
    # the previous condition. Without the continue statement,
    # this wouldn't be skipped.
    'value' {
        'This also matches'
    }
    'values' {
        'This does not match'
    }
}

# Example 3: Use of a break statement in an array
#            This is the same execution path in Example 1,
#            however, notice how the break statement
#            stops matching with the current and future iterations.

$string = 'value','value'
switch ($string) {
    'value' {
        'This matches'
        # Add the break statement
        break
    }
    # Note that the matching condition is the same as
    # the previous condition. Without the break statement,
    # this wouldn't be skipped.
    'value' {
        'This also matches'
    }
    'values' {
        'This does not match'
    }
}

# Example 4: Use of a continue statement in an array
#            This is the same execution path in Example 2,
#            however notice how the continue statement
#            stops matching with the current iteration, but
#            continues with future iterations.

$string = 'value','value'
switch ($string) {
    'value' {
        'This matches'
        # This time add the continue statement
        continue
    }
    # Note that matching condition is the same as
    # the previous condition. Without the break statement,
    # this wouldn't be skipped.
    'value' {
        'This also matches'
    }
    'values' {
        'This does not match'
    }
}
```

```
# Example 1:
This matches

# Example 2:
This matches

# Example 3:
This matches

# Example 4:
This matches
This matches
```

The `switch` statement can match on different object-types. In the following example, different object-types are tested from an array:

Example 14: Using expressions with a switch statement to change behavior based on type

```
# Parsing an Array

$values = @(
    # HashTable
    @{
        Key = 'HashTable Value'
    },
    # PSCustomObject
    [PSCustomObject]@{
        FirstProperty = 'Property Value'
        SecondProperty = 'Some other property value'
    }
    # DateTime
    ([datetime]::Now)
)

switch ($values) {
    { $_ -is [Hashtable] } {
        $_.Key
    }
    { $_ -is [PSCustomObject] } {
        $value = $_
        $_ | Get-Member -MemberType NoteProperty | ForEach-Object {
            $value."$($_.Name)"
        }
    }
    { $_ -is [DateTime] } {
        $_.Date.ToString()
    }
}
```

```
HashTable Value
Property Value
Some other property value
11/10/2021 12:00:00 AM
```

8.3 Type Comparison and Conversion Operators: `-is`, `-isnot` and `-as`

8.3.1 Using `-is` and `-isnot`

Syntax: `<object> -is [type]`.

The `-is` operator is used to test .NET object instance types, returning `$true` for a type match, and `$false` otherwise. Instead of using `Get-Member` or the `GetType()` method, this provides a cleaner, more robust solution.[8]

Example 15: Testing for type with the -is operator

```
# Example 1: Simple String Test with [string] type
$string = 'this is a string'
$string.GetType().Name
$string -is [string]


# Example 2: Simple String Test with [int] type
$string = 'this is a string'
$string.GetType().Name
$string -is [int]


# Example 3: String Array Test with [string[]]
$arr = @(
    'this is a string'
    'this is also a string'
    'this is some other string'
)
$arr -is [string[]]


# Example 4: String Array Test with [array] type
$arr = @(
    'this is a string'
    'this is also a string'
    'this is some other string'
)
$arr -is [array]


# Example 5: String Array Test with Explicit Cast
[string[]]$arr = @(
    'this is a string'
    'this is also a string'
    'this is some other string'
```

[8] Microsoft. (2022, Mar. 19). *About Comparison Operators (Microsoft.PowerShell.Core) - Type comparison*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_comparison_operators#type-comparison. [Accessed: Mar. 31, 2022].

```
)
$arr -is [string[]]
```

```
# Example 1: Simple String Test with [string] type
String
True


# Example 2: Simple String Test with [int] type
String
False


# Example 3: String Array Test with [string[]] type
False


# Example 4: String Array Test with [array] type
True


# Example 5: String Array Test with Explicit Cast
True
```

The `-isnot` operator is the inverse to `-is`. It's used to test that the object instance *is not* of the specified type.[9]

Example 16: Excluding types with the -isnot operator

```
# Example 1: Simple String Test with [string] type
$string = 'this is a string'
$string.GetType().Name
$string -isnot [string]


# Example 2: Simple String Test with [int] type
$string = 'this is a string'
$string.GetType().Name
$string -isnot [int]
```

```
# Example 1: Simple String Test with [string] type
String
False

# Example 2: Simple String Test with [int] type
String
True
```

[9] Microsoft. (2022, Mar. 19). *About Comparison Operators (Microsoft.PowerShell.Core) - Type comparison.* Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_comparison_operators#type-comparison. [Accessed: Mar. 31, 2022].

8.4 Using -as to Typecast Safely

The -as operator is used to typecast a .NET object to another type within PowerShell safely.[10] If successful, it will return the converted object. Otherwise, it will return $null. Both terminating and non-terminating errors are suppressed when using -as. Syntax:

```
$Object -as [NewObjectType]
```

In both examples below, the output is the same whether using explicit typecasting or the -as operator:

Example 17: Two ways to cast an object to a type

```
# Example 1: Using -as for type conversion to cast a string to a URI
"https://example.com" -as [URI] | Format-Table

# Example 2: Using typecasting to cast a string to a URI
[URI]"https://example.com" | Format-Table
```

```
# Example 1: Using -as for type conversion

AbsolutePath AbsoluteUri          LocalPath Authority   HostNameType
------------ -----------          --------- ---------   ------------
/            https://example.com/ /         example.com          Dns

# Example 2: Using typecasting

AbsolutePath AbsoluteUri          LocalPath Authority   HostNameType
------------ -----------          --------- ---------   ------------
/            https://example.com/ /         example.com          Dns
```

In the examples below, "String" can't be typecast to a [DateTime], but using -as suppresses the error.

Example 18: Safe typecasting with -as

```
# Example 1: Typecasting "String" to [DateTime] using -as
"String" -as [DateTime]
$null -eq ("String" -as [DateTime])

# Example 2: Explicit casting [String] to [DateTime]
[DateTime]"String"
```

[10] Microsoft. (2022, Mar. 19). *About Type Operators (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_type_operators. [Accessed: Mar. 31, 2022].

```
# Example 1: Typecasting "String" to [DateTime] using -as
True

# Example 2: Explicit casting [String] to [DateTime]

InvalidArgument: Cannot convert value "String" to type "System.DateTime". Error:
"The string 'String' was not recognized as a valid DateTime.
There is an unknown word starting at index '0'."
```

8.5 Bitwise Operators (`-band`, `-bor`, `-bxor`, `-bnot`, `-shl` and `-shr`)

Bitwise operators enable bit manipulation and comparison of binary numbers (similar to logic gates).[11] Bitwise operators are commonly used with enums to represent a selection of flags instead of a single value.

8.5.1 What is an `Enum`?

The Enumeration type (`Enum`) is a class used to represent a group of constants.[12] [13] In the example below, an `Enum` called 'Colors' is created with the following properties:

Example 19: Creating enumerations with Add-Type

```
# Example 1: Create an Enum and list it's properties.

# Create an Enum in PowerShell Called Colors
Add-Type -TypeDefinition @"
   public enum Colors
   {
      Red,
      Green,
      Blue,
      Yellow,
      Black,
      White
   }
"@

# Example 1:

# List the properties of the enum.
[Colors].GetEnumNames()


# Example 2:

# List the combinations:
foreach ($val in [Enum]::GetValues([Colors])) {
```

[11] Microsoft. (2022, Mar. 19). *About Arithmetic Operators (Microsoft.PowerShell.Core) - Bitwise operators*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_arithmetic_operators#bitwise-operators. [Accessed: Mar. 31, 2022].

[12] Microsoft. (2015, Aug. 27). *Working with Enums in PowerShell 5*. Microsoft Dev Blogs. [Online]. Available: https://devblogs.microsoft.com/scripting/working-with-enums-in-powershell-5/. [Accessed: Mar. 31, 2022].

[13] Microsoft. (2022, Jan. 25). *Enumeration types (C# reference)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/enum. [Accessed: Mar. 31, 2022].

```
    "{0} {1}" -f [int]$val, $val
}

# Example 3:

# Casting enums as integer's and strings
[Colors]'Green'
[Colors]'Red'
[Colors]0 # Red
```

```
# Example 1:
Red
Green
Blue
Yellow
Black
White

# Example 2:
0 Red
1 Green
2 Blue
3 Yellow
4 Black
5 White

# Example 3:
Green
Red
Red
```

Basic enums can only set a *singular value* within it (`[Enum]::Green`), which is fine for constant use cases. Ideally, from the example above, multiple colors should be allowed to be selected. We can change the functionality by using a ‘flag enum’ (or *bitflag*). Within the flag enum, item properties must use powers of two (for example 1,2,4,8,16,...) to work with bitwise operators.

Example 20: Creating bitflag enums with Add-Type

```
Add-Type -TypeDefinition @"
   [System.Flags]
   public enum Colors
   {
      Red = 1,
      Green = 2,
      Blue = 4,
      Yellow = 8,
      Black = 16,
      White = 32,
      // You must use powers of 2 (for example: 1,2,4,6,8,...)
   }
"@
# Same as the previous example, print Every Combination of the color up to 16
# Please note that the upper limit is 63.
for ($i = 0; $i -le 16; $i++) {
    "{0} {1}" -f $i, [Colors]$I
}
```

```
0 0
1 Red
2 Green
3 Red, Green
4 Blue
5 Red, Blue
6 Green, Blue
7 Red, Green, Blue
8 Yellow
9 Red, Yellow
10 Green, Yellow
11 Red, Green, Yellow
12 Blue, Yellow
13 Red, Blue, Yellow
14 Green, Blue, Yellow
15 Red, Green, Blue, Yellow
16 Black
```

Since version 5.0, PowerShell supports the `enum` statement, enabling enums to be declared natively without `Add-Type`.[14] Syntax:

```
enum <enum-name> {
    <label> [= <int-value>]
}
```

The `enum-name`, follows directly after the `enum` statement describing the enum type, followed by the labels. Use the `[Flags()]` attribute to create flag enums, as with `Add-Type`. Note in the example below how no commas are required at the end of each of the labels. In the example below, the Colors enum used previously will be updated:

Example 21: Creating flag enums with the enum statement

```
[Flags()]
enum Colors
{
    Red = 1
    Green = 2
    Blue = 4
    Yellow = 8
    Black = 16
    White = 32
}
```

Unlike enums created with `Add-Type`, you can update native enums with new values.

Enum Reference

For more information, refer to the about Enum[15] page at Microsoft Docs.

[14] Microsoft. (2021, Sep. 28). *About Enum.* Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_enum. [Accessed: Mar. 31, 2022].

[15] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_enum

8.5.2 Base-2 vs. Base-10 (Binary vs. Decimal)

Binary (base-2) is a numerical system that uses two values to represent `$true` or `$false` ("on" or "off"). Each digit in a binary number must be either '0' or '1' and represents a power of 2. You're probably much more familiar with the decimal (base-10) system where each digit is one of '0', '1', '2', '3', '4', '5', '6', '7', '8', and '9'. Each digit in a decimal number represents a power of 10.

For example:

- Base-10 [*1,10,100,1000*]: Each position in this block increments the power of 10 (10^0, 10^1, 10^2, and 10^3 in this case). Note: Any number raised to the zeroth power equals one.
- Base-2 [*1,2,4,8,16,32,64,128,256,512,1024*]: Each position in this block increments the power of 2 (2^0, 2^1, 2^2, 2^3, ... 2^{10}). One disadvantage of base-2 is that the sequence of digits needed to represent a number is usually longer than in base-10. Within base-2, each digit is called a 'bit', and a grouping of eight bits is called a 'byte'. Binary numbers are typically formatted into groups of 4 digits called 'nibbles', making them easier to read. Bit groups having less than 4 bits are prefix-padded with zeros.

  For example: The 7-digit binary number 1000001 is formatted as 2 nibbles: 0100 0001.

The *Least Significant Bit* (LSB) is the position that represents the binary one's place of an integer. The *Most Significant Bit* (MSB) is the "left-most" binary digit available for an integer type. For example:

MSB							LSB
0	1	0	0	0	0	0	1

Below are two tables representing the 6 least significant places for base-10 and base-2 (in decimal):

base-10:

100000	10000	1000	100	10	1

base-2:

32	16	8	4	2	1

For example, the decimal number "One-Thousand and Forty-Two" is represented in base-10 as:

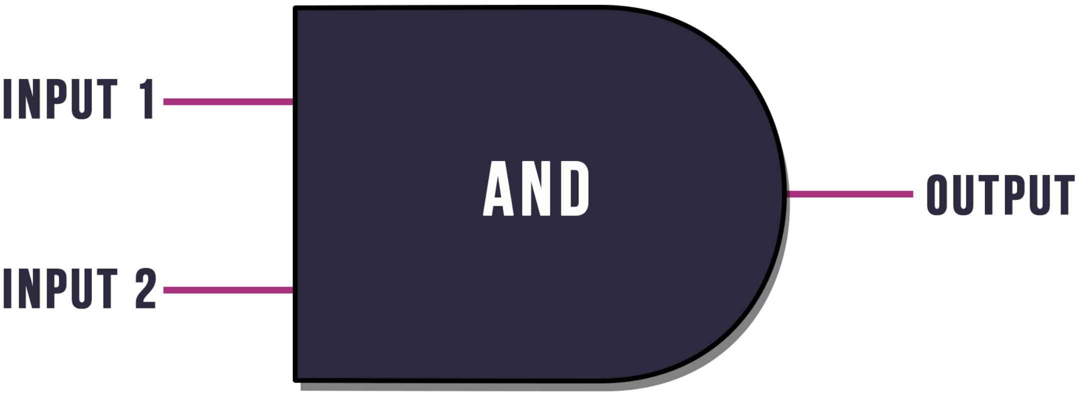
1000	100	10	1
1	0	4	2

Notice that this same value has a longer binary representation:

1024	512	256	128	64	32	16	8	4	2	1
1	0	0	0	0	0	1	0	0	1	0

This same value is represented: 1042 decimal = 0100 0001 0010 binary.

8.5.3 The AND Logic Gate



ANDLogicGate

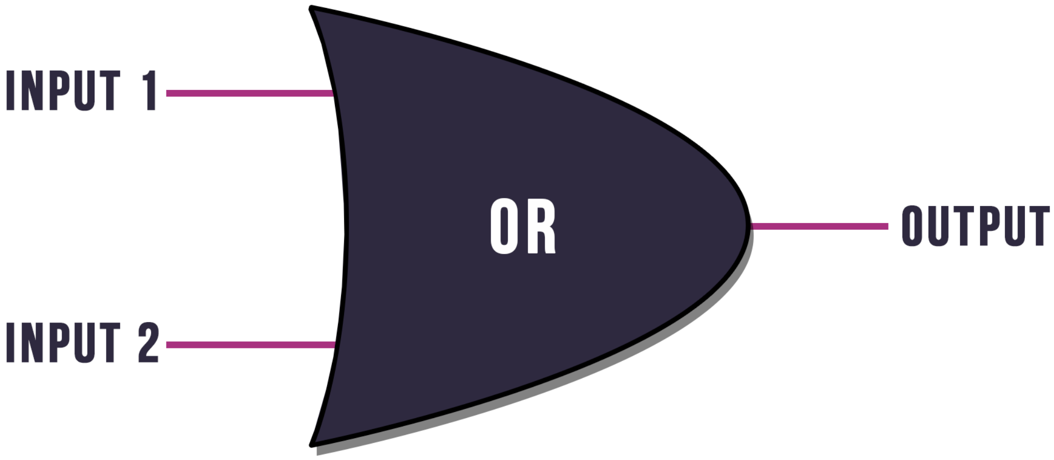
The AND logic gate requires both inputs to be '1' (`$true`) to output '1'. All other combinations will be '0' (`$false`).

Input 1	Input 2	Output
1	0	0
0	1	0
0	0	0
1	1	1

For example:

```
1001 - 9
0101 - 5 (AND)
----
0001 - 1
```

8.5.4 The OR Logic Gate



ORLogicGate

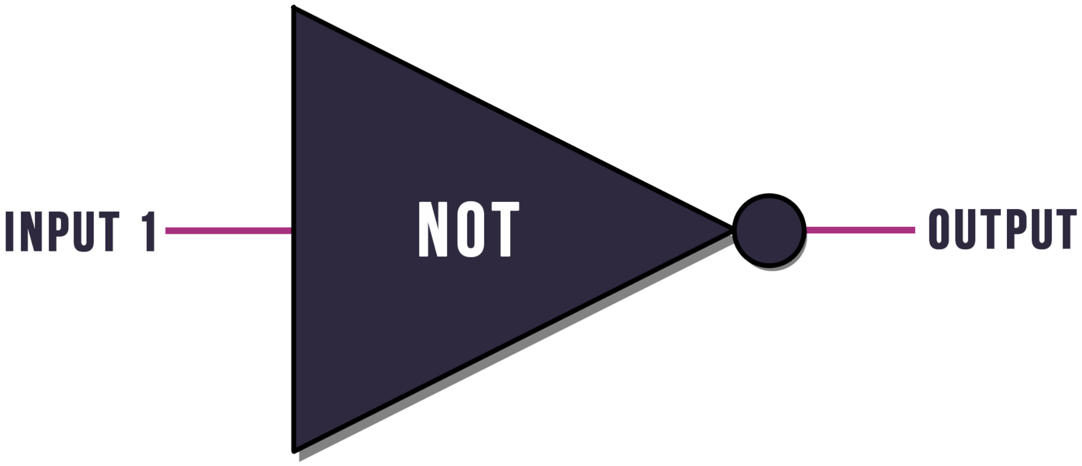
The OR logic gate requires only one input to be ‘1’ (`$true`) to output ‘1’.

Input 1	Input 2	Output
1	0	1
0	1	1
0	0	0
1	1	1

For example:

```
1001 - 9
0101 - 5 (OR)
----
1101 - 13
```

8.5.5 The NOT Logic Gate



NOTLogicGate

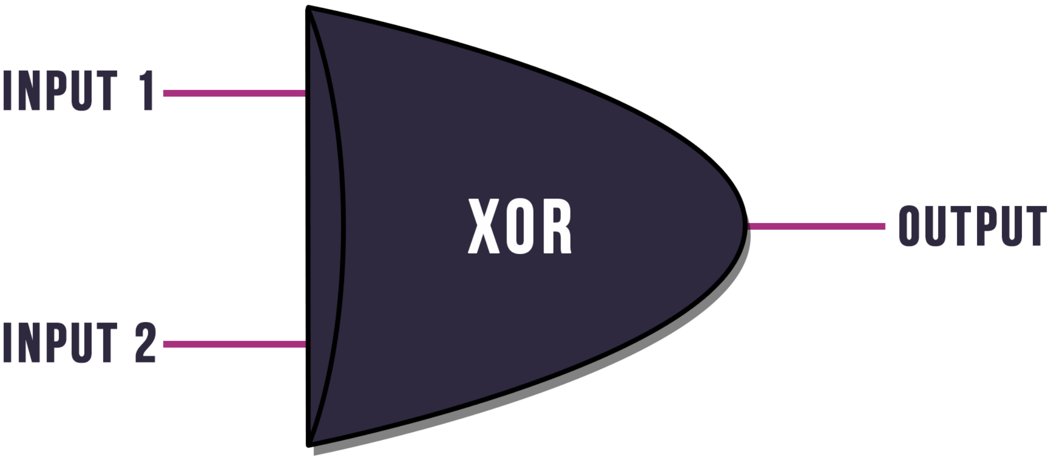
The NOT logic gate inverts the input.

Input 1	Output
1	0
0	1

For example:

```
1001 - 9 (NOT)
----
0110 - 6
```

8.5.6 The XOR Logic Gate



XORLogicGate

The XOR (exclusive-OR) logic gate is similar to an OR gate; however, if both inputs are '1', the output is '0'.

Input 1	Input 2	Output
1	0	1
0	1	1
0	0	0
1	1	0

For example:

```
1001 - 9
0101 - 5 (XOR)
----
1100 - 12
```

8.5.7 -band Bitwise AND

The bitwise 'AND' operator performs an 'AND' bitwise comparison for each input. Keep in mind, for this example; the inputs are ASCII representations of two characters.

Example 22: Bitwise AND with -band

```
$Value1 = [int]72 # 72 decimal
$Value2 = [int]101 # 101 decimal
$Value1 -band $Value2
```

```
64
```

The following table shows the same operation using binary (base-2):

- Value 1 = 0100 1000 binary = 72 decimal
- Value 2 = 0110 0101 binary = 101 decimal
- Perform a bitwise AND comparison between each of the inputs:

Value 1	Value 2	Output
0	0	0
1	1	1
0	1	0
0	0	0
1	0	0
0	1	0
0	0	0
0	1	0

- The output is 0100 0000 binary, which is 64 decimal.

8.5.8 `-bor` Bitwise OR

The 'bitwise OR' operator performs an 'OR' bitwise comparison for each input.

Example 23: Bitwise OR with -bor

```
$Value1 = [int]72 # 72 decimal
$Value2 = [int]101 # 101 decimal
$Value1 -bor $Value2
```

```
109
```

The following table shows the same operation using binary (base-2):

- Value 1 = 0100 1000 binary = 72 decimal
- Value 2 = 0110 0101 binary = 101 decimal
- Perform a bitwise OR comparison between each of the inputs:

Value 1	Value 2	Output
0	0	0
1	1	1
0	1	1
0	0	0
1	0	1
0	1	1
0	0	0
0	1	1

- The output is 0110 1101 binary, which is 109 decimal.

8.5.9 `-bxor` Bitwise XOR

The ‘bitwise XOR’ operator performs an exclusive-‘OR’ bitwise comparison for each input.

Example 24: Bitwise XOR with -bxor

```
$Value1 = [int]72 # 72 decimal
$Value2 = [int]101 # 101 decimal
$Value1 -bxor $Value2
```

```
45
```

The following example will perform the same calculation in binary (base-2):

- Value 1 = 0100 1000 binary = 72 decimal
- Value 2 = 0110 0101 binary = 101 decimal
- Perform a bitwise XOR comparison between each of the inputs:

Value 1	Value 2	Output
0	0	0
1	1	0
0	1	1
0	0	0
1	0	1
0	1	1
0	0	0
0	1	1

- The output is 0010 1101 binary, which is 45 decimal.

8.5.10 `-bnot` Bitwise NOT

The ‘bitwise NOT’ operator inverts the character string to its opposite binary value.

Example 25: Bitwise NOT with -bnot

```
$Value1 = [int]72 # 72 decimal
-bnot $Value1
```

```
-73
```

The following example will perform the same calculation in binary (base-2):

- Value 1 = 0100 1000 (base-2) [72 (base-10) represented in binary (base-2)]
- Perform a NOT inversion with the input:

Value 1	Output
0	1
1	0
0	1
0	1
1	0
0	1
0	1
0	1

- The output is 1011 0111 (base-2), which is 183 (base-10), different from the expected result to −73 (base-10).

Why? Technically, the output is correct. The table expresses an *unsigned* binary NOT operation, while PowerShell’s `-bnot` operator returns a *signed integer* (in this case, a 32-bit `[Int]`/`[Int32]`) for `[Int]$Value1`.

Signing describes an integer object’s capability to process negative numbers, using various ‘*methods of representation*’ for negative numbers. `[Int]` values are a formatted representation of a number in 32 bits (or 4 bytes), using the leading bit (`1000 0000 ...`) to denote a positive or negative number. For 8 bits, the leading bit is effectively equivalent to `-128`.

Signed:

-128	64	32	16	8	4	2	1
1	0	1	1	0	1	1	1

$$-128 + 32 + 16 + 4 + 2 + 1 = -\mathbf{73}$$

Unsigned:

128	64	32	16	8	4	2	1
1	0	1	1	0	1	1	1

$$128 + 32 + 16 + 4 + 2 + 1 = \mathbf{183}$$

So, repeating the same example as before, extended to a *signed 32-bit integer*:

- *Value 1* = 0000 0000 0000 0000 0000 0000 0100 1000 (base-2) [72 (base-10) represented in binary (base-2) as a **signed** integer]
- Perform a NOT inversion with the input:

Value 1	Output
0	1
0	1
0	1
0	1
…	…
0	1
0	1
0	1
0	1
0	1
1	0
0	1
0	1
1	0
0	1
0	1
0	1

- *Output* is 1111 1111 1111 1111 1111 1111 1011 0111 (base-2), which is −73 (base-10).

8.5.11 `-shl` Shift Bits Left

‘Shift Bits Left’ moves the `[Int]` bits to the left, ‘x’ number of times. If done in base-10, 455 (base-10) moved two times to the left would be 45500 (base-10).

The following example describes the usage in PowerShell:

Example 26: Bit-shifting to the left with -shl

```
$Value1 = 72
$Times = 2
$Value1 -shl $Times
```

```
288
```

The following example will perform the same bit-shift in binary (base-2):

- *Value 1* = 0000 0000 0000 0000 0000 0000 0100 1000 (base-2) [72 (base-10) represented in binary (base-2) as a **signed** integer]
- Perform a bit shift to the left **2** times.
- *Output* will be: 0000 0000 0000 0000 0000 0001 0010 0000 (base-2) [288 (base-10) represented in binary (base-2) as a **signed** integer]. *Note how the bits have been shifted to the left.*

8.5.12 `-shr` Shift Bits Right

‘Shift Bits Right’ moves the `[Int]` bits to the right, ‘x’ number of times. If done in base-10, 455 (base-10) moved two times to the left would be 4.55 (base-10).

Example 27: Bit-shifting to the right with -shr

```
$Value1 = 72
$Times = 2
$Value1 -shr $Times
```

```
18
```

> Note: Integer types can’t represent floating-point numbers. Bits shifted past the least significant bit (LSB) or binary one’s place are lost.

The following example will perform the same bit-shift in binary (base-2):

- *Value 1* = 0000 0000 0000 0000 0000 0000 0100 1000 (base-2) [72 (base-10) represented in binary (base-2) as a **signed** integer]
- Perform a bit shift to the right **2** times.
- *Output* will be: 0000 0000 0000 0000 0000 0000 0001 0010 (base-2) [18 (base-10) represented in binary (base-2) as a **signed** integer]. *Note how the bits have been shifted to the right this time.*

8.5.13 Practical Applications

Bitwise operators are used commonly with flag enums to perform different operations, such as combination and searching. In the example below, the `-bor` operator will be used to combine two separate enums together:

Example 28: Combining bitflag enums with -bor

```
[Flags()]
enum Colors
{
    Red = 1
    Green = 2
    Blue = 4
    Yellow = 8
    Black = 16
    White = 32
}

# Example 1: Combination
# In the following example, Red has been initially
# set within the variable $value.
# We will use the -bor operator to add Blue.
$value = [Colors]::Red
$value = $value -bor [Colors]::Blue
$value

# Example 1a: Return the integer.
[int]$value
```

```
# Example 1:
Red, Blue

# Example 1a:
5
```

Note the integer output is `5`. Red is equal to ‘1’, and Blue is equal to ‘4’. The `-bor` operator combined the two integers:

Binary	Output	Description
0001	Red	Initial item
0100	Blue	Item to combine
0101	Red, Blue	Combined item

In the following example, the `-band` operator will be used to search the value for ‘Red’ and ‘Yellow’.

Example 29: Checking for bitflag enums with -band

```
# Example 1: Searching
$value = [Colors]'Red, Green, Blue'
$value = $value -band [Colors]'Red, Yellow'
$value

# Example 1a: Return the integer.
[int]$value
```

```
# Example 1:
Red

# Example 1a:
1
```

Note the integer output is `1` being 'Red', matching only the found integer within the flag. In the following table, the `-band` calculation is performed:

Binary	Output	Description
0111	Red,Green,Blue	Initial list
1001	Red,Yellow	Items to find
0001	Red	Output

8.6 `-like` and `-notlike`

The `-like` operator is a case-insensitive simple `[String]` comparison operator similar to the `-match` operator.[16] `-like` can use the wildcard `*` character to denote a string matching pattern and returns either `$true` or `$false`. For example, this comparison will return `$true`, since the character `'a'` is present within the string:

Example 30: Searching for a phrase anywhere in a string with wildcards

```
# When a string is present

$string = 'This is a string'
$string -like '*a*'
```

```
True
```

In this example, the comparison will return `$false` since the string 'hello' is not present:

[16] Microsoft. (2022, Mar. 19). *About Comparison Operators (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_comparison_operators#matching-operators. [Accessed: Mar. 31, 2022].

Example 31: Match failures with -like return $false

```
# When a string is not present

$string = 'This is a string'
$string -like '*hello*'
```

```
False
```

Wildcard characters are used to represent one or more characters within a string (for example: `*`, `?` and `[]`). These are:

Wildcard	Description	Example
`*`	Match one or more characters	`Str*`
`?`	Match one character in that position	`Strin?`
`[rangeStart-rangeEnd]`	Match a range of characters (*separated by ‘hyphen’*)	`[a-z]tring`
`[charcharchar]`	Match a range of predefined characters (*no separator*)	`[sz]tring`

To escape matching for a non-wildcard string, use a singular backtick “’ to escape the character.

Example 32: Escaping wildcard characters with -like

```
# Example 1:
# Since the question mark is used, it will be escaped with a backtick.
'string?! 123' -like 'string`?!*'
```

```
# Example 1:
True
```

Different configurations can be used to fit different scenarios. The table below demonstrates the different configurations of the wildcards:[17]

Wildcard	Condition String	Description	String to test	Output
`*`	`"This*"`	Test for the first word in the string.	`This is a string!`	`$true`
`*`	`"this*"`	Test for the first word in the string. *(Case-insensitive)*	`This is a string!`	`$true`
`*`	`"This*"`	Test for the first word in the string.	`Hello! This is a string!`	`$false`
`*`	`"*This*"`	Test for a word inside the string.	`Hello! This is a string!`	`$true`

[17] Microsoft. (2022, Mar. 19). *About Comparison Operators (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_comparison_operators#matching-operators. [Accessed: Mar. 31, 2022].

Wildcard	Condition String	Description	String to test	Output
`*`	`"*This*"`	Test for a word inside the string	`This is a string!`	`$true`
`*`	`"*This*"`	Test for a word inside the string	`Hello! This`	`$true`
`*`	`"*String"`	Test for the last word in the string.	`The last word in string`	`$true`
`*`	`"*String"`	Test for the last word in the string.	`Hello! This my name is Michael!`	`$false`
`?`	`"Strin?"`	Test the last character in the position.	`Strin5`	`$true`
`?`	`"Strin?"`	Test the last character in the position.	`This is a String!`	`$false`
`?`	`"b?ok"`	Test a random character in the position.	`book`	`$true`
`?`	`"b?ok"`	Test a random character in the position.	`cook`	`$false`
text-range `[start-end]`	`"[a-z]ook"`	Test a range of characters.	`cook`	`$true`
text-range `[start-end]`	`"[a-z]ook"`	Test a range of characters.	`book`	`$true`
text-range `[start-end]`	`"[a-z]ook"`	Test a range of characters.	`took`	`$true`
text-range `[start-end]`	`"[a-z]ook"`	Test a range of characters.	`a book`	`$false`
`[charchar]`	`"[lb]ook"`	Match a range of predefined characters.	`book`	`$true`
`[charchar]`	`"[lb]ook"`	Match a range of predefined characters.	`look`	`$true`
`[charchar]`	`"[lb]ook"`	Match a range of predefined characters.	`cook`	`$false`

The `-like` operator can also be applied to lists and arrays, where the array is defined on the left side. Instead of a `boolean`, the operator will return any matching values in the array.

Example 33: Using the -like operator to filter arrays

```
# Example 1:
$arr = 'string','bees','timber'
$arr -like '*e*'
# Example 2:
$arr = 'A small string','There are some bees','Some timber'
$arr -like '*e*'
```

```
bees
timber

There are some bees
Some timber
```

If the array or list doesn't contain a value, `-like` will return an empty array, which can be tested using `-not ($arr -like '*is not in list*')`.

Example 34: -like returns an empty array if it finds no matches

```
$arr = 'string','bees','timber'
# Example 1
-not ($arr -like '*is not in list*')
# Example 2
($arr -like '*is not in list*').Count
# Example 3
($arr -like '*is not in list*').GetType()
```

```
# Example 1
True
# Example 2
0
# Example 3
IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Object[]                                 System.Array
```

The `-notlike` operator is the inverse of the `-like` operator.[18] This also applies to matching against lists and arrays, where `-notlike` will return an array of strings that don't match.

Example 35: -notlike provides the inverse results for scalars and arrays

```
$arr = 'string','bees','timber'
# Output 1
$arr -notlike '*is not in list*'
# Output 2
($arr -notlike '*is not in list*').Count
```

```
# Output 1
string
bees
timber
# Output 2
3
```

[18] Microsoft. (2022, Mar. 19). *About Comparison Operators (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_comparison_operators#matching-operators. [Accessed: Mar. 31, 2022].

Note: If advanced matching is required, consider using the -match operator.

8.7 -match and -notmatch

Syntax:

```
'String' -match 'Regex Expression'
```

The -match and -notmatch operators use regular expressions to search for a pattern within a string.[19] The left side of the expression contains the string, whereas the right side contains the regular expression. When parsing arrays into the left side, -match functions similar to -like, where it outputs each matching value in the defined array.

Example 36: Matching with regex using -match

```
# Example 1: Standard Match
$string = 'this is a string'
$string -match '.+'

# Example 2: Array Match
$arr = 'this is a string', 'this also is a string'
$arr -match '.+'
```

```
# Example 1:
True

# Example 2:
this is a string
this also is a string
```

-match is covered extensively in the Regex Chapters.

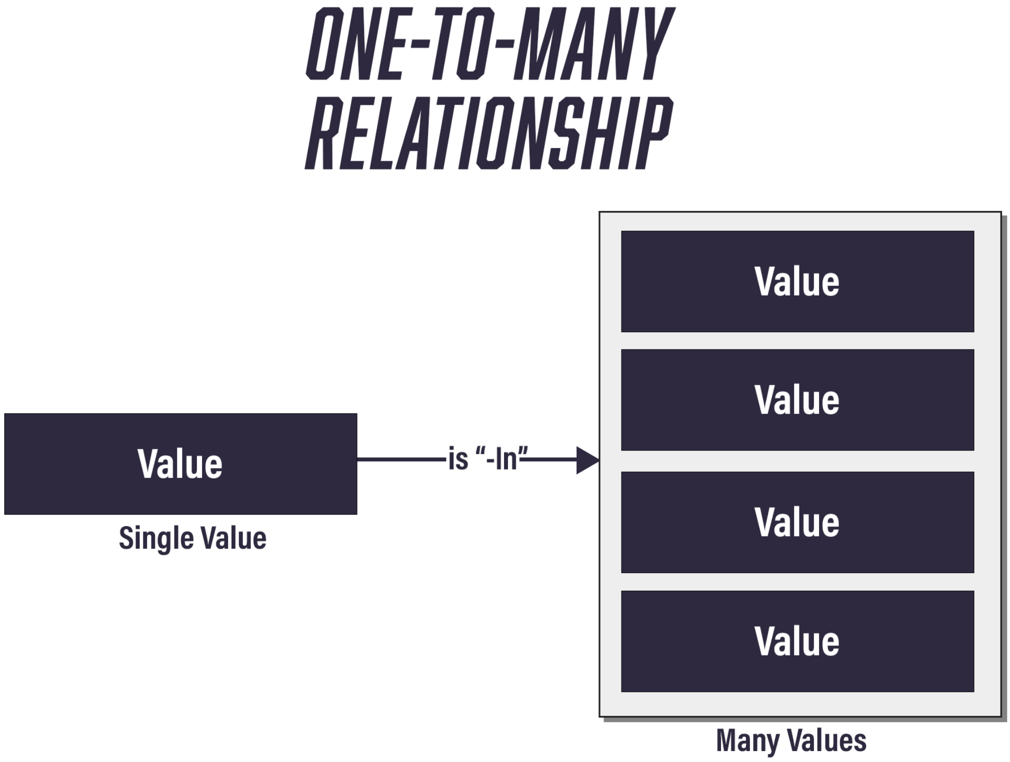
8.8 -in, -contains, -notin and -notcontains

The -in and -contains operators test whether a list, array or collection contains a specific value, similar to the IndexOf() method.[20]

The -in operator functions in the left-hand **one-to-many** case, where a value may exist in a list or array:

[19] Microsoft. (2022, Mar. 19). *About Comparison Operators (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_comparison_operators#matching-operators. [Accessed: Mar. 31, 2022].

[20] Microsoft. (2022, Mar. 19). *About Comparison Operators (Microsoft.PowerShell.Core) - Containment operators*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_comparison_operators#containment-operators. [Accessed: Mar. 31, 2022].



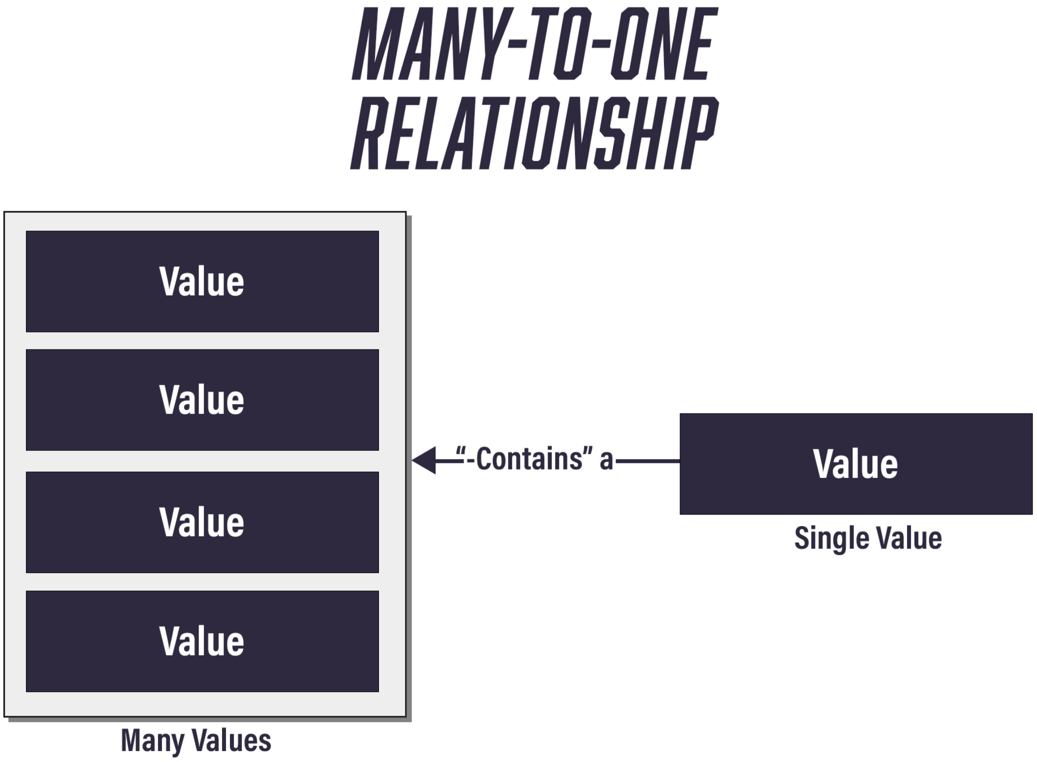
INOperatorImage

Example 37: Testing for values in collections with -in

```
$values = @(
    "Value1",
    "Value2",
    "Value3"
)

$string = "Value1"

$string -in $values
```

```
True
```

Conversely, the `-contains` operator functions in the right-hand **many-to-one** case, where the list or array may contain a value.



ContainsOperatorImage

Example 38: Testing whether a collection contains a value with -contains

```
$values = @(
    "Value1",
    "Value2",
    "Value3"
)

$string = "Value1"

$values -contains $string
```

```
True
```

All base types are supported.

Example 39: -contains works with all base types

```
$arr = @(
    [Int]1,
    [DateTime]"05-15-2021",
    [String]"String",
    [Byte]14,
    [TimeSpan]15,
    [Bool]'True',
    [char]'C'
)

$arr -contains ([DateTime]"05-15-2021")
$arr -contains [Int]1
$arr -contains [String]'String'
$arr -contains [Byte]14
$arr -contains [TimeSpan]15
$arr -contains $true
$arr -contains [char]'C'
```

```
True
True
True
True
True
True
True
```

The `-in` and `-contains` operators simplify code within conditions by removing the need to use a `Where-Object` within the logic.

Example 40: Testing an array for a value with -in

```
$values = @(
    "Value1",
    "Value2",
    "Value3"
)

$string = "Value1"

# Integrate the Operator into the Condition
if ($string -in $values) {
    # Do something
}
```

These operators are limited to comparison with members of a list or array. When interrogating a collection of objects, use an expression to store the desired property values from each object into an array.

In the following example, the `Name` property array is returned from `Get-Process`, allowing the use of `-contains`.

Example 41: Checking an array of process names for a value with -contains

```
# Fetch the Processes and Select the Name property
if ((Get-Process).Name -contains 'pwsh') {
    # Do something
}
```

The `-notin` and `-notcontains` operators test the inverse to `-in` -and `-contains`. They are used to test if an object *doesn't* exist in a list or array.

Example 42: -notcontains checks for the absence of a value

```
# In this instance, we are testing to make sure
# that the process doesn't contain the PowerShell Process
if ((Get-Process).Name -notcontains 'pwsh') {
    # The process doesn't exist
} else {
    # The process does exist
}
```

8.9 `-replace`

Syntax: `<input> -replace <regular-expression>, <substitute>`

The `-replace` operator is a regex-enabled string operator, similar to the `String.Replace()` method.[21]

For example, the -replace operator can perform a basic string replacement:

Example 43: Using the -replace operator to substitute a substring

```
$Name = 'Hello! My name is: Ben!'
$Name -replace 'Ben', 'Michael'
```

```
Hello! My name is: Michael!
```

Reviewing the example, the variable `$Name` is declared as: `'Hello! My name is: Ben!'`. Using the `-replace` operator, `Ben` is changed to `Michael`.

Unlike `String.Replace()`, the `-replace` operator uses regex to match patterns to replace. In this example, the first two words ‘X marks’ are replaced in ‘Lets find the spot!’

[21] Microsoft. (2022, Mar. 19). *About Comparison Operators (Microsoft.PowerShell.Core) - Replacement operator.* Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_comparison_operators#replacement-operator. [Accessed: Mar. 31, 2022].

Example 44: Using regex patterns with -replace

```
$string = 'Lets find the spot. OR should it go north?'
$string -replace '^(?:\S+\s){1}(\S+)', 'X marks'
```

```
X marks the spot. OR should it go north?
```

Here, the second word, ‘PowerShell’, is replaced with ‘is cool!’:

Example 45: Using substitutions with -replace

```
$string = 'PowerShell PowerShell'
$string -replace '(\w+) \w+$', '$1 is cool!'
```

```
PowerShell is cool!
```

When parsing a string that contains regex queries, use the `[Regex]::Escape()` method to escape these characters.

Example 46: Escaping regex language elements in a string with Escape()

```
$string = "PowerShell [(\w+)$)] is cool!"
[Regex]::Escape($string)
```

```
PowerShell\ \[\(\\w\+\)\$\)]\ is\ cool!
```

For more information on regex, please refer to the Regex Chapters.

8.10 Ternary Operator `(condition) ? <true> : <false>`

Note: This is a *PowerShell 7 feature.*

Syntax:

```
(condition) ? <value if true> : <value if $false>
```

The ternary operator is essentially a simplified if-statement.[22] It consists of three parts:

1. The condition (`condition`). The condition is wrapped in parentheses `()` (like the if-statement) and followed by a question mark.
2. The *true* expression (precedes the colon). The *true* expression denotes the output if the condition evaluates to `$true`.
3. The *false* expression (follows the colon). The *false* expression denotes the output if the condition evaluates to `$false`.

[22] Microsoft. (2022, Mar. 19). *About Comparison Operators (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_comparison_operators. [Accessed: Mar. 30, 2022].

Example 47: Using the ternary operator for conditional statements

```
# Example 1: A Simple Condition where it evaluates to be $true.
$result = ('value' -eq 'value') ? 'isPresent' : 'notPresent'
$result

# Example 2: A Simple Condition where it evaluates to be $false.
$result = ('value' -eq 'another value') ? 'isPresent' : 'notPresent'
$result

# Example 3: Using a different condition.
$result = ('value' -in
    ('value','option','selection')) ? 'isPresent' : 'notPresent'
$result
```

```
# Example 1:
isPresent

# Example 2:
notPresent

# Example 3:
isPresent
```

8.11 Null-Coalescing Operator `??`

> Note: This is a *PowerShell 7 feature.*

Syntax:

```
<Value to test> ?? <Value if $null>
```

The null-coalescing operator `??` tests a value or expression and returns an alternative value if `$null`.[23]

[23] Microsoft. (2022, Mar. 19). *About Comparison Operators (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_comparison_operators. [Accessed: Mar. 30, 2022].

Example 48: Using the null-coalescing operator to provide fallback values

```
# Example 1: Test 'not-null' -eq $null.
# In this test, it will evaluate to be $false,
# so no changes are applied.
'not-null' ?? 'is null'
# Example 2: Test $null -eq $null.
# In this test, it will evaluate to be $false,
# so 'is null' is returned.
$null ?? 'is null'
```

```
# Example 1:
'not-null'

# Example 2:
'is null'
```

There are many practical applications for using these operators.

Example 49: The traditional approach for handling null cmdlet results

```
$params = @{
    URI = 'https://example.com/'
    ErrorAction = 'SilentlyContinue'
    Method = $(
        if ($method -ne $null) { $method }
        else { 'GET' }
    )
}

$result = Invoke-WebRequest @params
if ($null -eq $result) {
    $result = @{error = 'no response'}
}
```

This can be refactored to:

Example 50: Handling null cmdlet results concisely with the null-coalescing operator

```
$params = @{
    URI = 'https://example.com/'
    ErrorAction = 'SilentlyContinue'
    Method = $method ?? 'GET'
}

$result = Invoke-WebRequest @params ?? @{error = 'no response'}
```

It's important to remember that empty `[String]` objects and those with only white space characters are not `$null` and the right side will not be evaluated in these cases. This is also the case for values that evaluate to *false*.

Example 51: Empty strings and false values don't count as null conditions

```
# Example 1:
'' ?? 'is null'

# Example 2:
' ' ?? 'is null'

# Example 3:
$false ?? 'is null'
```

```
# Example 1:


# Example 2:


# Example 3:
False
```

Expressions can be parsed by wrapping them in a subexpression `$()`.

Example 52: Using subexpressions with the null-coalescing operator

```
$($null) ?? $(Get-Process)
```

```
 NPM(K)    PM(M)      WS(M)     CPU(s)      Id  SI ProcessName
 ------    -----      -----     ------      --  -- -----------
      0     0.00      32.71       0.16      79  79 node
      0     0.00      65.89      13.95      98  79 node
      0     0.00      51.18       2.65     251  79 node
      0     0.00      41.03       2.57     451 451 node
      0     0.00      41.23       1.05     484  79 node
      0     0.00      52.50       6.13     496 496 node
      0     0.00     316.03     146.35     514  79 node
      0     0.00     126.63       3.49     529 529 pwsh
      0     0.00       1.60       0.41       1   1 sh
      0     0.00       1.59       0.04      12  12 sh
      0     0.00       0.52       0.00      96  79 sh
      0     0.00       0.59       0.00     585  79 sh
      0     0.00       0.57       0.00    8981   1 sleep
      0     0.00     104.09       4.56     586  79 vsls-agent
```

8.12 Null-Coalescing Assignment Operator `??=`

> Note: This is a *PowerShell 7 feature.*

The null-coalescing assignment operator is used to assign an expression to a variable only if the variable *value* is `$null`.[24]

[24] Microsoft. (2022, Mar. 19). *About Comparison Operators (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_comparison_operators. [Accessed: Mar. 30, 2022].

Example 53: Assigning values only to null variables with the null-coalescing assignment operator

```
# Example 1: $var will not be $null
$var = 'not-null'
$var ??= 'is null'
$var

# Example 2: $var will be $null
$var = $null
$var ??= 'is null'
$var

# Example 3: $var will be Empty String
$var = ''
$var ??= 'is null'
$var
```

```
# Example 1:
not-null

# Example 2:
is null

# Example 3:
```

For example, the following logic:

Example 54: The traditional approach for assigning alternative values to null variables

```
$var = Do-Something
if ($null -eq $var) {
    $var = 'something else'
}
```

Can be refactored to:

Example 55: Assigning a null-conditional alternative value with the null-coalescing assignment operator

```
$var = Do-Something
$var ??= 'something else'
```

8.13 Null-Conditional Operator `?.` and `?[]`

Note: This is a *PowerShell 7 feature.*

Null-Conditional object selection operators allow you to select object properties tentatively (`?.`) or array items (`?[]`), returning `$null` if the property or item is `$null`.[25]

The character '?' is permitted within variable names; variables must be wrapped with curly braces `${var}`. Subexpressions are also permitted as well `$($var)?.Method()`.

[25] Microsoft. (2022, Mar. 19). *About Comparison Operators (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_comparison_operators. [Accessed: Mar. 30, 2022].

Example 56: Using the null-conditional member access operator

```
# Example 1: VariableName
$va? = 'test'
${va?}

# Example 2: SubExpression
$var = 'string'
$($var)?.Replace('g','gs')

# Example 3: SubExpression, with quotation
$va? = 'test'
${va?}?.Replace('e', 'oa')

# Example 4: SubExpression, with quotation
$va? = $null
${va?}?.Replace('e', 'oa')
```

```
# Example 1:
test

# Example 2:
strings

# Example 3:
toast

# Example 4:
```

8.13.1 Examples of `?.`

The following examples demonstrate the use of: ‘`?.`’:

Example 57: Accessing a valid object property with and without the null-conditional operator

```
# Example 1: Create an Object and Select a Property
# Nothing is changed here.

$obj = [PSCustomObject]@{
    Property = 'Value'
}
$obj.Property

# Example 2: Create an Object and Select the Property
# using a Null Conditional Operator

$obj = [PSCustomObject]@{
    Property = 'Value'
}
# Note that the variable needs to be wrapped in curly braces:
${obj}?.Property
```

```
# Example 1:
Value

# Example 2:
Value
```

In this example, the object property will be set to `$null`:

Example 58: Trying to access an invalid property with the null-conditional operator

```
# Example 1: Create an Object. However, the property is $null
$obj = [PSCustomObject]@{
    Property = $null
}
# Note that the property returns nothing
${obj}?.Property
# Let's test to make sure it's null
${obj}?.Property -eq $null

# Example 2: Create an Object and select a different property:
$obj = [PSCustomObject]@{
    Property = 'value'
}
# Note that the property returns nothing
${obj}?.AnotherProperty
# Let's test to make sure it's null
${obj}?.AnotherProperty -eq $null
```

```
# Example 1:
True

# Example 2:
True
```

The use cases for this operator are unclear since PowerShell 5.1 already casts empty object properties to `$null` (null-soaking). However, attempting to access null properties in strict mode throws a reference error.[26] This operator therefore has some uses in strict mode.

Example 59: The null-conditional member access operator is useful in strict mode

```
# Example 1: Null-soaking in default mode
$obj = [PSCustomObject]@{
    Property = $null
}
$obj.Property.SecondProperty.ThirdProperty -eq $null

# Example 2: Reference errors in strict mode
Set-StrictMode -Version 2
$obj.Property.SecondProperty -eq $null

# Example 3: Safe access to null properties with ?.
$obj.Property?.SecondProperty -eq $null

# Example 4: The ?. operator doesn't protect against nonexistent properties
```

[26] Microsoft. (2022, Mar. 08). *Set-StrictMode (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/set-strictmode. [Accessed: Apr. 03, 2022].

```
# in strict mode
${obj}?.AnotherProperty -eq $null

Set-StrictMode -Off
```

```
# Example 1:
True

# Example 2:
PropertyNotFoundException: The property 'SecondProperty' cannot be found
on this object. Verify that the property exists.

# Example 3:
True

# Example 4:
PropertyNotFoundException: The property 'AnotherProperty' cannot be found
on this object. Verify that the property exists.
```

8.13.2 Examples of `?[]`

The following examples demonstrate the use of: '`?[]`':

Example 60: Accessing array elements safely with ?[]

```
# Example 1: Accessing an array element with []
$arr = 1,2,3,4
$arr[1]

# Example 2: Accessing an array element with ?[]
# Note: The variable needs to be wrapped with curly braces:
$arr = 1,2,3,4
${arr}?[1]

# Example 3: Accessing an array element that doesn't
# exist, without the operator.
$arr = $null
$arr[10] -eq $null

# Example 4: The same example as before, however
# using the ?[] operator.
$arr = $null
${arr}?[10] -eq $null

# Example 5: A Complex Object Structure.
$objlist = $null
${objlist}?[10].List?[10] -eq $null
```

```
# Example 1:
2

# Example 2:
2

# Example 3:
InvalidOperation: Cannot index into a null array.

# Example 4:
True

# Example 5:
True
```

8.14 `:parent` Loop Labels

Syntax:

```
:<LabelName> <Loop Construct> {
    # ScriptBlock
}
```

PowerShell loop labeling adds a label to a loop construct, which is used to control the flow of the loop.[27] [28] Labels are defined at the start of a loop construct, prefixing the label with a colon `:`.

Example 61: Labeling a loop statement

```
# Define the Loop Label inside a 'do loop'.

# Create a label called 'loopLabel'
:loopLabel do {
    # Do something
} until ($true)
```

Once the loop label has been defined, `break`/`continue` statements can reference the label to exit at that loop level. This is useful when managing nested loop statements, since there is less need for complex branching logic. In the following example, the nested loop statement will exit the parent loop once the counter reaches `3`:

[27] Microsoft. (2014, May. 08). *PowerShell Looping: Advanced Break.* Microsoft Dev Blogs. [Online]. Available: https://devblogs.microsoft.com/scripting/powershell-looping-advanced-break/. [Accessed: Mar. 31, 2022].

[28] Microsoft. (2022, Mar. 19). *About Break (Microsoft.PowerShell.Core) - Using a labeled break in a loop.* Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_break#using-a-labeled-break-in-a-loop. [Accessed: Mar. 31, 2022].

Example 62: Breaking an outer loop using a loop label

```
# Define the Loop Label to be 'loopLabel'
# This parent do-loop will loop forever by default.
:loopLabel do {
    # Nested for-loop within the do-loop
    # This loop will count from 0 to 10.
    for ($i=0; $i -ne 10; $i++) {

        # If the counter reaches 3, we will break (exit) the parent do-loop.
        if ($i -eq 3) {
            # Specify the target label for the break statement.
            break :loopLabel
        }

        Write-Host "Nested Counter: $i"

    }

} until ($true)
```

```
Nested Counter: 0
Nested Counter: 1
Nested Counter: 2
```

In the following example, the `Continue` statement will be used to increment the top-level counter:

Example 63: Continuing an outer loop using a loop label

```
# Define the Loop Label to be loopLabel
# This top-level loop will count from 0 to 5.
$Counter = 0
:loopLabel do {
    # Nest a secondary loop within the first.
    # This loop will count from 0 to 10.
    for ($i=0; $i -ne 10; $i++) {

        # If the counter reaches 3, we will exit the nested loop
        # and continue within the parent loop.
        if ($i -eq 3) {
            # Specify the target label for the continue statement.
            continue :loopLabel
        }

        Write-Host "Top Level Counter: $Counter Nested Counter: $i"

    }

    $Counter++

} until ($Counter -eq 5)
```

```
Top Level Counter: 0 Nested Counter: 0
Top Level Counter: 0 Nested Counter: 1
Top Level Counter: 0 Nested Counter: 2
Top Level Counter: 0 Nested Counter: 4
Top Level Counter: 0 Nested Counter: 5
Top Level Counter: 0 Nested Counter: 6
Top Level Counter: 0 Nested Counter: 7
Top Level Counter: 0 Nested Counter: 8
Top Level Counter: 0 Nested Counter: 9
Top Level Counter: 1 Nested Counter: 0
Top Level Counter: 1 Nested Counter: 1
Top Level Counter: 1 Nested Counter: 2
Top Level Counter: 1 Nested Counter: 4
Top Level Counter: 1 Nested Counter: 5
Top Level Counter: 1 Nested Counter: 6
Top Level Counter: 1 Nested Counter: 7
Top Level Counter: 1 Nested Counter: 8
Top Level Counter: 1 Nested Counter: 9
Top Level Counter: 2 Nested Counter: 0
Top Level Counter: 2 Nested Counter: 1
Top Level Counter: 2 Nested Counter: 2
Top Level Counter: 2 Nested Counter: 4
Top Level Counter: 2 Nested Counter: 5
Top Level Counter: 2 Nested Counter: 6
Top Level Counter: 2 Nested Counter: 7
Top Level Counter: 2 Nested Counter: 8
Top Level Counter: 2 Nested Counter: 9
Top Level Counter: 3 Nested Counter: 0
Top Level Counter: 3 Nested Counter: 1
Top Level Counter: 3 Nested Counter: 2
Top Level Counter: 3 Nested Counter: 4
Top Level Counter: 3 Nested Counter: 5
Top Level Counter: 3 Nested Counter: 6
Top Level Counter: 3 Nested Counter: 7
Top Level Counter: 3 Nested Counter: 8
Top Level Counter: 3 Nested Counter: 9
Top Level Counter: 4 Nested Counter: 0
Top Level Counter: 4 Nested Counter: 1
Top Level Counter: 4 Nested Counter: 2
Top Level Counter: 4 Nested Counter: 4
Top Level Counter: 4 Nested Counter: 5
Top Level Counter: 4 Nested Counter: 6
Top Level Counter: 4 Nested Counter: 7
Top Level Counter: 4 Nested Counter: 8
Top Level Counter: 4 Nested Counter: 9
```

Loop labels are unsupported in both the `foreach` and `ForEach-Object` statements and the `$List.ForEach()` method. For example, loop labels do not affect the `foreach` statement:

Example 64: Loop labels don't apply to foreach statements

```
$array = 1,2,3,4,5

:loopLabel foreach ($item in $array) {
    for ($i=0; $i -ne 10; $i++) {

        # If the counter reaches 3, we will break (exit) the loop,
        # On the parent loop.
        if ($i -eq 3) {
            # Define the loop label by appending the label after
            # the break statement.
            break :loopLabel
    }

    Write-Host "Top Level Counter: $item Nested Counter: $i"

  }
}
```

```
Top Level Counter: 1 Nested Counter: 0
Top Level Counter: 1 Nested Counter: 1
Top Level Counter: 1 Nested Counter: 2
Top Level Counter: 2 Nested Counter: 0
Top Level Counter: 2 Nested Counter: 1
Top Level Counter: 2 Nested Counter: 2
Top Level Counter: 3 Nested Counter: 0
Top Level Counter: 3 Nested Counter: 1
Top Level Counter: 3 Nested Counter: 2
Top Level Counter: 4 Nested Counter: 0
Top Level Counter: 4 Nested Counter: 1
Top Level Counter: 4 Nested Counter: 2
Top Level Counter: 5 Nested Counter: 0
Top Level Counter: 5 Nested Counter: 1
Top Level Counter: 5 Nested Counter: 2
```

8.15 PowerShell Operator Precedence

Operator precedence describes the list of rules for how PowerShell operators are evaluated within an expression.[29] The precedence order is an ordered list of which operators are evaluated by PowerShell first. An expression is a piece of valid code that returns a value (for example: `1 -eq 1`).

During the evaluation process, several rules apply:

- If the operator is an *assignment* operator (`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `++`, and `--`), a *cast* operator (`[type]$val`) or a *negation* operator (`-not`, `-isnot`, and `-notcontains`), the expression is evaluated from right to left.
- Otherwise, the precedence order is applied from the list (see table below):

Example 65: Assignment operators have the lowest precedence

```
$var = 'Value' -eq 'Value'
```

In this example, a variable is declared containing the string value `'value' -eq 'value'`. The process is as follows:

1. PowerShell reviews the expression by scanning *left-to-right*.

[29] Microsoft. (2022, Mar. 19). *About Operator Precedence (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_operator_precedence. [Accessed: Mar. 19, 2022].

2. The comparison operator (`-eq`) is found in `'Value' -eq 'Value'`. The comparison operator (`-eq`) is higher than the assignment operator, so it’s evaluated first, producing `$true`.
3. The assignment operator (`=`) comes last and will assign the output to the variable `$var` (*reading from right-to-left*).

The following table describes the order of precedence; items in the same *precedence group* follow the same precedence rules:

Precedence	Operator	Description	Example
1	`$()`, `@()`, `()`, `@{}`	Subexpression, array subexpression, grouping, and hashtable	`"$(1+1)"`, `@(1,2,3)`
2	`.`, `?.`	Member access and null-conditional	`$Object.Property`, `${Object}?.Property`
3	`::`	Static	`[String]::Copy($string)`
4	`[]`, `?[]`	Index and null-conditional	`$arr[0]`, `${arr}?[0]`
5	`[type]`	Cast	`[string]`
6	`-split`	Unary split	`-split 'string1 string2'`
7	`-join`	Unary join	`-join ('string1', 'string2')`
8	`,`	Comma	`1,2,3`
9	`++`, `--`	Increment, Decrement	`$i++`
10	`!`, `-not`	Logical inverse	`-not($false)`
11	`..`	Range	`[0..5]`
12	`-f`	Format	`"{0}" -f 'val'`
13	`-`	Unary minus/negative	`-1`
14	`*`, `/`, `%`	Multiply, Divide, Modulo	`3 * 1`, `10 / 2`, `10 % 2`
15	`+`, `-`	Add, Subtract	`3 + 1`, `10 - 2`
16	`-split`	Binary split	`'1,2,3,4' -split ','`
17	`-join`	Binary join	`1,2,3,4 -join ','`
18	`-is`, `-isnot`, `-as`	Type compare/Coerce	`1 -is [int]`
19	`-eq`, `-ne`, `-gt`, `-ge`, `-lt`, `-le`	Equality/Comparison	`1 -eq 1`
20	`-like`, `-notlike`	Matching (wildcard)	`'string' -like 's*'`
21	`-match`, `-notmatch`	Matching (regex)	`'string' -match 's'`
22	`-in`, `-notin`	Containment (right hand)	`1 -in 1,2,3`
23	`-contains`, `-notcontains`	Containment (left hand)	`1,2,3 -contains 1`
24	`-replace`	Replacement	`1,2,3 -replace 1, 0`
25	`-band`, `-bnot`, `-bor`, `-bxor`, `-shr`, `-shl`	Binary arithmetic	`10 -band 15`
26	`-and`, `-or`, `-xor`	Logical	`1 -eq 1 -and 2 -eq 2`
27	`.`	Dot source	`. { $val = 'string'}; $val`

Precedence	Operator	Description	Example
28	&	Call	& { Write-Host 'Hello!'}
29	? <True> : <False>	Ternary operator	$val = (1 -eq 1) ? 15 : 20
30	??	Null-coalescing	$null ?? 100
31	|	Pipe	Get-Process -Name Notepad | Stop-Process
32	>, >>, 2>, 2>>, 2>&1	Redirect	Get-Process -Name Notepad > np.txt
33	&&, ||	Pipeline chain	'First' || 'Second'
34	=, +=, -=, *=, /=, %=, ??=	Assignment	$var = 'value'

Remember, the semi-colon (;) represents the end of the statement!

8.15.1 Example - Operator Precedence (,, [])

The following examples will explore the operator precedence for different statements. This example will explore the order of precedence with the comma (,) and index ([]) operators.

Example 66: Array index operators have a higher precedence than commas

```
1,2,'string'[0]
```

```
1
2
s
```

PowerShell performs the following steps:

1. PowerShell reviews the expression by scanning *left-to-right*.
2. The comma operators 1,2,'string' are found.
3. The index operator [0] is found. **The operator precedence is: [] then ,.**
4. Since the *index operator* ([]) has higher precedence to the *comma operator* (,), 'string'[0] is evaluated first, returning a substring value of 's'.
5. The *comma operator* is parsed, returning an array of 1,2,'s'.

8.15.2 Example - Parentheses ()

In the example below, the same expression is evaluated, but parentheses are added:

Example 67: Parentheses have the highest precedence

```
('string',1,2)[0]
```

```
string
```

PowerShell performs the following steps:

1. PowerShell reviews the expression by scanning *left-to-right*.
2. The *parentheses operators* `()` are found.
3. The comma operators `'string',1,2` are found.
4. The index operator `[0]` is found. **The operator precedence is: `()` then `[]`.**
5. The *parentheses operators* have the highest precedence, and the nested statement is parsed. The statement contains the *comma operator* (`,`) and no other operators. The *comma operator* will have the highest precedence in this subexpression.
6. The *comma operator* is parsed, returning an array of `string,1,2`.
7. The *index operator* lastly is parsed, returning: `'string'`.

8.15.3 Example - Negation Operator `-not`

In the example below, the following expression is parsed:

Example 68: Negation operators change the processing direction locally

```
@{index=1}, -not $true, 3
```

```
Name                           Value
----                           -----
index                          1
False
3
```

PowerShell performs the following steps:

1. PowerShell reviews the expression by scanning *left-to-right*.
2. The *hashtable literal syntax* `@{}` is found.
3. The comma operators `@{index=1}, -not$true, 3` are found.
4. The `-not` operator is found. **The operator precedence is: `@{}`, `,` then `-not`.**
5. The *hashtable literal syntax* has the highest precedence, and the key/item expressions are evaluated first.
6. `@{index=1}` is parsed.

   - PowerShell reviews the expression by scanning *left-to-right*.
   - The assignment operator is the only operator.

   - The value is assigned to `index`.

7. The comma operator has the next highest precedence, and the items are cast as an array.
8. The `-not` expression is evaluated:

   - PowerShell reviews the expression by scanning *left-to-right*.
   - The expression `-not` is found.
   - The expression switches from *left-to-right* to *right-to-left*, returning `$false`.

9. The array is returned to the output stream.

8.15.4 Example - Equal Precedence

In the example below, the operators `-eq` and `-ne` will be used multiple times in the same expression:

Example 69: Left-to-right processing is used for operators with equal precedence

```
1 -eq 2 -eq 2 -ne 2
```

```
True
```

PowerShell performs the following steps:

1. PowerShell reviews the expression by scanning *left-to-right*.
2. The first `-eq` comparison operator is found.
3. The second `-eq` comparison operator is found.
4. The third `-eq` comparison operator is found.
5. The `-ne` comparison operator is found. Since all the operators have ‘equal precedence’, the order of operations will be from left to right. **The operator precedence is**: **`-eq`**, **`-eq`**, **`-eq`**, **then `-ne`**.
6. `1 -eq 2` is evaluated, returning `$false`.
7. `$false -eq 2` is evaluated, returning `$false`.
8. `$false -ne 2` is evaluated as $true, returning the output.

8.15.5 Example - A Complex Expression

In the example below, a complex expression will be broken down:

Example 70: Processing for a complex expression with many operators

```
$PSCustomObject = [PSCustomObject]@{
    DateTime = '1/1/2021' -as [DateTime]
    Processes = (Get-Process -Name 'code').id -join ''
    BinaryOperation = -bnot 15 -shl 1; AnotherValue = 4
}
$PSCustomObject
```

```
DateTime               Processes
--------               ---------
1/1/2021 12:00:00 AM 605261729300122641423220776299883124035616

BinaryOperation AnotherValue
--------------- ------------
-32            4
```

PowerShell performs the following steps:

1. The multiline expression is formatted as a single-line expression.
2. PowerShell reviews the expression by scanning *left-to-right*.
3. The assignment operator is found (=).
4. PowerShell finds the 'cast operator' `[PSCustomObject]`.
5. The *hashtable literal syntax* `@{}` is found, with several nested expressions. The expressions are separated by 'semi-colon' (`;`) or newline, split 'BinaryOperation' and 'AnotherValue' into two different expressions. **The operator precedence is: `@{}`, `[]`, then `=`.** The *hashtable literal syntax* has the highest precedence; PowerShell will now evaluate the nested key/item expressions.
6. `DateTime = '1/1/2021' -as [DateTime]` is parsed:

    - PowerShell reviews the expression by scanning *left-to-right*.
    - The assignment operator is found (=).
    - The type operator `-as` is found. **The operator precedence is: `-as` then `=`.**
    - `-as` is parsed, and the output is stored in the key '`DateTime`'.

7. `Processes = (Get-Process -Name 'code').id -join ''` is parsed:

    - PowerShell reviews the expression by scanning *left-to-right*.
    - The assignment operator is found (=).
    - Parentheses `()` operators are found.
    - The member access operator (`$object.`) is found.
    - The `-join` operator is found. **The operator precedence is: `()`, `.`, `-join`, then `=`.**
    - Since the parentheses have the highest precedence to `-join`, `.` and `=`, the expression inside is evaluated first.
    - The output of the expression is:

```
Handles  NPM(K)    PM(K)      WS(K)     CPU(s)     Id  SI ProcessName
-------  ------    -----      -----     ------     --  -- -----------
    214      12     9768      25856       0.09   6052   1 Code
    631      40   326800     157644      97.91   6172   1 Code
    887      37    47232     101088     221.19   9300   1 Code
    334      44    50116     104012      26.23  12264   1 Code
    725      32   229060     281708     456.44  14232   1 Code
    279      19    12920      39904      17.27  20776   1 Code
    350      19    42996      90804      91.52  29988   1 Code
    170      13    20244      67196       0.31  31240   1 Code
    173      13    18924      70404       0.72  35616   1 Code
```

- Once the expression is evaluated, the remaining operators (`.`, `-join` and `=`) are evaluated. The member access operator (`.`) has the next highest precedence over the `-join` operator and the assignment operator (=). PowerShell will evaluate `(expression).id` first. The output of `(expression).id`:

  ```
  6052
  6172
  9300
  12264
  14232
  20776
  29988
  31240
  35616
  ```

- Two operators remain (`-join` and `=`). The `-join` operator has the higher precedence and will be evaluated first. The output of `(expression).id -join ''`:

  ```
  605261729300122641423220776299883124035616
  ```

- The remaining operator is the assignment operator `=`.
- The output is stored in the key '`Processes`'.

8. `BinaryOperation = -bnot 15 -shl 1` is parsed:

   - PowerShell reviews the expression by scanning *left-to-right*.
   - The assignment operator is found (=).
   - The `-shl` arithmetic operator is found.
   - The `-bnot` arithmetic is found. Note that `-shl` and `-bnot` have equal precedence. The expression will be evaluated from *left-to-right*. **The operator precedence is: `-bnot`, `-shl`, then `=`.**
   - `-bnot 15` is evaluated (giving -16), leaving `-shl` and `=`.
   - `-16 -shl 1` is evaluated, leaving `=`.
   - The output `-32` is stored.

9. `AnotherValue = 4` is parsed:

   - PowerShell reviews the expression by scanning *left-to-right*.
   - The assignment operator is the only operator.
   - The value is assigned to `AnotherValue`.

10. The hashtable expression parsing is complete, leaving operators `[PSCustomObject]` and `=`. The cast operator `[PSCustomObject]` has the higher precedence, so the `[HashTable]` object is cast as a `[PSCustomObject]`.
11. The returned `[PSCustomObject]` is stored in the variable `$PSCustomObject`.

8.16 Further Reading

- About Switch—Microsoft Docs[30]
- About If—Microsoft Docs[31]
- About Assignment Operators—Microsoft Docs[32]
- About Operator Precedence—Microsoft Docs[33]
- About Operators—Microsoft Docs[34]
- PowerShell Parser Tokenizer Source Code—GitHub[35]
- About Wildcards—Microsoft Docs[36]

[30] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_switch

[31] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_if

[32] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_assignment_operators

[33] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_operator_precedence

[34] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_operators

[35] https://github.com/PowerShell/PowerShell/blob/master/src/System.Management.Automation/engine/parser/tokenizer.cs

[36] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_wildcards#long-description

9. Logging

On the surface, the subject of logging seems simple. It could be as straightforward as "saving output for later use or review." It is, however, much more nuanced—and important—than many realize.

This chapter covers the basics of logging in PowerShell. It lays the groundwork for establishing some best practices that most developers and engineers should implement. The chapter aims to combine the latest information available for a range of logging options and present it to you in a single resource.

> At the time of writing, *Windows PowerShell* is at version 5.1, and *PowerShell* is at version 7.2. These version numbers aren't referenced here. Rather, the edition names are used when there are differences in functionality between editions.

There are several well-known logging options for Windows PowerShell.[1] However, with PowerShell now available—a cross platform solution—there are more differences that need to be addressed. Methods commonly employed in the past may not be the best solution in the present and future. You may even find that approaches you've taken in the past with Windows PowerShell are no longer available in PowerShell.

There are now technical considerations needed with different versions and platforms:[2]

- Differences in logging between Windows PowerShell and PowerShell.
- Differences in logging across Windows, WSL (Windows Subsystem for Linux), Linux, Mac, and CloudShell.
- There are many cloud shells and other serverless options available.

The two categories of logging, *system-level* and *on-demand.* **Both** provide a way to track the actions of your code. Both options are presented ahead and you can use either, or both, based on your needs.

The chapter presents simple use cases as examples. Note that these are simplistic and intended to show the ways you can use logging. These aren't scripts that you would use in production.

[1] Microsoft. (2022, Mar. 19). *About Logging (Microsoft.PowerShell.Core).* Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_logging. [Accessed: Aug. 03, 2022].

[2] Microsoft. (2022, May. 16). *Differences between Windows PowerShell 5.1 and PowerShell 7.x.* Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/scripting/whats-new/differences-from-windows-powershell?view=powershell-7.2. [Accessed: Aug. 03, 2022].

9.1 Why Log?

One of the first question to ask is *why* you're logging.

There are a variety of reasons for doing this and these change the logging methods you select:

- Are you logging output to help debug your code?
- Do you need a history you can refer to that shows what actions were taken when?
- Is there an audit requirement for compliance?

Logging all code run on a system for security is a common security requirement. You may also need to have JEA session logs available for review. You can learn more about JEA in the Just Enough Administration chapter.

9.2 What Makes for Good Logging

As you create custom logging for your code, think about the level of detail you want to capture. If you're writing code that will only affect you, minimal logging may be acceptable.

It's useful to contemplate who else will review the logs. Ensure you take the reviewer's competency level into account for the level of data you're logging. Be consistent in the style and level of logging you perform across all your code.

So, where should you store the data? Logging locally to where your code lives is a straightforward answer. However, you must ask yourself if others need to review the output. Consider if you should centralize it for monitoring and alerting based on the output.

When writing output, try to find a balance between human readability and consumption by an event monitoring solution. Logs should be readable enough for reviewers to identify errors and target details quickly. They should also be consistently formatted, such that software can parse and process them without complex analysis or extensive pattern-matching.

Always consider the purposes of the log. Are you creating data only for troubleshooting, or is it capturing the day-to-day operation of your software? Implement your logging such that you can easily change it when new requirements come in for auditing, statistics, or tracking performance.

One outcome to avoid is logging everything to a point where the output is overwhelming and unusable. Capture enough context so a reviewer can understand what created the message and why.

While it may seem excessive to account for so many factors, it's worth your time to go through and consider them. Getting this right up front could save hours of time for people looking through your logs and minimize any rework required by you to adjust the level of output you've created.

As a minimum, the log data you create should provide answers to these questions:

- What was done?
- Where was it done?

- When was it done?
- Who did it?

For high-impact changes, such as writing code to change users' access in AD (Active Directory) or Azure, your output should be comprehensive enough that a person or system could identify and undo all the changes.

When working with live data that affects the ability of others to work, taking this extra effort and time to create detailed output can reduce the impact if something goes wrong. You should also test the output to make sure you could use it to revert a change in practice.

9.3 What Should Never Be Logged

As you consider what to log, you should also note there are many items that shouldn't be logged:

- Credentials
- Passwords
- Access tokens
- API keys
- Encryption keys
- Personally identifiable information
- Social Security or national identity numbers
- Credit card numbers
- Any other data protected by laws such as GDPR (General Data Protection Regulation)[3]

System-level logging may be configured in such a way that it will capture all on-screen input and output. No matter what logging options you select, it's important to keep secrets out of your code, and know when user input may expose secrets within your logs.

In most cases, system-level logging prevents properly identified secrets from being captured. However, if your code is written such that the system doesn't recognize sensitive data, secrets may be stored in plain text in your logs. Ensure you understand how to identify sensitive data and prevent its accidental capture.[4]

The chapter discusses ways to protect sensitive data stored in logs later. However, following best practices and preventing it from being logged at the start is the best solution.

While outside the scope of logging, the use of the SecretManagement[5] Module and Azure KeyVault[6] would help ensure this data is stored and protected properly. There's an informative Microsoft DevBlogs article[7] for getting up to speed on this module.

[3] Proton AG. (2022, May. 26). *What is GDPR, the EU's new data protection law?*. GDPR.EU. [Online]. Available: https://gdpr.eu/what-is-gdpr/. [Accessed: Aug. 29, 2022].

[4] Microsoft. (2022, Mar. 18). *About_Logging (Microsoft.PowerShell.Core) - Protected Event Logging*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_logging?view=powershell-5.1#protected-event-logging. [Accessed: Aug. 03, 2022].

[5] https://www.powershellgallery.com/packages/Microsoft.PowerShell.SecretManagement

[6] https://learn.microsoft.com/en-us/azure/key-vault/general/basic-concepts

[7] https://devblogs.microsoft.com/powershell/secretmanagement-and-secretstore-are-generally-available/

9.4 Logging Basics

Because of PowerShell's ease of use, it was popular with *red teams* and attackers, and used in many environments before 2015. In response, Microsoft posted a DevBlogs article[8] detailing the improvements to PowerShell logging that engineers and developers use today.

By the release of Windows PowerShell 5, Microsoft had added a multitude of logging features. This was enough that some prominent *red teamers* posted blogs stating that PowerShell had too much tracking and was too easily detected to continue to be their primary method for *living off the land*. They changed some tools of their trade to C#, as the system-level logging there still isn't as robust.[9] [10]

While the ability to log is now best-in-class, the enablement of such logging and the review of logs may still trail behind.

9.5 Enable System-Level Logging

The section starts with system-level logging, as you can leverage this configuration in other methods discussed later in the chapter.

9.5.1 Windows

There are a few options to enable logging in Windows.[11] For enterprise systems, the enablement should be completed by your IT or security department, which has historically been deployed via Group Policy or Intune. For a standalone system, you can use the Local Group Policy Editor[12] (`gpedit.msc`) to access the same settings. Once the Local Group Policy Editor is open, find the following configuration path:

```
Computer Configuration\
  Administrative Templates\
    Windows Components\
      Windows PowerShell
```

Change the following four group policy settings:

- **Turn on Module Logging**: Enabled. Use `*` in the *Module Names* selection to apply logging to all modules, or add each module you wish to log.
- **Turn on PowerShell Script Block Logging**: Enabled.

[8] https://devblogs.microsoft.com/powershell/powershell-the-blue-team/

[9] SecTor. (2019, Oct. 09). *Powershell is Dead. Long Live C# - Lee Kagan.* SecTor 2019. [Online]. Available: https://sector.ca/sessions/powershell-is-dead-long-live-c/. [Accessed: Aug. 29, 2022].

[10] BSides Scotland. (2019, Apr. 23). *Powershell Is DEAD – Epic Learnings! - Ben Turner and Doug McLeod - BSides Scotland 2019.* YouTube. [Online]. Available: https://www.youtube.com/watch?v=PPDUU3ObX88. [Accessed: Aug. 29, 2022].

[11] Microsoft. (2022, Mar. 18). *About Group Policy Settings (Microsoft.PowerShell.Core).* Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_group_policy_settings. [Accessed: Aug. 29, 2022].

[12] https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/dn789185(v=ws.11)

  - Check the box for **Log script block invocation start/stop events**.

- **Turn on Script Execution**: Enabled.
  - Set **Execution Policy** to *Allow local scripts and remote signed scripts*. This is the default for Windows 10+ and Windows Server 2016+.
- **Turn on PowerShell Transcription**: Disabled. This is because of the likelihood of capturing plain text secrets if best practices aren't followed. With this set, you'll still be able to enable transcription temporarily as needed.

Another setting only used by enterprises is found in the Local Group Policy Editor, at the following configuration path:

```
Computer Configuration\
  Administrative Templates\
    Windows Components\
      Event Logging\
        Enable Protected Event Logging
```

This setting requires PKI (Public Key Infrastructure) certificates to encrypt and decrypt sensitive data being written to the Windows Event Logs. You wouldn't use this for standalone systems, as the private key shouldn't be available on the computer where the logs are created and encrypted. Instead, the private key would live only in the location where the logs are being collected and decrypted.

An event log consolidation solution like this is highly encouraged in enterprises, but is outside the scope of this chapter.

9.5.2 Event Log Locations

PowerShell-specific logs on Windows are written to the following default locations:[13]

Windows PowerShell:

```
Application and Services Logs\
  Windows PowerShell

Application and Services Logs\
  Microsoft\
    Windows\
      PowerShell\
        Operational
```

PowerShell:

[13] Microsoft. (2022, Mar. 18). *About Eventlogs (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_eventlogs?view=powershell-5.1. [Accessed: Aug. 29, 2022].

```
Application and Services Logs\
  PowerShell Core\
    Operational
```

These are viewable in the Windows Event Viewer[14] or can be queried from PowerShell or other tools.

9.6 Linux, macOS, WSL

According to Microsoft documentation,[15] the logging configuration for non-Windows systems is managed via a JSON (JavaScript Object Notation) configuration file named `PowerShell.config.json` and stored in one of two locations:

- System-wide settings are stored in `$PSHome`. This is often `/opt/microsoft/PowerShell/7/`.
- User-specific settings are stored in the same directory as a user's PowerShell profile. Use `Split-Path $PROFILE.CurrentUserCurrentHost` to find this folder.

If the configuration file doesn't exist, create it using your preferred text editor, and copy the sample configuration. Be sure to change the paths to suit your needs.

The configuration options available within the JSON files are the same as those covered for Windows above:

Example 1: PowerShell.config.json configuration file for non-Windows systems

```
{
    "Microsoft.PowerShell:ExecutionPolicy": "RemoteSigned",
    "PowerShellPolicies": {
        // Equivalent: Turn on Script Execution
        "ScriptExecution": {
            "ExecutionPolicy": "RemoteSigned",
            "EnableScripts": true
        },
        // Equivalent: Turn on PowerShell Script Block Logging
        "ScriptBlockLogging": {
            "EnableScriptBlockInvocationLogging": true,
            "EnableScriptBlockLogging": true
        },
        // Equivalent: Turn on Module Logging
        "ModuleLogging": {
            "EnableModuleLogging": false,
            "ModuleNames": [
                // Use "*" to log all modules
                "PSReadline",
                "PowerShellGet"
            ]
        },
        // Equivalent: Enable Protected Event Logging
        "ProtectedEventLogging": {
```

[14] https://learn.microsoft.com/en-us/shows/inside/event-viewer

[15] Microsoft. (2022, Mar. 18). *About PowerShell Config (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_powershell_config. [Accessed: Aug. 26, 2022].

```
            "EnableProtectedEventLogging": false,
            "EncryptionCertificate": [""]
        },
        // Equivalent: Turn on PowerShell Transcription
        "Transcription": {
            "EnableTranscripting": true,
            "EnableInvocationHeader": true,
            "OutputDirectory": "\\tmp\\new"
        },
        // Other settings
        "UpdatableHelp": {
            "DefaultSourcePath": "\\temp"
        },
        "ConsoleSessionConfiguration": {
            "EnableConsoleSessionConfiguration": false,
            "ConsoleSessionConfigurationName": "name"
        }
    },
    "LogLevel": "verbose"
}
```

The following additional configuration option is also noted for macOS:

```
log config --subsystem com.microsoft.powershell --mode=persist:info,level:info
```

9.7 Logging for Troubleshooting

This section covers logging specifically for troubleshooting or debugging.

9.7.1 Writing Console Output

- **Pro**: Quick and easy
- **Pro**: Flexible output options
- **Pro**: Cross platform support with minor changes
- **Con**: Alone, this output isn't persistent

As you're writing code, a common troubleshooting method is to write a status message to the console to see results using `Write-Host` or `Write-Output`. This is a quick-and-dirty option; however, there are better ways to perform this simple task. As you probably know, PowerShell has multiple output streams available.[16] [17] The following write cmdlets are available and provide better solutions than simply sending your output to the console during each run:

- **Write-Verbose**[18]: Show events only when running with the `-Verbose` parameter.

[16] Microsoft. (2022, Mar. 18). *About Output Streams (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_output_streams. [Accessed: Aug. 27, 2022].

[17] Microsoft. (2022, Mar. 18). *About Redirection (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_redirection. [Accessed: Aug. 29, 2022].

[18] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/write-verbose

- **Write-Debug**[19]: Show events only when running with the `-Debug` parameter.
- **Write-Information**[20]: Add informational messages to your output.
- **Write-Warning**[21]: Adds a warning message to your output.
- **Write-Error**[22]: Declares a non-terminating error, and adds it to the error stream.
- **Write-Progress**[23]: Displays a progress bar in the PowerShell console.
- **Write-Host**[24]: Writes customized output to a host and terminates the pipeline (no longer kills a puppy[25]).

By default, these cmdlets only write to the console and provide no permanent storage of the output.

9.8 Persistent Logging Options

Ahead are some approaches for creating more persistent logging data.

9.8.1 PowerShell Transcription

PowerShell transcription records all commands entered, and all output produced. This data is written into plain text files. There are two basic commands to control this (aside from the system-level controls noted earlier), which are self-explanatory and can be used with or without additional parameters for file names and locations.

Example 2: Starting and stopping transcription in PowerShell

```
Start-Transcript
Stop-Transcript
```

Transcription is also stopped when closing the PowerShell console.

You can read more about transcription on the Microsoft Docs page for Start-Transcript[26].

[19] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/write-debug

[20] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/write-information

[21] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/write-warning

[22] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/write-error

[23] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/write-progress

[24] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/write-host

[25] The phrase ***“Every time you use Write-Host, you kill a puppy…”*** was coined by PowerShell MVP Don Jones. Around 2013, both he and the father of PowerShell, Jeffrey Snover, advocated for limited use of this cmdlet as it negatively impacted automation. Many people were using `Write-Host` to convey results or information to the user. `Write-Host` had some limited use cases, but was considered the wrong tool in the toolbelt for many situations because it didn’t write to any of the available output streams. Starting with PowerShell 5.0, `Write-Host` is now just a wrapper for `Write-Information`. The etymology of this phrase could be linked back to a famous line from the 1946 classic movie *It’s a Wonderful Life* when the character Zuzu Bailey, daughter of the protagonist, George Bailey, proclaims at the end of the movie, ***“Look, Daddy! Teacher says ‘every time a bell rings, an angel gets his wings.’“***.

[26] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.host/start-transcript

9.8.2 Logging to Files

- **Pro**: Quick and easy
- **Pro**: Flexible output options
- **Pro**: Cross-platform support with minor changes
- **Con**: A custom solution may not be easily discoverable or usable by other automation

If you want to add data to existing log files without overwriting them, don't forget to use the `-Append` switch where noted below. The cmdlets available are:

- **Out-File**[27]: Writes the output of a pipeline to a file.
- **Export-Csv**[28]: Exports data in the common CSV (comma-separated values) format.
- **Add-Content**[29]: Appends text to a file.
- **Set-Content**[30]: Writes text to a file, overwriting any existing content.

Using `Out-File` is the simplest option.

As an example, say you're tasked with daily tracking of all services on your systems set to start automatically, but have stopped. You could create a separate file, including the date and time it was created as a part of the file name, for each run with:

Example 3: Logging all stopped automatic-start services to a dated file

```
$FormattedDate = Get-Date -Format 'yyyyMMddTHHmm'
Get-Service |
    Where-Object {
        ($_.Status -eq 'Stopped') -and ($_.StartType -eq 'Automatic')
    } |
    Out-File "C:\temp\StoppedServices-$FormattedDate.txt"
```

When using `Export-Csv`, it's best to create PSCustomObjects[31] before attempting to write the data.

You could use an approach similar to the following to collect the required data and keep a running log in a single file:

[27] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/out-file
[28] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/export-csv
[29] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.management/add-content
[30] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.management/set-content
[31] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_pscustomobject

Example 4: A more advanced logging strategy keeping a tally in a single file

```
$FormattedDate = Get-Date -Format 'yyyyMMddTHHmm'
$Count = (
    Get-Service | Where-Object {
        ($_.Status -eq 'Stopped') -and ($_.StartType -eq 'Automatic')
    }
).Count
$MyCustomObject = [PSCustomObject]@{
    DateTime = $FormattedDate
    Count    = $Count
}
$MyCustomObject |
    Export-Csv 'C:\temp\StoppedServices.csv' -NoTypeInformation -Append

Get-Content 'C:\temp\StoppedServices.csv'
```

```
"DateTime","Count"
"20210504T1933","8"
```

Depending on your use case, there are other options available. You can use `Set-Content` to replace all the content of an existing file. You can use `Add-Content` to *append* content to a file, without removing its existing content.

Both cmdlets create the file if it doesn't exist. You can use `Set-Content` to empty an existing file and then `Add-Content` to append additional data to it. This is useful in situations where old log data can be safely cleared to reduce filesystem space utilization.

Your use case and personal preferences will determine which cmdlets you use.

Example 5: Optionally clearing old log data

```
$FormattedDate = Get-Date -Format 'yyyyMMddTHHmm'
$Count = (
    Get-Service | Where-Object {
        ($_.Status -eq 'Stopped') -and ($_.StartType -eq 'Automatic')
    }
).Count

$Params = @{
    Path  = 'C:\temp\test.csv'
    Value = "$FormattedDate,$Count"
}
if ($ClearLog) {
    $Params.Value = @('"DateTime","Count"', $Params.Value)
    Set-Content @Params
}
else {
    Add-Content @Params
}
```

In the example, a log file is overwritten if `$ClearLog` is `$true`. Otherwise, a new entry is appended to the end of the file.

9.8.3 Using Tee-Object

The Tee-Object[32] cmdlet is used to send data both to a file and display the output in the console:

[32] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/tee-object

Example 6: Using Tee-Object to write information to the console and a file simultaneously

```
$FormattedDate = Get-Date -Format 'yyyyMMddTHHmm'
$Count = (
    Get-Service | Where-Object {
        ($_.Status -eq 'Stopped') -and ($_.StartType -eq 'Automatic')
    }
).Count
"$FormattedDate,$Count" | Tee-Object -Path 'C:\temp\test.csv' -Append
```

This is useful when you want to monitor the work being done, but you also want to keep a persistent log.

9.9 History

PowerShell has two native history providers that record the list of commands run, but not their output.[33]

The built-in history is only available within the current PowerShell session. It isn't persistent and not available from other open sessions.

9.9.1 Built-in History

To view this history, use Get-History[34]:

Example 7: Using Get-History to retrieve session command history

```
# Example 7a: Get all session command history
Get-History

# Example 7b: Get the last 2 commands
Get-History -Count 2

# Example 7c: Get the 2nd command from the start of the session
Get-History -Id 2
```

```
# Example 7a:
  Id     Duration CommandLine
  --     -------- -----------
   1        0.145 Start-Transcript…
   2        0.368 $FormattedDate = Get-Date -Format 'yyyyMMddTHHmm'…
   3        0.276 $FormattedDate = Get-Date -Format 'yyyyMMddTHHmm'…
   4        0.283 $FormattedDate = Get-Date -Format 'yyyyMMddTHHmm'…
   5        0.258 $FormattedDate = Get-Date -Format yyyyMMddTHHmm…

# Example 7b:
  Id     Duration CommandLine
  --     -------- -----------
   5        0.258 $FormattedDate = Get-Date -Format yyyyMMddTHHmm…
```

[33] Microsoft. (2022, Mar. 18). *About History (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_history. [Accessed: Aug. 29, 2022].

[34] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/get-history

```
   6        0.015 Get-History

# Example 7c:
  Id     Duration CommandLine
  --     -------- -----------
   1        0.145 Start-Transcript…
```

You can use standard PowerShell parsing to find specific commands that have run and use `Export-Csv` to store the history if you desire.

For example, to find all commands in your history that include ‘UserName’, you could run:

Example 8: Finding commands with the term ‘UserName’

```
Get-History |
    Where-Object CommandLine -Like '*UserName*' |
    Export-Csv 'C:\temp\history.csv'
```

9.9.2 PSReadline History

PSReadline is a cross-platform module that stores information on disk, so it’s available to all sessions on your system.[35] Unlike the built-in history, PSReadline history is kept between sessions. You can also use the module’s command and argument completion, called *Predictive IntelliSense*.[36] To find the path where PSReadLine stores command history, use:

Example 8: Finding commands with the term ‘UserName’

```
(Get-PSReadlineOption).HistorySavePath
```

```
C:\Users\User\AppData\Roaming\Microsoft\Windows\PowerShell\PSReadLine\
ConsoleHost_history.txt
```

Command history on Linux is stored in the following location:

```
/home/<USERNAME>/.local/share/powershell/PSReadLine/ConsoleHost_history.txt
```

Command history on Windows is located here:

[35] Microsoft. (2022, Jul. 25). *About PSReadLine*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/-module/psreadline/about/about_psreadline. [Accessed: Aug. 29, 2022].

[36] Microsoft. (2022, Aug. 17). *Using predictors in PSReadLine*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/scripting/learn/shell/using-predictors. [Accessed: Aug. 29, 2022].

```
C:\Users\<USERNAME>\AppData\Roaming\Microsoft\Windows\PowerShell\PSReadLine\
ConsoleHost_history.txt
```

These files provide logging of the commands run, but not their output.

It's possible to clean the history file. You would do this to remove typos or mistakes so the autocompleted items in your history are correct, and don't cause future confusion or problems.

If you accidentally enter secrets in plain text as part of a command, you'll have to edit this file to remove them from history. Many engineers and developers have accidentally typed passwords into username fields or started typing a second command, not realizing there was already text in the console. Being able to fix mistakes like these can be done quickly and easily.

If you're using VS Code (Visual Studio Code), this command opens the history file for review and editing:

Example 9: Opening the PSReadline history file in VS Code

```
code (Get-PSReadlineOption).HistorySavePath
```

As an example, here's a typo from the history file:

```
clsGet-Process -IncludeUserName | Where-Object { $null -ne $_.UserName } |
Select-Object Name,Username
```

Once in the history, this will be the first highlighted option shown by Predictive IntelliSense when you type `cls`. To fix issues like this, open the history file, use the *find* command to locate the typo, delete the row containing the typo, and save the file. Once those steps are completed, and you start a new session, the IntelliSense suggestion will no longer be presented.

9.9.3 Writing to Windows Event Logs

- **Pro**: Logs can be pulled to a central location using existing tooling already deployed for log management.
- **Pro**: After registering a provider, you can write to the *Application* log without administrative rights.
- **Con**: You must perform a one-time configuration step, requiring administrative rights (elevation), before using this approach.
- **Con**: No cross-platform support.
- **Con**: Only available in Windows PowerShell.

Be sure to validate the event log retention settings if you're going to rely on this solution.

Ahead is a simple example function to show how you can write data to Windows Event logs. While you can create custom event logs, using the existing *Application* log and registering a custom provider allows data to be handled by existing event collection engines without additional modifications.

Creating your own message and category resource files for your event IDs and categories will also simplify validation and alerting.[37]

Example 10: Writing to the Windows event log in Windows PowerShell

```powershell
function Write-PS5Event {
    param (
        [Parameter(Mandatory = $true)]
        $Message,
        [Parameter(Mandatory = $true)]
        $EventID,
        [Parameter(Mandatory = $true)]
        [ValidateSet('Warning', 'Error', 'Information')]
        $EventLevel
    )

    $SourceName = 'MyPS5Log'
    $ErrorActionPreference = 'Stop'

    # Create the EventLog Source
    if (-not [System.Diagnostics.EventLog]::SourceExists($SourceName)) {
        Write-Verbose (
            'EventSource was missing, attempting to add. This will fail' +
            ' if not running in an elevated PowerShell session'
        )
        New-EventLog -LogName Application -Source $SourceName
    }

    Write-Verbose $Message
    $EventLogParams = @{
        LogName   = 'Application'
        Source    = $SourceName
        EventID   = $EventID
        EntryType = $EventLevel
        Message   = $Message
        Category  = $EventID
    }
    Write-EventLog @EventLogParams
}
```

The function first checks if an event source with the name ‘MyPS5Log’ exists and, if not, creates it. This step requires elevation and fails otherwise. It then writes an event to the *Application* event log using the source name.

9.9.4 Cloud Shell

Once configured, Azure Cloud Shell[38] has access to persistent storage.[39] This storage can store script files and hold the output of the scripts.

[37] Microsoft. (2021, Jul. 01). *Message Files*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/eventlog/message-files. [Accessed: Aug. 29, 2022].

[38] https://learn.microsoft.com/en-us/azure/cloud-shell/overview

[39] Microsoft. (2022, Mar. 01). *Persist files in Azure Cloud Shell*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/azure/cloud-shell/persisting-shell-storage. [Accessed: Aug. 29, 2022].

Cloud Shell configuration is beyond the scope of this chapter, but you can read more in the Persisting Shell Storage[40] article on Microsoft Docs.

You can use this storage in the same way as any local storage. For example, to list all *Azure Resource Group* names in your subscription, you can run:

Example 11: Displaying Azure Resource Group Names

```
Get-AzResourceGroup |
    Select-Object ResourceGroupName |
    Out-File ./output/ResourceGroups.txt
```

To create a list of all Azure AD users, and write the output to a file, you could run the following in Cloud Shell:

Example 12: Logging an Azure AD user list to a file

```
Connect-AzureAD

Get-AzureADUser |
    Select-Object UserPrincipalName, DisplayName |
    Out-File ./output/AzureUserList.txt
```

To see the output, you can access it as you would any other file. To do so from the shell itself, use `Get-Content` to read the file into the console:

Example 13: Reading logs directly in Azure Cloud Shell

```
# Example 13a: Displaying resource groups list
Get-Content ./output/ResourceGroups.txt

# Example 13b: Displaying AD users list
Get-Content ./output/AzureUserList.txt
```

```
# Example 13a:
ResourceGroupName
-----------------
MyTestApp

# Example 13b:
UserPrincipalName              DisplayName
-----------------              -----------
JaneDoe@contoso.onmicrosoft.com Jane Doe
```

You could also open it in the web version of VS Code if you need to change it as well:

[40] https://learn.microsoft.com/en-us/azure/cloud-shell/persisting-shell-storage

Example 14: Opening logs in VS Code for the Web

```
code ./output/ResourceGroups.txt
code ./output/AzureUserList.txt
```

You can see from these examples that you can specify a relative path in Cloud Shell. Without specifying the path, file names are relative to the current directory, as in other shells.

If you desire, you can also enable transcription logs for your Cloud Shell sessions. While you can do this manually as you would in any local session, you can also create a profile that will start transcription with the launch of any future session.

From within a Cloud Shell session, edit your profile by opening the *current user all hosts* profile file with VS Code for the Web.

Example 15: Opening the PowerShell profile for modification in Cloud Shell

```
code $PROFILE.CurrentUserAllHosts
```

Add the following two lines to your profile to create a new transcription log file each time you launch Cloud Shell.

Example 16: Code for the Cloud Shell profile to start transcription

```
$Now = Get-Date -Format 'yyyyMMddTHHmm'
Start-Transcript "./Transcripts/Transcript-$Now.txt"
```

If the path you enter doesn’t exist, `Start-Transcript` creates intermediate folders automatically.

9.9.5 Using Third Party modules for logging

The primary emphasis in this chapter is covering the built-in tools available within the different editions of PowerShell. However, there are third party modules that can be configured to hide much of the complexity involved in creating logs.

The *PSFramework* module includes the ability to simplify logging as just one of its many features.[41]

Going in depth with this module would require an entire chapter of its own, and there have been multiple conference sessions on the subject. You can find out more about the PSFramework module on the PSFramework Website[42].

[41] PowerShell Framework Collective. (2021, Jan. 18). *PowerShell Framework - The Logging System*. PowerShell Framework Documentation. [Online]. Available: https://psframework.org/documentation/documents/psframework/logging.html. [Accessed: Aug. 28, 2022].

[42] https://psframework.org/

9.10 Summary

This has been a relatively brief chapter, but you should now have a better grasp of the logging options available to you in PowerShell. Start with the questions on why you're logging the data—and who'll consume it—to help you choose the options you need to meet your logging requirements.

Understand that several options have changed from Windows PowerShell to PowerShell. Conversely, several options work in the same way regardless of the edition and platform on which they're run.

Finally, realize that there are exceptional third-party modules available that may have already solved the problem in front of you.

9.11 Further Reading

- Script Tracing and Logging—Microsoft Docs[43]
- About Event Logs (Windows PowerShell 5)—Microsoft Docs[44]
- About Windows Logging (Windows PowerShell 5)—Microsoft Docs[45]
- About Windows Logging—Microsoft Docs[46]
- About Non-Windows Logging—Microsoft Docs[47]
- About History—Microsoft Docs[48]
- About PSReadLine—Microsoft Docs[49]
- About PowerShell Config—Microsoft Docs[50]
- PSReadLine—PowerShell Gallery[51]
- PSFramework—PowerShell Gallery[52]
- SecretManagement—PowerShell Gallery[53]

[43] https://learn.microsoft.com/en-us/powershell/scripting/windows-powershell/wmf/whats-new/script-logging
[44] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_eventlogs?view=powershell-5.1
[45] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_logging?view=powershell-5.1
[46] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_logging_windows
[47] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_logging_non-windows
[48] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_history
[49] https://learn.microsoft.com/en-us/powershell/module/psreadline/about/about_psreadline
[50] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_powershell_config
[51] https://www.powershellgallery.com/packages/PSReadLine
[52] https://www.powershellgallery.com/packages/PSFramework
[53] https://www.powershellgallery.com/packages/Microsoft.PowerShell.SecretManagement

10. Infrastructure as Code (IaC)

This chapter covers the concepts, technical elements, and benefits of IaC. It also provides common guidance for developing your IaC artifacts. It introduces the concept of Configuration as Code (CaC) with PowerShell Desired State Configuration (DSC) and its use cases.

> IaC and PowerShell aren't restricted to a specific platform.[1] However, considering PowerShell is a Microsoft product, the examples used in this chapter demonstrate the provisioning and configuration of resources in Microsoft Azure. You can create a free trial account[2] in Microsoft Azure.

10.1 Overview

Historically, datacenters' infrastructure was built by manual processes and the use of configuration tools requiring human interaction.[3] However, with the rise of virtualization and the advent of cloud computing, scalability became an issue. Large infrastructure deployments were labor-intensive, inefficient, and plagued with configuration drift and human error. For this reason, the concept of Infrastructure as Code was born as a method to solve these problems and automate the deployment of infrastructure resources.

Imagine the sizes of Microsoft, Google, and Amazon cloud infrastructures and the challenges they would face if they were to provision and configure all their infrastructure manually. You can't because it would be impossible and would not scale. IaC is one of many concepts engineers envisioned to solve those problems and make large-scale deployments feasible.

It's important to note that IaC isn't only for cloud computing. You can apply the same concept to virtual servers or physical hardware hosted on-premises.

10.2 IaC Key Concepts

IaC is the DevOps practice of defining configuration files, usually stored in source control, to automate the provisioning of your infrastructure. These configuration files fall into two distinct categories.[4]

- **Imperative IaC**: You make use of scripts—such as PowerShell and Bash scripts—to define a series of steps to provision the infrastructure.

[1] Microsoft. (2021, Jun. 29). *What is Infrastructure as Code?*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/devops/deliver/what-is-infrastructure-as-code. [Accessed: Jul. 09, 2022].

[2] https://azure.microsoft.com/en-us/free/

[3] Rendón, D. (2022, Jan.). *Why Infrastructure as Code?*. In: Building Applications with Azure Resource Manager (ARM). Berkeley, CA: Apress. ISBN: 978-1-4842-7747-8.

[4] Microsoft. (2021, Jun. 29). *What is Infrastructure as Code?*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/devops/deliver/what-is-infrastructure-as-code. [Accessed: Jul. 09, 2022].

- **Declarative IaC**: You declare how the infrastructure should be and tools—such as Azure Resource Manager (ARM), Amazon Web Services (AWS) CloudFormation, and Terraform—take care of the outcome.

This chapter focuses on **Imperative IaC** as it relates to PowerShell and **Declarative IaC** as it relates to PowerShell DSC.

It's also important to define the concepts of idempotency and immutability as they're important to IaC.

- **Idempotency**: An infrastructure resource is idempotent when, independently of how many times you deploy it, the results are always the same.
- **Immutability**: An immutable infrastructure resource never changes. If the resource needs to be patched or tweaked, it's destroyed and redeployed.

Declarative IaC is the most common implementation of IaC. In the examples used in this chapter, the infrastructure would most likely be deployed using tools such as Azure ARM templates, Azure Biceps, or Terraform.

10.3 IaC Benefits

IaC brings several benefits for organizations that implement it, ranging from cost savings to increased scalability. The following are some examples:

- **Scalability**: IaC introduces a high level of automation which allows you to provision entire infrastructure stacks as code.
- **Cost**: IaC reduces the effort required to deploy, maintain and troubleshoot infrastructure resources, thus decreasing the operational cost.
- **Consistency**: You can provision entire environments—such as non-production and production—and be confident they're the same because they're deployed using the same code.
- **Speed**: You can deploy infrastructure resources in minutes if not seconds as opposed to doing it manually, which could take hours or even days.
- **CI/CD**: You can integrate your scripts and configuration files with your continuous integration/continuous delivery pipelines to deploy entire applications.
- **Source Control**: All your configuration files can be stored in source control, which gives you the ability to version your infrastructure, providing easy roll-back capabilities.

10.4 IaC Principles

To get all the benefits from IaC, follow these principles of Infrastructure as Code.

10.4.1 Source Control as the Single Source of Truth

Store all your scripts and configuration files in a single repository. Change your infrastructure resources by changing the code and committing to source control. To keep your source control system as the single source of truth, only allow changes to the infrastructure from accounts used in your CI/CD system. This principle is aligned with the key concept of idempotency.

To learn more about source control, see the Git chapter.

10.4.2 Modular

You shouldn't define all your resources in a single script or configuration file because, in the long term, they'll be hard to maintain. Instead, break your code into modules. Modularizing your code makes it easier to maintain, increases readability, and allows it to be independently deployed.

Once you have modules defined, you can build blueprints that are a combination of modules for a specific deployment pattern. Because of the modular approach, each module can be updated to a newer version without impacting the blueprint or deployments that use previous versions of the module.

For example, say you create a new pattern that comprises a load-balanced two-tier web application. You can then bring to life that pattern by creating a blueprint with the required modules. You would create a blueprint and include the modules for the web application and the load balancer. You can use those same modules in other patterns.

10.4.3 Versioning

The versioning and modular principles are aligned. Every time you update a module, you should also update its version. When you version your modules, you can identify changes between versions that may create unexpected issues in your infrastructure and roll back to an older version if required. By versioning your modules, when you update them to a newer version, you won't impact other teams that may already have utilized the previous version. Versioning also allows you to track what code changed and who made the changes.

10.4.4 Repeatable

The replacement of manual deployments with IaC and the abstraction of the infrastructure as code make the process less prone to human error and, most importantly, repeatable. This principle aligns with the concept of idempotency. This means you should be able to deploy a piece of code multiple times and always get the same results. Because it's repeatable, IaC gives you the confidence that what you've deployed in one environment—such as a development—will be the same when deployed in production.

10.4.5 Disposable

This principle aligns with the concept of immutability. IaC enables you to create, replace, and destroy resources. A classic example is the time wasted by systems admins troubleshooting issues with servers. When using IaC, if you encounter issues with servers, you can simply destroy and recreate them. Therefore, they're considered disposable.

10.4.6 Self-Documented

Infrastructure documentation is often outdated, either because people forget to update it or unauthorized changes weren't documented. When you write your infrastructure as code, it becomes a minimum set of documentation. You can refer to the code and understand what was deployed and how it was configured. In addition, when all your changes come from code modifications rather than manual infrastructure changes, the documentation is always kept up-to-date.

10.4.7 Testing and Monitoring

IaC is an intersection of development and operations and therefore should incorporate one of the core practices of development teams, which is automated testing. Automated testing is important when implementing IaC because small changes to the infrastructure can carry great impact. There's a variety of tests that can be performed for IaC such as unit testing, integration testing, and smoke tests. You can use Pester[5] to perform unit and integration tests on your PowerShell scripts. Refer to the **PowerShell Testing** part of the book for more details about Pester.

You use automated testing to ensure the quality of your code and that it meets your expectations. However, this isn't all. You should also monitor the infrastructure that the code deployed and continuously monitor your code to ensure that quality is maintained.

10.5 IaC in Action

Now that you've been introduced to the concepts, benefits, and principles of IaC, it's time to examine how you can use PowerShell to provision infrastructure resources.

Throughout the rest of this chapter, the examples work with a load balanced, two-tier web application that was mentioned in an earlier section. You're going to deploy a load balancer, two virtual machines, and one Azure SQL server to the Microsoft Cloud.

As covered in the IaC Key Concepts section, using PowerShell scripts to deploy infrastructure is **Imperative IaC**.[6] The scripts contain a series of commands which have to be executed for the infrastructure to reach the desired state.

> For brevity, the code for connecting to an Azure subscription isn't displayed. Azure Resource Manager modules are also required to execute the commands covered in this example. It's recommended you execute the commands in Azure Cloud Shell[7], as modules are pre-installed and connectivity to your Azure subscription is already established.

Following the principle of modularization, you'll build four modules and two scripts. One is a DSC configuration script and the other imports the modules and executes the required commands.

[5] Pester Team. (2022, Jun. 22). *Quick Start - Pester*. Pester Docs. [Online]. Available: https://pester.dev/docs/quick-start. [Accessed: Jul. 10, 2022].

[6] Microsoft. (2021, Jun. 29). *What is Infrastructure as Code?*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/devops/deliver/what-is-infrastructure-as-code. [Accessed: Jul. 09, 2022].

[7] https://learn.microsoft.com/en-us/azure/cloud-shell/overview

- **Azure-SQL-Server.psm1**: A module to deploy an Azure SQL instance.
- **Azure-Storage-Account.psm1**: A module to deploy an Azure Storage Account where your DSC scripts will be stored.
- **Azure-Load-Balancer.psm1**: A module to deploy an Azure Load Balancer with a public IP address.
- **Azure-Virtual-Machine.psm1**: A module to deploy Azure VMs to an existing network and add the VMs to the back-end pool of an Azure Load Balancer.
- **Deploy-WebServer.ps1**: A DSC configuration script that adds the Internet Information Services (IIS) Windows feature and modifies the default website start page to display the SQL database connection string.
- **Two-Tier-App-Blueprint.ps1**: The blueprint script that imports the four modules and calls the required functions which deploy the resources. You can find this later in the chapter, where it ties together all the concepts.

Each module ahead contains a single function. To simplify the provided examples, all function parameters in the modules are optional and include default values. Comments summarize what each function is doing.

Some examples ahead are very long, and they may be difficult to read across pages. You can find the scripts and modules[8] from this chapter in the Extras repository[9] for this book.

10.5.1 Azure-SQL-Server.psm1

Example 1: The `Azure-SQL-Server` module contains a function that creates a new Azure SQL server instance and database, and configures the firewall for access

```
function New-AzureSQLServer {
    param (
        [Parameter(Mandatory = $false)]
        [String]$RGName = 'MyApp',

        [Parameter(Mandatory = $false)]
        [String]$ServerName = 'myuniquesqlserver956x',

        [Parameter(Mandatory = $false)]
        [String]$DbName = 'mydb',

        [Parameter(Mandatory = $false)]
        [String]$Location = 'AustraliaEast',

        [Parameter(Mandatory = $false)]
        [String]$StartIP = '0.0.0.0',

        [Parameter(Mandatory = $false)]
        [String]$EndIP = '0.0.0.0',

        [Parameter(Mandatory = $false)]
        [String]$User = 'sqladmin',

```

[8] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/tree/main/Edition-01/IaC/Scripts/
[9] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/tree/main/Edition-01/

```
        [Parameter(Mandatory = $false)]
        [String]$Password = 'MyC0mplexP@ssWord!'
    )

    ## Create Resource Group if it doesn't exist
    if (-not (Get-AzResourceGroup -Name $RGName -ea:si)) {
        $Rg = New-AzResourceGroup -Name $RGName -Location $Location
    }

    ## Create a username and password for the SQL server.
    ## You wouldn't have credentials in your code.
    ## This is for demonstration purposes only.
    $Pw = ConvertTo-SecureString $Password -AsPlainText -Force
    $Cred = New-Object PSCredential $User, $Pw

    ## Create SQL Server
    $SqlParams = @{
        ResourceGroupName            = $RGName
        ServerName                   = $ServerName
        Location                     = $Location
        SqlAdministratorCredentials = $Cred
    }

    $Server = New-AzSqlServer @SqlParams

    ## Create firewall rule allowing access from the specified IP range
    $FwRuleParams = @{
        ResourceGroupName = $RGName
        ServerName        = $ServerName
        FirewallRuleName  = 'AllowedIPs'
        StartIpAddress    = $StartIP
        EndIpAddress      = $EndIP
    }

    $ServerFirewallRule = New-AzSqlServerFirewallRule @FwRuleParams

    ## Create a blank database with an S0 performance level
    $DbParams = @{
        ResourceGroupName             = $RGName
        ServerName                    = $ServerName
        DatabaseName                  = $DbName
        RequestedServiceObjectiveName = 'S0'
        SampleName                    = 'AdventureWorksLT'
    }

    $Database = New-AzSqlDatabase @DbParams

    ## Return database connection string
    $ConnectionString = @(
        "Server=tcp:$ServerName.database.windows.net,1433;",
        "Database=$DbName;",
        "User ID=$User;",
        "Password=$Password;",
        "Trusted_Connection=False;",
        "Encrypt=True;"
    )

    return $ConnectionString

}
```

10.5.2 Azure-Storage-Account.psm1

Example 2: The **Azure-Storage-Account** contains a function that creates a new Azure storage account

```
function New-AzureStorageAccount {
    param (
        [Parameter(Mandatory = $false)]
        [String]$RGName = 'MyApp',

        [Parameter(Mandatory = $false)]
        [String]$StorageAccountName = 'myuniquestorage593x',

        [Parameter(Mandatory = $false)]
        [String]$Location = 'AustraliaEast'
    )

    ## Create Resource Group if it doesn't exist
    if (-not (Get-AzResourceGroup -Name $RGName -ea:si)) {
        $Rg = New-AzResourceGroup -Name $RGName -Location $Location
    }

    ## Create Storage Account
    $StorageAccountParams = @{
        Name              = $StorageAccountName
        ResourceGroupName = $RGName
        Location          = $Location
        SkuName           = 'Standard_LRS'
        Kind              = 'StorageV2'
    }

    $StorageAccount = New-AzStorageAccount @StorageAccountParams

    return $StorageAccount.StorageAccountName
}
```

10.5.3 Azure-Load-Balancer.psm1

Example 3: The **Azure-Load-Balancer** module contains a function that creates a new Azure load balancer and configures the necessary resources

```
function New-AzureLoadBalancer {
    param (
        [Parameter(Mandatory = $false)]
        [String]$RGName = 'MyApp',

        [Parameter(Mandatory = $false)]
        [String]$LbName = 'MyLb',

        [Parameter(Mandatory = $false)]
        [String]$Location = 'AustraliaEast'
    )

    ## Create Resource Group if it doesn't exist
    if (-not (Get-AzResourceGroup -Name $RGName -ea:si)) {
        $Rg = New-AzResourceGroup -Name $RGName -Location $Location
    }

    ## Create public ip and place in variable
    $PublicIPParams = @{
        Name              = "$LbName-pip"
        ResourceGroupName = $RGName
        Location          = $Location
        Sku               = 'Basic'
        AllocationMethod  = 'static'
    }

    $Pip = New-AzPublicIpAddress @PublicIPParams

    ## Create load balancer frontend configuration and place in variable
    $FeIPParams = @{
        Name            = 'fePool'
        PublicIpAddress = $Pip
    }

    $FeIP = New-AzLoadBalancerFrontendIpConfig @FeIPParams

    ## Create Backend address pool configuration and place in variable
    $BePool = New-AzLoadBalancerBackendAddressPoolConfig -Name 'bePool'

    ## Create the health probe and place in variable
    $ProbeParams = @{
        Name              = 'healthProbe'
        Protocol          = 'http'
        Port              = '80'
        IntervalInSeconds = '360'
        ProbeCount        = '5'
        RequestPath       = '/'
    }

    $HealthProbe = New-AzLoadBalancerProbeConfig @ProbeParams

    ## Create the load balancer rule and place in variable
    $LbRuleParams = @{
        Name                    = 'HTTPRule'
        Protocol                = 'tcp'
        FrontendPort            = '80'
        BackendPort             = '80'
        IdleTimeoutInMinutes    = '15'
        FrontendIpConfiguration = $FeIP
        BackendAddressPool      = $BePool
    }

    $Rule = New-AzLoadBalancerRuleConfig @LbRuleParams

    ## Create the load balancer resource
    $LoadBalancerParams = @{
        ResourceGroupName       = $RGName
        Name                    = $LbName
        Location                = $Location
        Sku                     = 'Basic'
        FrontendIpConfiguration = $FeIP
        BackendAddressPool      = $BePool
        LoadBalancingRule       = $Rule
        Probe                   = $HealthProbe
    }

    $Lb = New-AzLoadBalancer @LoadBalancerParams

    return $Pip.IpAddress

}
```

10.5.4 Azure-Virtual-Machine.psm1

Example 4: The **Azure-Virtual-Machine** module contains a function that creates and configures a new Azure Windows Server VM

```
function New-AzureVirtualMachine {
    param (
        [Parameter(Mandatory = $false)]
        [String]$RGName = 'MyApp',

        [Parameter(Mandatory = $false)]
        [String]$VmBaseName = 'MyVM',

        [Parameter(Mandatory = $false)]
        [int]$VMInstances = 2,

        [Parameter(Mandatory = $false)]
        [String]$LbName = 'MyLb',

        [Parameter(Mandatory = $false)]
        [String]$PoolName = 'BEPool',

        [Parameter(Mandatory = $false)]
        [String]$VnetName = 'VNet-01',

        [Parameter(Mandatory = $false)]
        [String]$VnetRGName = 'Connectivity',

        [Parameter(Mandatory = $false)]
        [String]$SubnetName = 'default',

        [Parameter(Mandatory = $false)]
        [String]$Location = 'AustraliaEast',

        [Parameter(Mandatory = $false)]
        [String]$User = 'iacadmin',

        [Parameter(Mandatory = $false)]
        [String]$Password = 'MyC0mplexP@ssWord!'
    )

    ## Create Resource Group if it doesn't exist
    if (-not (Get-AzResourceGroup -Name $RGName -ea:si)) {
        $Rg = New-AzResourceGroup -Name $RGName -Location $Location
    }

    ## Get vnet, subnet and Backend pool objects
    $VnetParams = @{
        Name              = $VnetName
        ResourceGroupName = $VnetRGName
    }

    $Vnet = Get-AzVirtualNetwork @VnetParams
    $Subnet = Get-AzVirtualNetworkSubnetConfig -VirtualNetwork $Vnet

    $BePoolParams = @{
        ResourceGroupName = $RGName
        LoadBalancerName  = $LbName
        Name              = $PoolName
    }

    $BePool = Get-AzLoadBalancerBackendAddressPool @BePoolParams

    ## Deploy Availability Set
    $AvSetParams = @{
        Location                  = $Location
        Name                      = "$VMBaseName-avset"
        ResourceGroupName         = $RGName
        Sku                       = 'Aligned'
        PlatformFaultDomainCount  = 2
        PlatformUpdateDomainCount = 2
    }

    $AvSet = New-AzAvailabilitySet @AvSetParams

    for ($i = 1; $i -lt $VMInstances + 1; $i++) {
        $Id = '{0:d3}' -f $i

        ## Create network interface
        $NicParams = @{
            ResourceGroupName              = $RGName
            Location                       = $Location
            Name                           = "$VmBaseName$id-nic"
            LoadBalancerBackendAddressPool = $BePool
            Subnet                         = $Subnet
        }

        $Nic = New-AzNetworkInterface @NicParams

        ## Create a username and password for the virtual machine.
        ## You wouldn't have credentials in your code.
        ## This is for demonstration purposes only.
        $Pw = ConvertTo-SecureString $Password -AsPlainText -Force
        $Cred = New-Object PSCredential $User, $Pw

        ## Create a virtual machine configuration
        $VmSize = 'Standard_DS1_v2'
        $Pub = 'MicrosoftWindowsServer'
        $Offer = 'WindowsServer'
        $Sku = '2019-Datacenter'

        $VmConfigParams = @{
            VMName            = "$VmBaseName$Id"
            VMSize            = $VmSize
            AvailabilitySetId = $($AvSet.Id)
        }

        $VMOSParams = @{
            Windows      = $true
            ComputerName = "$VmBaseName$Id"
            Credential   = $Cred
        }

        $VMSourceImageParams = @{
            PublisherName = $Pub
            Offer         = $Offer
            Skus          = $Sku
            Version       = 'latest'
        }

        $VMNicParams = @{
            Id = $Nic.Id
        }

        $VmConfig = New-AzVMConfig @VmConfigParams
        $VmConfig = $VmConfig | Set-AzVMOperatingSystem @VMOSParams
        $VmConfig = $VmConfig | Set-AzVMSourceImage @VMSourceImageParams
        $VmConfig = $VmConfig | Add-AzVMNetworkInterface @VMNicParams

        ## Create a virtual machine using the configuration
        $VmParams = @{
            ResourceGroupName = $RGName
            Location          = $Location
            VM                = $VmConfig
        }

        New-AzVM @VmParams
    }
}
```

At this point, your resources should be deployed. However, you still don't have your desired state because no configuration was performed. You still need to configure the servers as web servers and set the connection string to point to the SQL server. This is where Configuration as Code complements IaC.

> Because you aren't deploying a full application with source code, you'll just display the connection string on the home page so you understand how file manipulation works in DSC.

10.6 Configuration as Code (CaC)

CaC is a different concept from IaC. However, it falls under the umbrella of IaC, where CaC can be categorized as an enabler of IaC. CaC configures infrastructure resources after they're provisioned by IaC. If you want to provision a Windows virtual machine (VM) with IIS installed and configured, first use IaC to provision the VM and its dependent resources. You can then use CaC to install and configure IIS on that VM.

10.6.1 PowerShell Desired State Configuration (DSC)

PowerShell DSC isn't the same as PowerShell.[10] DSC is a feature introduced with PowerShell 4.0 that leverages the PowerShell scripting language to enable you to configure your infrastructure resources in a **declarative** way. DSC can help you have the correct configuration and avoid configuration drift.

PowerShell DSC scripts comprise three main blocks:

- **Configurations**: This is the outermost part of the script defined by the `Configuration` keyword.[11] It can contain one or more `Node` blocks and one or more `Resource` blocks.
- **Nodes**: These are the targets of the configuration.[12] You can define multiple computer names in a `Node` block.

[10] Microsoft. (2021, Dec. 15). *PowerShell Desired State Configuration (DSC) Overview*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/scripting/dsc/overview. [Accessed: Jul. 10, 2022].

[11] Microsoft. (2022, Jun. 06). *DSC Configurations*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/dsc/configurations/configurations. [Accessed: Jul. 10, 2022].

[12] Microsoft. (2021, Dec. 13). *Apply, Get, and Test Configurations on a Node*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/dsc/managing-nodes/apply-get-test. [Accessed: Jul. 10, 2022].

- **Resources**: This is where you define the properties of a `Resource` that sets the desired state of your configuration.[13]

The Local Configuration Manager (LCM) is the DSC engine that ensures the state you've defined in `Configurations` is achieved and maintained.[14]

PowerShell DSC scripts have the same *.ps1* extension as PowerShell scripts. The code snippet below is an example of a `Configuration` that declares a `Resource` of type *WindowsFeature*[15] and name *Web-Server*, should be present in the specified `Node`. It contains a second *Script* resource,[16] called *EditStartPage*, that modifies the IIS start page to display a custom message.

Example 5: `Deploy-WebServer.ps1` is a DSC script that installs IIS and adds some custom data to the default start page

```
Configuration Configure-IISServer
{
    param (
        [String[]]$ComputerName = 'localhost',
        [String]$ConnectionString
    )
    Node $ComputerName
    {
        ## Install IIS
        WindowsFeature AddIIS {
            Ensure = 'Present'
            Name   = 'Web-Server' # Internal name for IIS
        }
        ## Manipulate home page to display custom strings
        Script EditStartPage {
            GetScript  = { @{ Result = { "" } } }
            SetScript  = {
                $GetContentParams = @{
                    Path = "$env:SystemDrive\inetpub\wwwroot\iisstart.htm"
                }
                $File = Get-Content @GetContentParams
                $NewFile = $File | ForEach-Object {
                    $_
                    if ($_ -match 'IIS Windows Server') {
                        '<h1>IaC and CaC: Better Together!</h1>'
                        "<h2>Connection String is $using:ConnectionString</h2>"
                    }
                }
                $SetContentParams = @{
                    Value = $NewFile
                    Path  = "$env:SystemDrive\inetpub\wwwroot\iisstart.htm"
                }
                Set-Content @SetContentParams
            }
            TestScript = {
                $GetContentParams = @{
                    Path = "$env:SystemDrive\inetpub\wwwroot\iisstart.htm"
                }
                $Content = Get-Content @GetContentParams
```

[13] Microsoft. (2021, Dec. 13). *DSC Resources*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/dsc/resources/resources. [Accessed: Jul. 10, 2022].

[14] Microsoft. (2021, Dec. 15). *Configuring the Local Configuration Manager*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/dsc/managing-nodes/metaconfig. [Accessed: Jul. 10, 2022].

[15] Microsoft. (2021, Dec. 13). *DSC WindowsFeature Resource*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/dsc/reference/resources/windows/windowsfeatureresource. [Accessed: Jul. 10, 2022].

[16] Microsoft. (2021, Dec. 13). *DSC Script Resource*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/dsc/reference/resources/windows/scriptresource. [Accessed: Jul. 10, 2022].

```
                if ($Content -match 'IaC and CaC') { return $true }
                else { return $false }
            }
        }
    }
}
```

Use the `$using:` scope modifier[17] to access variables in the DSC script file from within a script resource block.

The code in the example behaves as a PowerShell function. You can test this configuration by adding the configuration name `Configure-IISServer` to the end of the script and running it, or by “dot-sourcing” it and calling it as a function from within PowerShell.

The following is an example of using the “dot-sourcing” method and calling the function with no arguments.

Example 6: Load and run DSC configurations by dot-sourcing and running them as functions

```
. .\Deploy-WebServer.ps1
Configure-IISServer
```

```
    Directory: C:\DSC\Configure-IISServer


Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        9/11/2021   5:48 PM           1984 localhost.mof
```

The following is how you can pass multiple computer names as arguments.

Example 7: You can run DSC configurations with parameters

```
. .\Deploy-WebServer.ps1
Configure-IISServer -ComputerName @('server01','server02')
```

[17] Microsoft. (2022, Mar. 18). *About Remote Variables*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_remote_variables. [Accessed: Jul. 12, 2022].

```
    Directory: C:\DSC\Configure-IISServer


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         9/11/2021   4:56 PM           1982 server01.mof
-a----         9/11/2021   4:56 PM           1982 server02.mof
```

Per the results in the example, two Managed Object Format (MOF) files are created inside a new folder that has the same name as the `Configuration` block. These MOF files contain what you defined in your `Configuration` block. The data in the MOF files is formatted so that it can be applied using Windows Management Instrumentation (WMI). PowerShell generates one MOF file per node.[18]

You can apply those configurations by running the following PowerShell cmdlet from the same folder where you executed the previous script.

Example 8: Applying the `Configure-IISServer` DSC configuration locally

```
Start-DscConfiguration -Path .\Configure-IISServer
```

```
Id Name PSJobTypeName   State   HasMoreData Location  Command
-- ---- -------------   -----   ----------- --------  -------
1  Job1 Configuratio... Running True        localhost Start-DscConfiguration...
```

To check the status of the DSC configuration, use `Get-DscConfigurationStatus`.[19]

Example 9: Checking the DSC configuration status

```
Get-DscConfigurationStatus
```

```
Status           Type            Mode   RebootRequested      NumberOfResources
------           ----            ----   ---------------      -----------------
Success          Initial         PUSH   False                2
```

From the results, you can see the configuration was applied successfully to two resources (*AddIIS* and *EditStartPage*) and no reboot is required. The `Type` is *Initial* since you're applying a new configuration. Consistency checks made by the LCM show up as the *Consistency* type.

You can pass the `-All` switch parameter to display all configuration status history. The output also contains a property named `Mode` that represents the LCM Refresh Mode, which can be *Disabled*, *Push*, or *Pull*.[20]

[18] Microsoft. (2021, Dec. 13). *Apply, Get, and Test Configurations on a Node*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/dsc/managing-nodes/apply-get-test. [Accessed: Jul. 10, 2022].

[19] Microsoft. (2022, Apr. 11). *Get-DscConfigurationStatus*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/PSDesiredStateConfiguration/Get-DscConfigurationStatus. [Accessed: Jul. 12, 2022].

[20] Microsoft. (2021, Dec. 13). *Enacting configurations*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/dsc/pull-server/enactingconfigurations. [Accessed: Jul. 10, 2022].

- **Disabled**: DSC configurations are disabled for that specific node
- **Push**: You have to manually initiate the configuration by calling `Start-DscConfiguration`. This is the mode used in the previous examples.
- **Pull**: The configuration is pulled from a pull service or Server Message Block (SMB) path on a specified interval.

To list the DSC configurations and associated resources on the current node, use `Get-DscConfigurationStatus`.[21]

Example 10: Checking all DSC configurations on the current node

```
Get-DscConfiguration | Select-Object ConfigurationName, ResourceId
```

```
ConfigurationName   ResourceId
-----------------   ----------
Configure_IISServer [WindowsFeature]InstallIIS
Configure_IISServer [Script]EditStartPage
```

The LCM has many other settings that you can change. Refer to Configuring the Local Configuration Manager[22] for more information on these.

For more information about DSC in general, refer to The DSC Book[23] by Don Jones and Missy Januszko.

10.6.2 CaC in Action

The previous section described how to use PowerShell DSC triggered locally on a computer. You'll now use PowerShell DSC to configure your remote Azure virtual machines.

For simplicity, DSC configurations are stored in an Azure Storage Account. However, for more advanced scenarios, you can use an Azure Automation Account.

Imagine you've already deployed the Azure SQL server, storage account, and load balancer covered in previous sections. Using the `Deploy-WebServer.ps1` DSC script as an example, the code snippet below installs IIS to a VM named *MyVM001*.

[21] Microsoft. (2022, Apr. 11). *Get-DscConfiguration*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/PSDesiredStateConfiguration/Get-DscConfiguration. [Accessed: Jul. 12, 2022].

[22] https://learn.microsoft.com/en-us/powershell/scripting/dsc/managing-nodes/metaconfig

[23] https://leanpub.com/the-dsc-book

Example 11: These two commands publish DSC configurations to Azure storage and apply them to VMs

```
$ResourceGroup = 'MyApp'
$VmName = 'MyVM001'
$StorageName = 'myuniquestorage593x'

## Publish the configuration script to user storage
$DSCConfigurationParams = @{
    ConfigurationPath  = '.\Deploy-WebServer.ps1'
    ResourceGroupName  = $ResourceGroup
    StorageAccountName = $StorageName
    Force              = $true
}

Publish-AzVMDscConfiguration @DSCConfigurationParams

## Set the VM to run the DSC configuration
$DSCExtensionParams = @{
    Version                   = '2.76'
    ResourceGroupName         = $ResourceGroup
    VMName                    = $VmName
    ArchiveStorageAccountName = $StorageName
    ArchiveBlobName           = 'Deploy-WebServer.ps1.zip'
    AutoUpdate                = $true
    ConfigurationName         = 'Configure-IISServer'
}

Set-AzVMDscExtension @DSCExtensionParams
```

The next logical step is to incorporate the code snippet into the *Azure-Virtual-Machine* module.

Add this to the bottom of the `New-AzureVirtualMachine` function of your *Azure-Virtual-Machine.psm1* module file:

Example 12: Add the Azure DSC configuration code to the `Azure-Virtual-Machine` module

```
for ($i = 1; $i -lt $VMInstances + 1; $i++) {

    $Instance = '{0:d3}' -f $i

    ## Publish the configuration script to user storage
    $DSCConfigurationParams = @{
        ConfigurationPath  = "$PSScriptRoot\Deploy-WebServer.ps1"
        ResourceGroupName  = $RGName
        StorageAccountName = $StorageAccountName
        Force              = $true
    }

    Publish-AzVMDscConfiguration @DSCConfigurationParams

    ## Set the VM to run the DSC configuration
    $DSCExtensionParams = @{
        Version                   = '2.76'
        ResourceGroupName         = $RGName
        VMName                    = "$VmBaseName$Instance"
        ArchiveStorageAccountName = $StorageAccountName
        ArchiveBlobName           = 'Deploy-WebServer.ps1.zip'
        AutoUpdate                = $true
        ConfigurationName         = 'Configure-IISServer'
        ConfigurationArgument     = @{
            ConnectionString = $ConnectionString
        }
    }

    Set-AzVMDscExtension @DSCExtensionParams

}
```

You'll also need to add two parameters to the `param()` block:

Example 13: Add the necessary additional parameters to the `New-AzureVirtualMachine` function

```
[Parameter(Mandatory = $false)]
[String]$StorageAccountName,

[Parameter(Mandatory = $false)]
[String]$ConnectionString
```

You can find the completed *Azure-Virtual-Machine.psm1* module file in the IaC Scripts[24] folder of the Extras[25] repository.

10.7 IaC and CaC: Better Together

So far, you've learned about Infrastructure as Code and Configuration as Code. You've learned the concepts and developed code for both IaC and CaC. Now it's time to see how they work together. Let's combine all scripts and code snippets covered in earlier sections into a blueprint script. The blueprint script below, `Two-Tier-App-Blueprint.ps1`, deploys all resources and triggers DSC, providing a full application deployment.

This script uses the default parameter values for the four module functions `New-Azure*`.

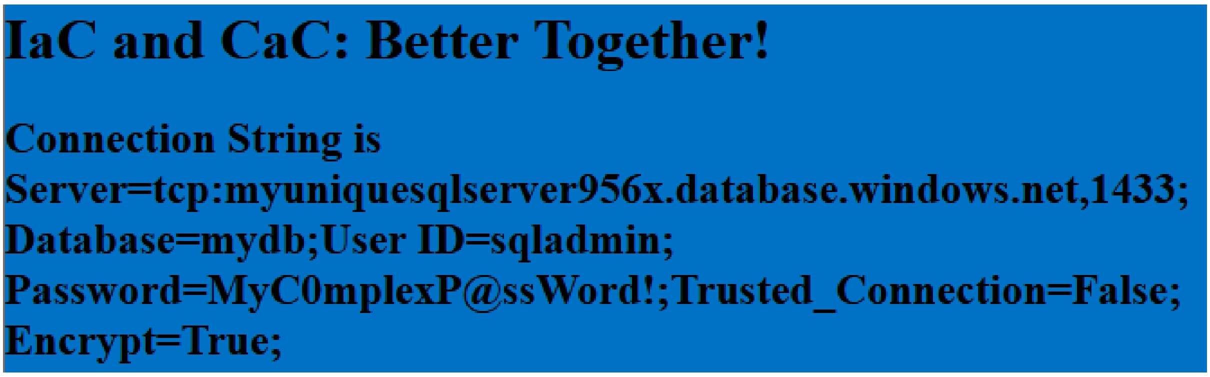
Example 14: The `Two-Tier-App-Blueprint.ps1` script ties together all the code to deploy the Azure resources and apply the DSC configuration

```
## Import all modules in current directory
foreach ($Module in Get-ChildItem $PSScriptRoot -Filter '*.psm1') {
    Import-Module -Name $Module.FullName
}

## Create SQL Server and store connection string
$ConnectionString = New-AzureSQLServer
$ConnectionString = -join $ConnectionString

## Create Storage Account
$StorageAccountName = New-AzureStorageAccount

## Create Load Balancer and store public ip address
$Pip = New-AzureLoadBalancer
```

[24] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/tree/main/Edition-01/IaC/Scripts/
[25] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/tree/main/Edition-01/

```

## Create Virtual Machine(s)
$VMParams = @{
    ConnectionString    = $ConnectionString
    StorageAccountName = $StorageAccountName
}
New-AzureVirtualMachine @VMParams

## Load webpage on default browser
Start-Process "http://$Pip"
```

The blueprint script deploys all resources. At the end, it opens your default browser and displays a customized IIS start page. This customized page is based on the string manipulation you did with PowerShell DSC in `Deploy-WebServer.ps1`.



IaC and CaC: Better Together!

Connection String is Server=tcp:myuniquesqlserver956x.database.windows.net,1433; Database=mydb;User ID=sqladmin; Password=MyC0mplexP@ssWord!;Trusted_Connection=False; Encrypt=True;

Custom strings on the IIS start page

10.8 Conclusion

Infrastructure as Code is at the core of DevOps practices, and it solves real problems. It helps you deploy consistently across different environments at scale while avoiding configuration drift.

In this chapter, you learned about the key concepts and principles of Infrastructure as Code. You also learned how to use IaC and CaC together to deploy and configure your infrastructure resources.

All the examples of IaC are based on PowerShell scripts. You're encouraged to create a free trial account in Microsoft Azure and experiment with the scripts provided in previous sections. Replicate what was explained in this chapter and get comfortable with the basics. Once you understand the basics, start making changes of your own and re-deploy. There's nothing like trial and error to get a deeper understanding of a subject.

10.9 Further Reading

- What is IaC?—Microsoft Docs[26]
- DSC Overview for Engineers—Microsoft Docs[27]
- PowerShell DSC Scripting Overview—Microsoft Docs[28]

[26] https://learn.microsoft.com/en-us/devops/deliver/what-is-infrastructure-as-code
[27] https://learn.microsoft.com/en-us/powershell/dsc/overview/dscforengineers
[28] https://learn.microsoft.com/en-us/powershell/scripting/dsc/overview

- PowerShell DSC 2.0 Overview—Microsoft Docs[29]
- Configuring the LCM—Microsoft Docs[30]
- Get Started with DSC for Windows—Microsoft Docs[31]
- Get Started with DSC for Linux—Microsoft Docs[32]
- Apply, Get, and Test Configurations on a Node—Microsoft Docs[33]
- Enacting DSC Configurations—Microsoft Docs[34]
- The DSC book—LeanPub[35]
- Microsoft Azure free trial[36]
- Azure Cloud Shell—Microsoft Docs[37]
- Azure DSC Extension Handler—Microsoft Docs[38]

[29] https://learn.microsoft.com/en-us/powershell/dsc/overview?view=dsc-2.0

[30] https://learn.microsoft.com/en-us/powershell/scripting/dsc/managing-nodes/metaconfig

[31] https://learn.microsoft.com/en-us/powershell/dsc/getting-started/wingettingstarted

[32] https://learn.microsoft.com/en-us/powershell/dsc/getting-started/lnxgettingstarted

[33] https://learn.microsoft.com/en-us/powershell/dsc/managing-nodes/apply-get-test

[34] https://learn.microsoft.com/en-us/powershell/dsc/pull-server/enactingconfigurations

[35] https://leanpub.com/the-dsc-book

[36] https://azure.microsoft.com/en-us/free/

[37] https://learn.microsoft.com/en-us/azure/cloud-shell/overview

[38] https://learn.microsoft.com/en-us/azure/virtual-machines/extensions/dsc-overview

IV Using Regexes

> *“Garbage in, Garbage out.”* — George Fuechsel, Raymond Crowley, et al.

Administrators must often work with external data, so pattern matching is a valuable tool. Regex patterns play an important role in transforming and bringing data into the PowerShell environment. This role becomes even more significant now that PowerShell is cross-platform. This section takes you right from the beginning, through to advanced uses for regexes, including:

- A 101-style introduction to regexes.
- Accessing and using regex patterns in PowerShell.
- A deep dive into PowerShell and .NET regex.
- Best practices for using regexes in PowerShell.

11. Regex 101

Regular expressions, or **regexes**[1], are a powerful pattern-matching technology and language for parsing plain text into usable objects. In PowerShell, much of the data you work with is already in this form. A regex is therefore most useful when processing the output from external applications or environments, such as log files and templates. You may have happened upon regexes in other learning materials, or other programming languages and environments. They might have appeared to be a daunting subject or otherwise a niche area of expertise. Developers, engineers, and administrators use these patterns often, if not daily. They're an essential tool in your toolkit and can become intuitive with a basic understanding of how regex engines process text strings.

Did you know?

Mathematician Stephen Cole Kleene developed *regular language*, the concept behind regular expressions, in the early 1950s.[2] It was almost twenty more years before the first computer implementations existed. An early adopter of the syntax was Ken Thompson's Quick Editor (QED) in 1968.[3]

This chapter recaps the fundamental syntax and structure of regex patterns and explains how to use these in the PowerShell environment. Accessing Regexes introduces more complex constructs, solving relevant examples to explain these and demonstrate the power of regexes. Regex Deep Dive discusses deconstructing and debugging your patterns, looking from the perspective of the regex engine, and getting familiar with the machinery within. It also covers the remaining syntax to complete your PowerShell regex toolkit. Finally, Regex Best Practices rounds off with some best practices, design strategies, and where to go for more on regexes in PowerShell and in general.

11.1 First Principles and Limitations

> There is some debate about whether modern regexes can still be considered regular expressions. The regex chapters use the terms **regex** and **regexes** to avoid confusion.

Regexes are, in essence, instructions that tell a regex engine how to read some text for you. They define patterns to *match* in a text string and can *capture* substrings. You can use these *captures*

[1] Though **regexes** is the correct plural, the uncountable plural form **regex** is also common when referring to the topic as a whole, as is **regex** to describe the underlying theory.

[2] S. C. Kleene. (1956). *Representation of Events in Nerve Nets and Finite Automata.* In: Automata Studies, (AM-34), pp. 3–42. C. E. Shannon and J. McCarthy (eds.). Princeton University Press. DOI: 10.1515/9781400882618-002.

[3] K. Thompson. (1968). *Programming Techniques: Regular expression search algorithm.* Commun. ACM, vol. 11, no. 6, pp. 419–422. DOI: 10.1145/363347.363387.

as *backreferences* later in the pattern, and they're also returned by the engine back into the programming environment. Captures are also used to substitute a replacement into the input string.

Regexes have limitations, of course. The idea of *Garbage In, Garbage Out* (GIGO) is relevant here. The regex engine interprets the provided pattern sequentially, reading through the input text and backtracking as necessary until no more matches are possible. It can't observe the input text as a whole or make decisions from a *big picture* perspective, as humans can. It's therefore prudent to make use of the programming environment to supplement your regex patterns. You can find out more about this later.

PowerShell regex uses the *.NET* regex engine. Examples in this book can be applied to other *.NET* languages (**C#**, **VB.NET**, **F#**, and **ASP.NET**).

11.1.1 Wildcard Patterns vs. Regexes

PowerShell also supports **wildcard expressions**. These are much simpler pattern matching expressions that don't support capturing. You can use these with the `-like` and `-notlike` operators, and with cmdlet parameters that support wildcards. The Advanced Conditions chapter covers wildcard patterns comprehensively.

11.1.2 Differences Between PowerShell Regexes and Others

PowerShell's regex flavor is largely based on Perl 5[4]. The flavor hasn't changed since 2.0 and is therefore relevant to all supported editions of PowerShell at the time of writing.

There are several differences between PowerShell regexes and other common flavors, such as Perl-Compatible Regular Expressions (PCRE), Python, and Java.[5] The more important ones are:

Not Supported:

- Pattern recursion (alternative: *balancing groups*)
- Possessive quantifiers (alternative: *atomic groups*)
- Unicode scripts (alternative: *Unicode blocks*)
- Unicode whole graphemes (alternative: individual code points)
- Global `/g` modifier (alternative: use `[regex]::Matches()` instead of `::Match()` or `-match`)
- Subroutines
- Branch reset groups
- Literal text spans
- Character class intersection

Supported:

[4] Microsoft. (2020, Jun. 30). *.NET regular expressions*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/standard/base-types/regular-expressions. [Accessed: Jun. 12, 2021].

[5] J. E. F. Friedl. (2006). *Mastering Regular Expressions*. 3rd ed. Beijing [u.a.]: O'Reilly. ISBN: 978-0-596-52812-6.

- Named groups `(?<name>)` or `(?'name')`
- Backreferences `\NN` and `\k<name>` or `\k'name'`
- Conditional subexpressions
- Atomic groups
- Unicode
  - Code points `\uHHHH`
  - Blocks `\p{Is<BlockName>}`
  - Categories Short form/key: `\p{X}` or `\p{Xy}`
- Balancing groups `(?<First>...)...(?<Second-First>...)`
- Inline comments `(?# Comment)`
- Inline options `(?msnix-msnix)`
- Option spans `(?msnix-msnix:...)`
- Variable-length lookarounds, including lookbehinds
  - Positive lookahead `(?=...)`
  - Negative lookahead `(?!...)`
  - Positive lookbehind `(?<=...)`
  - Negative lookbehind `(?<!...)`

This list is primarily for reference, and you can learn more about the topics described later.

11.2 Getting Started

This chapter covers the basics of regexes. If you already have an intermediate understanding of regexes, you can consider skipping ahead to the Accessing Regexes chapter.

However, there are lots of useful pointers, so reading through will help set you up for later.

To introduce the fundamentals of regexes, consider this example. The days of the week in the English language all end in ‘day’, with three to six preceding letters. You could simply match each day individually.

Example 1: Matching 'Friday'

```
$MyString = 'Fridays are the best days.'

$MyString -match 'Monday'
$MyString -match 'Tuesday'
$MyString -match 'Wednesday'
$MyString -match 'Thursday'
$MyString -match 'Friday'
$MyString -match 'Saturday'
$MyString -match 'Sunday'
```

```
False
False
False
False
True
False
False
```

This works because the regex engine interprets the letters of *Friday* and *Saturday* literally. To avoid using the `-match` operator many times, you can use **alternation**. Think of this as a *logical OR*. Regexes use the pipe **|** character for alternation.

Example 2: Matching any day of the week

```
$MyString = 'It rained on Friday, but Monday will be clear.'

$MyPattern = 'Monday|Tuesday|Wednesday|Thursday|Friday|Saturday|Sunday'

$MyString -match $MyPattern
```

```
True
```

11.3 Character Classes

What about matching any word ending in 'day'? **Character classes** match one character from a **list**. Perhaps the most common of these is **`\w`**, the *word character* class. This matches letters, numbers, and marks that aren't punctuation (except the underscore _) and aren't white space. PowerShell regexes support Unicode, so this includes word characters for many other languages, too. In fact, of the first 65,536 Unicode code points, `\w` matches 50,320 of them. For everyday use with English language characters, this matches `A-Z`, `a-z`, `0-9`, and `_`.

Example 3: Matching 4 letters before 'day'

```
$MyRegex = '\w\w\w\wday'

'Tomorrow is a holiday.' -match $MyRegex
'Tomorrow is Monday.' -match $MyRegex
'Tomorrow is Wednesday.' -match $MyRegex
```

```
True
False
True
```

Notice the backslash \ used with the `w` here. Backslashes are **metacharacters** in regexes—they have a special meaning. They denote special tokens, including character classes. They can also denote escape sequences.

1. *holiday* results in a match, since `\w\w\w\w` matches '**holi**'.
2. *Monday* results in no match, because `\w` doesn't match spaces, and there are only three letters before '**day**'.
3. *Wednesday* results in a match, even though there are more than four letters before '**day**'. This is because the pattern can still match `\w\w\w\w` with '**dnes**'. **Anchors** overcome this problem. You can read more about them later in the chapter.

There are three other predefined character classes in PowerShell regexes:

- To match **numeric digits**, use **`\d`** (for everyday English, think `0-9`).
- To match **space characters**, use **`\s`** (for everyday English, think spaces, tabs, and newlines).
- To match **any character** except a newline, use a period (**`.`**).

There are also three **inverse** character classes matching any character that **isn't** in that class. Each uses the same letter but capitalized: **`\W \D \S`**.

Example 4: Using character classes and their inverses

```
'ab' -match '\S\s'
'ab' -match '\S\S'
'ab c1' -match '\w\w\s\D\d'
```

```
False
True
True
```

The first pattern is looking for one not-space and one space, but the string contains no spaces. The second pattern is looking for two not-spaces, which match '**ab**'. The third pattern is looking for two word characters, a space, one not-decimal, and one decimal. This matches '**ab c1**', as '**c**' isn't a decimal.

There are also escape sequences that match single characters, such as a newline. You can find out about these later in the chapter.

11.4 Custom Character Classes

If you need to define your own set of characters to match, use square brackets `[...]`. This can be useful for narrowing the scope when searching for specific text, such as punctuation or disallowed characters in user input.

Example 5: Using a custom class to match punctuation

```
$MyPattern = '\w[,."'':;?!]'

'This sentence has no punctuation' -match $MyPattern
'This is punctuated with a period.' -match $MyPattern
'The period in "1.2" also matches' -match $MyPattern
```

```
False
True
True
```

Note the two single quotation marks `''`, which insert a single mark `'` into a literal string. Observe also that a period `.` inside a custom character class matches only a **period**, not *any character*.

Custom character classes are also capable of inversion. By inserting a caret `^` after the opening square bracket `[`, the class matches **any character** not in it.

The following example uses the `$Matches` automatic variable. The next chapter, Accessing Regexes, discusses this further. For now, know that PowerShell sets `$Matches` each time you call `-match` and it succeeds. The entry `$Matches[0]` reveals what the **entire pattern** matched in the input string.

Example 6: Inverse character classes

```
$MyPattern = '"[^"]+"'

'This is a string with a "quoted" word' -match $MyPattern
$Matches[0]
```

```
True
"quoted"
```

Example 6 matches one or more characters that aren't quotation marks, surrounded by quotation marks. The plus `+` is a **quantifier** meaning, *match **one or more** instances of the last token.*

Character Classes Reference

You can view a complete reference for character classes at **Microsoft Docs**[6].

[6] https://learn.microsoft.com/en-us/dotnet/standard/base-types/character-classes-in-regular-expressions

11.5 Quantifiers

Repeating literal characters or character classes for large match spans would take up a lot of space. Regexes make use of **quantifiers** to control the number of allowed repetitions of the preceding token in the pattern. You can rewrite the pattern from earlier, `\w\w\w\wday`, as `\w{4}day`. What about a range of repeats?

Example 7: Using the range quantifier

```
$MyPattern = '\w{3,6}day'

'Today' -match $MyPattern # 2 letters before 'day'
'Tuesday' -match $MyPattern # 4 letters before 'day'
```

```
False
True
```

This results in the general formula `{min,max}`. Don't use a space character on either side of the comma. You can omit `max`, leaving the comma (`,`) in place, to match `min` *or more*.

Example 8: Using the 'at least' quantifier

```
# 6 or more word characters before 'day'
$MyPattern = '\w{6,}day'

'Tuesday' -match $MyPattern # 4 letters before 'day'
'Wednesday' -match $MyPattern # 6 letters before 'day'
'Postholiday' -match $MyPattern # 8 letters before 'day'
```

```
False
True
True
```

There are also shorthand quantifiers for common requirements.

- Plus `+`, which matches **one or more** repetitions (think `{1,}`)
- Star `*`, which matches **zero or more** repetitions (think `{0,}`)
- Question mark `?`, which matches **zero or one** repetition (think `{0,1}`)

You can use the **zero or one** quantifier to make tokens optional.

Example 9: Using the 'zero or one' quantifier

```
$MyPattern = 'h?ello'

'hello' -match $MyPattern
'yellow' -match $MyPattern
```

```
True
True
```

Example 10: The 'zero or more' quantifier, and greedy matching

```
$MyPattern = 'ab.*ly'

'absolutely' -match $MyPattern
$Matches[0]

'ably' -match $MyPattern
$Matches[0]

'absolutely lovely' -match $MyPattern
$Matches[0]
```

```
True
absolutely

True
ably

True
absolutely lovely
```

The first match understandably succeeds, with `.*` matching '**solute**'. The second proves that `*` can match zero times. What's going on with number three, then? Wouldn't it just match '**absolutely**'?

Going back to the idea of **sequential** processing, the engine performs the following steps:

1. The regex engine matches `a` to '**a**', then `b` to '**b**', so it moves on.
2. `.*` matches zero **or more** characters, and the engine doesn't yet care what comes next in the pattern. Therefore, it matches '**solutely lovely**'.
3. The engine tries to match `l`, but there are no more characters, so the `.*` gives up the last '**y**'.
4. The engine still can't match `l` to '**y**', so the `.*` gives up the last '**l**'.
5. The engine now matches `l` to '**l**' and moves on.
6. Finally, the engine matches `y` to '**y**', and has now matched the pattern fully, so processing stops.

Matching **as much as possible**, then giving back as needed, is **greedy** matching. This is the default in PowerShell regexes. The opposite of greedy is **lazy** matching, where the quantifier matches **as little as possible** and takes more if needed. You can make any quantifier lazy by adding a question mark `?` after it:

- **`+?`**: Matches one or more lazily
- **`*?`**: Matches zero or more lazily
- **`??`**: Matches zero or one lazily
- **`{min,max}?`**: Matches between *min* and *max* times lazily
- **`{min,}?`**: Matches *min* or more lazily

With this in mind, the lazy `.*?` fixes the last match in Example 10.

Example 11: Using the 'lazy quantifier' modifier

```
$MyPattern = 'ab.*?ly'

'absolutely lovely' -match $MyPattern
$Matches[0]
```

```
True
absolutely
```

The engine only matches '**absolutely**' this time. Initially, `.*?` only matches the first '**s**' but takes more characters until the next token `l` matches the first '**l**'. This matches, but the following '**u**' doesn't. Therefore, the engine **backtracks** in the pattern, with `.*?` taking more characters until the next token `l` matches the second '**l**'. The final token `y` then matches the last '**y**' and processing stops with a successful match.

The Regex Deep Dive chapter discusses backtracking and branching in more detail.

Quantifiers Reference

You can view a complete reference for quantifiers at **Microsoft Docs**[7].

11.6 Character Escape Sequences

Character escape sequences match a single character instead of a character from a range, like character classes. You can use them to match characters you can't type, or would otherwise have a special meaning (think `?`, `*`, or `\`, for example).

The following example uses a literal here-string. Use these to create multiline strings in PowerShell. Expandable here-strings `@"` and `"@` also exist.

[7] https://learn.microsoft.com/en-us/dotnet/standard/base-types/quantifiers-in-regular-expressions

Example 12: Matching line endings with character escape sequences

```
$MyPattern = '\r?\n'

@'
This is a uniline (single line) string
'@ -match $MyPattern

@'
This is
a multiline string
'@ -match $MyPattern
```

```
False

True
```

Example 12 matches line endings for both Windows and Unix-like systems. This is useful when working in PowerShell 6.0 and later, which is cross-platform. The first part of the pattern, `\r?`, matches an optional single *carriage return* (0x0D). The second part, `\n`, matches a *line feed* (0x0A).

You can escape character classes too. Escaping the backslash `\` causes the engine to interpret it literally. The sequence `\\n` matches a backslash followed by an 'n', not a line feed.

Character Escape Sequences Reference

You can view a complete reference for escape sequences at **Microsoft Docs**[8].

11.7 Anchors (*Zero-Width Assertions*)

The pattern `\w{3,6}day` from Example 7 matches all days of the week, but also longer words. The engine matches the word '**postholiday**' after backtracking twice, skipping the '**po**'. Is this preventable? The answer is yes, with **anchors**. These are assertions that set conditions for matches but don't take any characters from the string. Therefore, they're also known as *zero-width* assertions.

Perhaps the most common of these is the **caret** `^` and **dollar** `$`. A caret matches the **beginning** of the string, so `^aa\w+` matches 'aardvark', but not 'the aardvark'. A dollar matches the **end** of the string, so `ero$` matches 'zero', but not 'zeroes'. *Multiline mode*, a regex option, causes these anchors to match the beginning and end of lines, too. You can read more about regex options in the Regex Deep Dive chapter.

Another commonly used anchor is the **word boundary** `\b`. This matches the boundary between a word `\w` character and a not-word `\W` character. It also has an inverse, the not-word-boundary `\B`. This can never match a word boundary.

[8] https://learn.microsoft.com/en-us/dotnet/standard/base-types/character-escapes-in-regular-expressions

Another kind of zero-width assertion is the **lookaround**. This feature matches one or more regex tokens in a subexpression while still being zero-width. The Regex Deep Dive chapter discusses subexpressions further.

Anchors Reference

You can view a complete reference for regex anchors at **Microsoft Docs**[9]

11.8 Captures

The final topic that this chapter covers is the **capturing subexpression**, or *capturing group*. These enable you to extract substrings from the input string and use brackets (parentheses) `(` and `)`. You can wrap any part of a regex pattern to make it a capturing subexpression. The engine assigns it a number starting from `1`, and if it makes a match, it returns the *captured* substring into the programming environment. The next example extracts the original word from an *infixed* one.

The terms capturing *group* and capturing *subexpression* are often used interchangeably. Technically, a subexpression is any part of a regex pattern delimited by grouping constructs. Capturing subexpressions create capturing groups, which capture matches made by the subexpressions within. To avoid confusion, this chapter uses the term *group* for both the construct and the resulting group.

Example 13: Extracting infixed text with a capturing group

```
$MyPattern = '\b(\w+)-\w{6,}-(\w+)\b'

'This is fan-flaming-tastic!' -match $MyPattern
$Matches[0]

$Matches[1], $Matches[2] -join ''
```

```
True
fan-flaming-tastic

fantastic
```

This pattern may seem a bit complicated but, when broken down, it's straightforward.

- The word boundary `\b` anchors at each end force the engine to match no more or less than the infixed word.
- The first `(\w+)` matches **and captures** the part of the word before the first hyphen.
- The `-\w{6,}-` matches an infixation with at least six letters. This eliminates most other multi-word phrases like '*up-to-date*', but not phrases with longer middle words, like '*well-thought-out*'.

[9] https://learn.microsoft.com/en-us/dotnet/standard/base-types/anchors-in-regular-expressions

- The last `(\w+)` matches **and captures** the part of the word after the last hyphen.

Try the examples in this chapter in PowerShell and observe how changing the input or pattern changes the output.

Experimentation is a great way to familiarize yourself with regexes.

The `$Matches` automatic variable stores these numbered captures as entries in a hashtable. Recall that the zeroth entry `$Matches[0]` is the *capture* of the entire pattern. It's also possible to group regex tokens without capturing them. This is a **non-capturing group** and takes the form `(?:...)` where `...` is any regex subexpression.

The `$Matches` automatic variable isn't nullified before each regex operation. This means that if a match fails, `$Matches` will still contain the result of the last successful match.[10] You should therefore check for match success before reading this variable.

11.9 Visualizing Captures

Sometimes, it can be a little hard to visualize what matches, groups, and captures actually are. The following example provides a helpful representation using .NET methods. You can find out more about the .NET methods in the Accessing Regexes chapter.

Example 14: Visualizing captures

```
$Result = [regex]::Matches(
    'abcdefg hijklmn', '(?:(?<first>\w)(?<second>\w))+'
)

if ($Result.Success) {
    $matchCount = -1
    $Result.ForEach{
        Write-Host ('Match {0}:' -f ++$matchCount)
        $groupCount = -1
        $_.Groups.ForEach{
            Write-Host ('  Group {0} (Name = {1}):' -f ++$groupCount, $_.Name)
            $captureCount = -1
            $_.Captures.ForEach{
                Write-Host (
                    '    Capture {0} (pos {1}) = {2}' -f
                    ++$captureCount, $_.Index, $_.Value
                )
            }
        }
    }
} else { Write-Host 'No matches' }
```

[10] Microsoft. (2021, Oct. 27). *About Automatic Variables (Microsoft.PowerShell.Core) - Matches*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_automatic_variables#matches. [Accessed: Nov. 15, 2021].

```
Match 0:
  Group 0 (Name = 0):          <-- Entire match 'abcdef'
    Capture 0 (pos 0) = abcdef <-- $0 or $& in replacement pattern
  Group 1 (Name = first):      <-- From subexpression (?<first>\w)
    Capture 0 (pos 0) = a
    Capture 1 (pos 2) = c
    Capture 2 (pos 4) = e      <-- $1 or ${first} in replacement pattern
  Group 2 (Name = second):     <-- From subexpression (?<second>\w)
    Capture 0 (pos 1) = b
    Capture 1 (pos 3) = d
    Capture 2 (pos 5) = f      <-- $2 or ${second} in replacement pattern
Match 1:
  Group 0 (Name = 0):          <-- Entire match 'hijklm'
    Capture 0 (pos 8) = hijklm <-- $0 or $& in replacement pattern
  Group 1 (Name = first):      <-- (From subexpression (?<first>\w)
    Capture 0 (pos 8) = h
    Capture 1 (pos 10) = j
    Capture 2 (pos 12) = l     <-- $1 or ${first} in replacement pattern
  Group 2 (Name = second):     <-- From subexpression (?<second>\w)
    Capture 0 (pos 9) = i
    Capture 1 (pos 11) = k
    Capture 2 (pos 13) = m     <-- $2 or ${second} in replacement pattern
```

Grouping Constructs Reference

You can view a complete reference for grouping constructs at **Microsoft Docs**[11].

[11] https://learn.microsoft.com/en-us/dotnet/standard/base-types/grouping-constructs-in-regular-expressions

12. Accessing Regexes

The Regex 101 chapter introduced the `-match` operator. There are, however, lots of ways to use regexes in PowerShell. This chapter introduces you to them.

12.1 Using PowerShell Syntax

The native PowerShell operators, statements, and cmdlets are the easiest way to use regex patterns in PowerShell. They're likely the ones you'll use when writing quick PowerShell scripts or performing tasks. This section covers many of them comprehensively.

12.1.1 -match Operator with Strings

This is arguably the most-used operator when working with regexes in PowerShell. It has two behavior modes—one for strings, and one for arrays. With strings, it returns `$true` or `$false`, depending on whether the pattern matches the input string. If the engine finds a match, it sets the `$Matches` automatic variable, a hashtable. The first entry, `$Matches[0]`, contains a captured match of the whole regex pattern. If it makes any captures during the match, it assigns a numeric entry for each, starting from `1`. The engine assigns string entries for named captures. If you pass an empty string as the pattern, `-match` always returns `$true` and sets `$Matches[0]` to an empty string.

Example 1: Use of the `-match` operator with strings

```
$MyString = 'This is the input string'
$MyPattern = '(.*is).*?(?<NamedCapture>\w+ring)'

$Result = $MyString -match $MyPattern

if ($Result -eq $true) {
    $Matches[0]
    $Matches[1]
    $Matches['NamedCapture']
}
```

```
This is the input string
This is
string
```

12.1.2 -match Operator with String Arrays

The versatile `-match` operator has a trick up its sleeve. What would you expect if you were to use it on a string array? In this instance, the operator becomes more of an *is-match* ***filter***, returning an array. Only elements that match the pattern exist in this new array. The `$Matches` automatic variable isn't set at all.

Example 2: Use of the `-match` operator with arrays

```
$MyArray = @('red', 'yellow', 'green', 'blue', 'purple')
$MyPattern = '(\w)\1'

$MyArray -match $MyPattern
```

```
yellow
green
```

Example 2 includes a **backreference** `\1` to the first capture. Backreferences match a capture elsewhere in a pattern.

In Example 2, only colors that have two repeated letters match, so the output array contains only '*yellow*' and '*green*'.

12.1.3 -cmatch and -imatch Operators and Inverses

By default, the `-match` operator matches in a **case-insensitive** manner. Case-sensitive matching is possible using `-cmatch` instead. You can use the inline options to force case-insensitive matching, even with `-cmatch`. You can learn more about inline options in the Regex Deep Dive chapter. `-imatch` is simply an alias of `-match`.[1] [2] You can use this to explicitly declare that case-insensitive matching is intentional.

Each matching operator also has an inverse. `-notmatch`, `-cnotmatch`, and `-inotmatch` behave identically to their conventional counterparts, but produce negated boolean outputs.

12.1.4 -replace Operator with Strings

The regex part of this book has only discussed *matching* so far, but **replacement** is another important aspect of text manipulation. The `-replace` operator has three operands, instead of two. The third is a **replacement** pattern, also called a substitution string. The only *metacharacter* in replacement patterns is the dollar `$`. To replace with a literal dollar, use two dollar symbols `$$`. If you pass an empty string—or nothing—as the replacement, matches get removed.

[1] Microsoft. (2017, Jan. 16). *System.Management.Automation - tokenizer.cs*. L657-L692. PowerShell/PowerShell on GitHub. [Online]. Available: https://github.com/PowerShell/PowerShell/blob/master/src/System.Management.Automation/engine/parser/tokenizer.cs. [Accessed: Jul. 10, 2021].

[2] Microsoft. (2016, Mar. 31). *System.Management.Automation - InternalCommands.cs*. L1699. PowerShell/PowerShell on GitHub. [Online]. Available: https://github.com/PowerShell/PowerShell/blob/master/src/System.Management.Automation/engine/InternalCommands.cs. [Accessed: Jul. 04, 2021].

Example 3: Replacing strings with the `-replace` operator

```
$MyString = 'May: The sunshine is mellow.'

$MyString -replace '\bm(\w+)\b', 'y$1'
$MyString -replace '([aeiou])'
```

```
yay: The sunshine is yellow.
My: Th snshn s mllw.
```

12.1.4.1 Using a Script Block

The `-replace` operator also accepts a script block instead of a replacement pattern. This provides for more advanced string manipulation.

Example 4: Replacing infixed capital letters with `-replace` and a script block

```
$MyString = 'This sentEnce has infixed cApital leTTErs.'

# Any matches are converted to lowercase
$Evaluator = {
    Write-Host "Run on '$_'"
    ([string]$_.Value).ToLower()
}

$MyString -creplace '(?!\A)\b[a-z]*[A-Z][A-Za-z]*\b', $Evaluator
```

```
Run on 'sentEnce'
Run on 'cApital'
Run on 'leTTErs'
This sentence has infixed capital letters.
```

- `(?!\A)`: Must not be at the start of the string
- `\b[a-z]*`: A word boundary followed by zero or more lowercase letters
- `[A-Z]`: An uppercase letter
- `[A-Za-z]*\b`: Zero or more uppercase or lowercase letters followed by a word boundary

Note that the engine populates the `$_` automatic variable with the `[Match]` object for the *whole* match, regardless of any capturing groups. You can learn more about `[Match]` objects in the Using the .NET Methods section, later in this chapter.

The subexpression `(?!\A)` is a negative lookahead, a kind of lookaround. You can find out more about lookarounds in the Regex Deep Dive chapter.

12.1.5 -replace Operator with String Arrays

Much like with the `-match` operator, the `-replace` operator is also usable on string arrays. As you might expect, the regex engine applies the pattern to each string in the array individually. This returns a new array with the replaced strings from the old one. Unlike with the `-match` operator, **all** elements are present in the new array, even if the engine made no replacements to some of them.

Example 5: Replacing an array of strings with the `-replace` operator

```
$MyArray = @(
    'Monday'
    'Tuesday'
    'Wednesday'
    'yesterday'
)

$MyArray -creplace '[A-Z]\w+(day)', 'to$1'
```

```
today
today
today
yesterday
```

12.1.6 -creplace and -ireplace Operators

A case-sensitive operator also exists, `-creplace`. Once again, this works identically to `-replace` with case-sensitive matching. `-ireplace` is also an alias for `-replace`, provided to explicitly declare case-insensitivity.

Comparison Operators Reference

You can view a complete reference for regex-related comparison operators at **Microsoft Docs**[3]. The relevant headings are *Matching Operators* and *Replacement Operator*.

12.1.7 -split Operator with Strings

Joining and splitting are important tools in string manipulation. The `-split` operator in PowerShell is powerful, supporting regex patterns and substring count limits. In its default mode, the operator splits the input string at all white space characters. To use the default mode, place the operator before the string.

Example 6: Splitting a string at all white space

```
-split 'Red     Green Blue'
```

```
Red
Green
Blue
```

To split using a regex pattern for the delimiter, place the operator after the string and follow it with the pattern. The engine removes all parts of the string that match the delimiter pattern and returns the intervening chunks as an array of strings. Use quantifiers if the delimiter may appear more than once, as the zero-width gaps between them result in empty strings in the output array.

[3] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_comparison_operators

Example 7: Splitting strings with repeated delimiters

```
'Red, Green,, Blue' -split '[, ]+'
#   ^-     ^--
# 2 delimiter matches producing 3 chunks.
# Note the '+' quantifier

'Cyan, Magenta,, Yellow' -split '[, ]'
#    ^^       ^^^
# 5 delimiter matches producing 6 chunks,
# 3 of which are empty strings
```

```
Red
Green
Blue
Cyan

Magenta


Yellow
```

12.1.7.1 Capturing the Delimiter with -split

Another interesting feature of using regexes with `-split` is the ability to capture all or part of the delimiter, and include it in the substring output. Use capturing groups as you would in any other regex pattern. The engine returns any captures as substrings, and there is no limit to the number of captures. `-split` supports numbered and named captures, too.

Example 8: Splitting strings with captured delimiters

```
'Red,Green:Blue.Cyan' -split '([.,])|\p{P}'
```

```
Red
,
Green
Blue
.
Cyan
```

Example 8 uses alternation to capture the delimiter only if it's a period '.' or comma ','. The character class `\p{P}` matches all punctuation characters and is a *Unicode category*. You can find out more about Unicode categories in the Regex Deep Dive chapter.

The example also reveals a little about how the regex engine processes matches. The `\p{P}` category also includes periods and commas, but the first alternative `[.,]` matches first. The engine, therefore, doesn't need to try these delimiters against the second alternative and, since the first alternative includes a capture, the engine returns them. With this in mind, it's now clear why the engine doesn't return the colon ':' between '**Green**' and '**Blue**'. It doesn't make a match, so must try the second alternative. This *does* match but doesn't include a capture.

12.1.7.2 Substring Count Limits with -split Using Max-Substrings

You can also limit the number of substrings returned by the `-split` operator. To achieve this, follow the delimiter pattern with a comma and an integer. This isn't an absolute limit on the substrings returned, however. If passing an array of strings, the limit applies to each string individually. If capturing all or part of the delimiter, these captures don't count towards the limit.

Example 9: Splitting strings with substring limits and captures

```
# Without capture
'Red,Green.Blue/Cyan:Magenta' -split '\p{P}', 3

# With capture
'Red,Green.Blue/Cyan:Magenta' -split '(\p{P})', 3

# Array
@('Red,Green.Blue', 'Cyan:Magenta') -split '\p{P}', 3
```

```
Red
Green
Blue/Cyan:Magenta

Red
,
Green
.
Blue/Cyan:Magenta

Red
Green
Blue
Cyan
Magenta
```

Example 9 demonstrates how passing an array of strings or capturing the delimiter affects the substring limit. The max-substrings parameter only guarantees the number of substrings **per string** that **aren't delimiters**.

A new feature, available in PowerShell 7 and later, means `-split` accepts a negative integer for the max-substrings parameter. Negative values invert the substring limit, and the engine applies it in a last-to-first order.

Example 10: PowerShell 7 inverse max-substrings

```
'Red,Green,Blue,Cyan' -split ',', -3
```

```
Red,Green
Blue
Cyan
```

12.1.7.3 Regex Options with -split

Besides supporting inline regex options `(?msnix)`, the `-split` operator is special because it supports some regex options. You must pass these flags by their names as a comma-separated list within a string.

Example 11: Using regex options with -split

```
# Unnamed capture ignored
'Red,Green,Blue' -split '(,)', 100, 'ExplicitCapture'

# Capture with name 'name' captures the delimiter
'Red,Green,Blue' -split '(?<name>,)', 100,
    'ExplicitCapture'
```

```
Red
Green
Blue

Red
,
Green
,
Blue
```

In Example 11, the engine ignores the unnamed capturing group because the 'ExplicitCapture' regex option causes only **named** groups to capture their contents. The first `-split` command, therefore, returns no commas between substrings. An important point to note is that `-split` is case-insensitive, and therefore the 'IgnoreCase' makes no difference. If using the `-csplit` operator, however, 'IgnoreCase' or indeed the inline option `(?i)`, would change the output. To view the options by their names, use the `[Enum]::GetNames()` method:

```
[Enum]::GetNames([System.Management.Automation.SplitOptions])
```

12.1.7.4 Simple Matching with -split

The `-split` operator supports plain text matching, too, using the 'SimpleMatch' parameter. This goes in the same place as the regex options would. The only other option available with 'SimpleMatch' mode is 'IgnoreCase', which enables case-insensitive matching with `-csplit`, and is inconsequential with `-split`.

12.1.8 -split Operator with String Arrays

As mentioned earlier, `-split` accepts one or more strings. When passed an array, the `-split` operator treats each string individually. You can use the operator in both unary and binary modes with arrays, just like with strings.

Example 12: Splitting string arrays by white space and by punctuation

```
$MyArray = @('Red Green', 'Blue/Cyan')

# Unary mode, splitting by white space
-split $MyArray

# Binary mode, splitting by white space or slashes '/'
$MyArray -split '[\s/]'
```

```
Red
Green
Blue/Cyan

Red
Green
Blue
Cyan
```

12.1.9 Splitting Strings with -split and a Script Block

In place of a regex pattern, you can pass `-split` a script block. Within this block, the `$PSItem` or `$_` automatic variable becomes each character in each of the input strings. Returning `$true` inside this block declares the current character to be a delimiter.

It's important to note that this form of the `-split` operator doesn't use the regex engine and can't capture delimiters. It also passes the string to the evaluator one character at a time, so doesn't support splitting on multicharacter delimiters.

12.1.10 -csplit and -isplit Operators

As with most regex-enabled operators, a case-sensitive analog exists as `-csplit`. This works identically to `-split` with case-insensitive matching turned off. `-isplit` is also an alias for `-split`, provided to explicitly declare case-insensitivity.

Split Reference

You can view a complete reference for the `-split` operator at **Microsoft Docs**[4].

12.1.11 Select-String Cmdlet

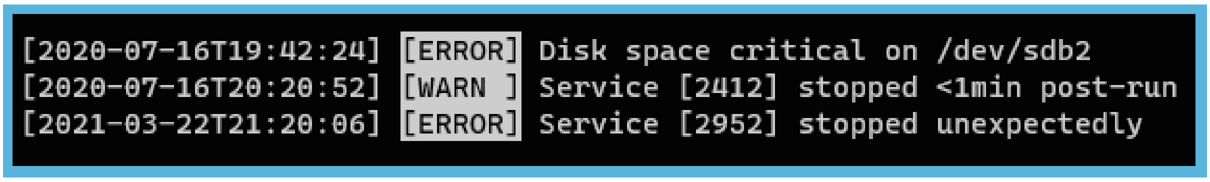
Another great feature in PowerShell is the `Select-String` cmdlet. Unlike with the `-match` and `-replace` operators, you can use this as part of a pipeline. You may liken it to *grep* in Unix-like

[4] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_split

environments.[5] [6] In its default mode, the cmdlet accepts regex patterns and returns `[MatchInfo]` objects. Since it accepts pipeline input, you can use `Select-String` to filter large data, such as logs. It also accepts one or more file paths as input, searching each file individually. In this mode, the cmdlet prepends the name of the file where it found each match.

Example 13: Displaying only error and warning lines in a log file

```
$SampleLog = 'Sample.log'

@'
[2020-07-16T19:35:46] [DEBUG] Connections waiting on [1508]:443
[2020-07-16T19:42:24] [ERROR] Disk space critical on /dev/sdb2
[2020-07-16T20:20:52] [WARN ] Service [2412] stopped <1min post-run
[2020-07-16T20:21:09] [INFO ] Service [2896] stopped
[2020-07-16T20:25:26] [DEBUG] Service [2896] ready
[2021-03-22T21:20:06] [ERROR] Service [2952] stopped unexpectedly
[2021-03-22T21:20:23] [INFO ] Closed [1663]:443
'@ | Out-File $SampleLog

Get-Content $SampleLog | Select-String '\[(?:ERROR|WARN )\]'
```



```
[2020-07-16T19:42:24] [ERROR] Disk space critical on /dev/sdb2
[2020-07-16T20:20:52] [WARN ] Service [2412] stopped <1min post-run
[2021-03-22T21:20:06] [ERROR] Service [2952] stopped unexpectedly
```

Example 13 Output

12.1.11.1 Changing Matching Behavior

Several features that change the matching mode for `Select-String` exist. Use the `-NotMatch` parameter to invert pattern matches, as with the `-notmatch` operator. Use the `-CaseSensitive` parameter to enable case-sensitive matching. It's also possible to use any of the inline options discussed in the Regex Deep Dive chapter.

When working with files via the `-Path` or `-LiteralPath` parameters, use the `-List` parameter to show only the first match in each file. This is an efficient way to retrieve a list of files matching your pattern at least once.[7] By default, `Select-String` only matches the first occurrence on each line. You can change this behavior with the `-AllMatches` parameter, and this works both with files and pipeline input.

12.1.11.2 Changing the Output Behavior

The `-Raw` and `-NoEmphasis` parameters of `Select-String` are available in PowerShell 7.0 and later.

[5] Adam Listek. (2020, Aug. 13). *How to Use PowerShell's Grep (Select-String)*. Adam The Automator. [Online]. Available: https://adamtheautomator.com/powershell-grep/. [Accessed: Jul. 11, 2021].

[6] PowerShell Team. (2008, Mar. 23). *Select-String and Grep*. Microsoft DevBlogs. [Online]. Available: https://devblogs.microsoft.com/powershell/select-string-and-grep/. [Accessed: Jul. 11, 2021].

[7] Microsoft. (2020, Nov. 17). *Select-String (Microsoft.PowerShell.Utility)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/select-string. [Accessed: Jul. 11, 2021].

The -Raw parameter used in the following example causes plain string output of the matches, instead of [MatchInfo] objects. To disable emphasis on the matches, but keep rich [MatchInfo] output, use the -NoEmphasis parameter.

Example 14: Extracting errors from log files with a specific range of dates

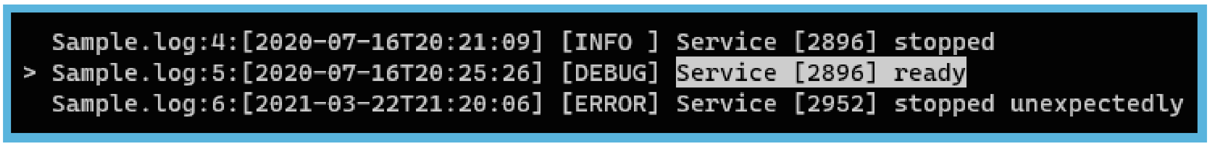
```
$FilteredLog = 'SampleLog-Errors-2021-Q1.log'
$Pattern = '^\[2021-0[1-3].+? \[ERROR\]'
Select-String $Pattern -Path $SampleLog -Raw | Set-Content $FilteredLog

Get-Content $FilteredLog
```

```
[2021-03-22T21:20:06] [ERROR] Service [2952] stopped unexpectedly
```

Another useful feature is the -Context parameter. This displays the lines immediately before and after the matching line.

Example 15: Displaying the context of matches with Select-String

```
$Pattern = 'Service \[\d+\] ready'
Select-String $Pattern -Path $SampleLog -Context 1
```



```
  Sample.log:4:[2020-07-16T20:21:09] [INFO ] Service [2896] stopped
> Sample.log:5:[2020-07-16T20:25:26] [DEBUG] Service [2896] ready
  Sample.log:6:[2021-03-22T21:20:06] [ERROR] Service [2952] stopped unexpectedly
```

Example 15 Output

In Example 15, the -Context parameter shows the log entries immediately before and after the matched line.

As with many cmdlets that accept file input in PowerShell, Select-String also has -Include, -Exclude, and -Encoding parameters. These let you filter the files searched and declare the encoding used to read them. Select-String supports plain text matching, too, using the -SimpleMatch parameter.

12.1.11.3 MatchInfo Object and the Matches Property

The Select-String cmdlet returns one [MatchInfo] object for each line in the input where it made a match. The properties of this object contain lots of information. The Line property is a plain string version of the matching line, while LineNumber, Path, and Context tell you about the location of the matches. The Context property contains a [MatchInfoContext] structure with information about the adjacent lines, but only if you used the -Context parameter.

Finally, the Matches property contains an array of all the underlying regex [Match] objects that the [MatchInfo] used for the line. You can learn more about [Match] objects in the second half of this chapter, Using the .NET Methods.

Example 16: Properties of the [MatchInfo] object

```
$MatchInfos = Get-Content $SampleLog -First 2 |
    Select-String -Pattern '\[.+?\]' -AllMatches

$MatchInfos.ForEach{
    '{0}, Line {1}: {2}' -f $_.Path, $_.LineNumber, $_.Line
    $_.Matches.ForEach{ "  Match: $($_.Value)" } # Match objects
}
```

```
InputStream, Line 1: [2020-07-16T19:35:46] [DEBUG] Connections waiting
on [1508]:443
  Match: [2020-07-16T19:35:46]
  Match: [DEBUG]
  Match: [1508]
InputStream, Line 2: [2020-07-16T19:42:24] [ERROR] Disk space critical
on /dev/sdb2
  Match: [2020-07-16T19:42:24]
  Match: [ERROR]
```

Select-String Reference

You can view a complete reference for `Select-String` at **Microsoft Docs**[8].

12.1.12 Where-Object Cmdlet with the -Match Parameter

You can also use regex with the `Where-Object` cmdlet to filter collections based on a pattern. This is equivalent to using the `-match` operator within a script block.

Example 17: Matching regex with Where-Object

```
Get-Alias | Where-Object { $_.Name -match '^s.{4}$' }

Get-Alias | Where-Object Name -Match '^s.{4}$'
```

```
CommandType Name                    Version Source
----------- ----                    ------- ------
Alias       sleep -> Start-Sleep
Alias       start -> Start-Process


CommandType Name                    Version Source
----------- ----                    ------- ------
Alias       sleep -> Start-Sleep
Alias       start -> Start-Process
```

[8] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/select-string

Like with regex operators, there are case-sensitive and inverse parameters, too. These are `-IMatch`, `-CMatch`, `-NotMatch`, `-INotMatch`, and `-CNotMatch`.

Where-Object Reference

You can view a complete reference for `Where-Object` at **Microsoft Docs**[9].

12.1.13 switch -Regex Statement

The `switch` statement can use regex—just pass the `-Regex` parameter before the value clause. In this mode, PowerShell processes each string match clause as a regex pattern. It processes non-string match clauses normally.

Example 18: `switch` statement with regex

```
$RandomNumber = Get-Random -Minimum 0 -Maximum 500

switch -Regex ($RandomNumber) {
    '\d{3}' { "3-digit number: $_"; break }
    '\d\d'   { "2-digit number: $_"; break }
    default  { "Single-digit number: $_" }
}
```

```
3-digit number: 438
```

Like with the operators and `Select-String`, the matches are case-insensitive by default. To use case-sensitive matching, pass the `-CaseSensitive` switch along with `-Regex`.

Switch Reference

You can view a complete reference for the `switch` statement at **Microsoft Docs**[10].

12.1.14 ValidatePattern() Parameter Attribute

When constructing advanced functions, you'll likely want to use regex to validate string parameters at some point. You could use one of the aforementioned operators, methods, or cmdlets inside a `[ValidateScript({...})]` attribute, but PowerShell has a solution already! The `ValidatePattern()` parameter attribute accepts a string pattern. When you pass a value via this parameter, the function accepts it only if the value matches the pattern.

[9] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/where-object

[10] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_switch

Example 19: ValidatePattern() attribute for function parameters

```
function MyFunction {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory = $true)]
        [ValidatePattern('^\w+\b\s*\b\w+$')]
        [string]
        $Words
    )
    Write-Host "The function ran with string = '$Words'"
}

MyFunction -Words 'hello world'
MyFunction -Words 'hello'
```

```
The function ran with string = 'hello world'

MyFunction: Cannot validate argument on parameter 'Words'.
The argument "hello" does not match the "^\w+\b\s*\b\w+$"
pattern. Supply an argument that matches "^\w+\b\s*\b\w+$"
and try the command again.
```

You can specify regex options and a custom error message, too.

```
[ValidatePattern(
    '^\w+\b\s*\b\w+$',
    Options = [System.Text.RegularExpressions.RegexOptions]::IgnoreCase -bor
    [System.Text.RegularExpressions.RegexOptions]::Multiline,
    ErrorMessage = 'The text "{0}" failed to match pattern "{1}"'
)]
```

You can learn more about Regex Options later in the chapter.

ValidatePattern Reference

You can view a complete reference for the `ValidatePattern()` attribute at **Microsoft Docs**[11].

12.1.15 Pester Should -Match and -MatchExactly Assertions

The *Pester* testing suite supports regex matches as part of its *Should* assertions. You can learn more about testing in PowerShell in Part II of this book, beginning with The AAA Approach.

There are two *Should* operators in Pester for regex matching. These are `-Match` and `-MatchExactly`. The first of these uses the `-match` operator for case-insensitive matching.[12] The second uses the `-cmatch` operator for case-sensitive matching.[13]

[11] https://learn.microsoft.com/en-us/powershell/scripting/developer/cmdlet/validatepattern-attribute-declaration

[12] Pester Team. (2021, Jun. 14). *Match.ps1.* L26. Pester/Pester on GitHub. [Online]. Available: https://github.com/pester/Pester/blob/main/src/functions/assertions/Match.ps1. [Accessed: Feb. 27, 2022].

[13] Pester Team. (2021, Jun. 14). *MatchExactly.ps1.* L19. Pester/Pester on GitHub. [Online]. Available: https://github.com/pester/Pester/blob/main/src/functions/assertions/MatchExactly.ps1. [Accessed: Feb. 27, 2022].

Example 20: Using regexes in Pester assertions

```
$PesterSample = 'PesterExample.ps1'
@'
Describe "Regex in Pester" {
    It "Matches the pattern" {
        'The sky is Blue.' | Should -Match 'sky\b.+\bblue'
    }
    It "Won't match due to capitals" {
        'The sky is Blue.' | Should -MatchExactly 'sky\b.+\bblue'
    }
}
'@ | Out-File $PesterSample

Invoke-Pester $PesterSample -Output Detailed
```

```
...

Describing Regex in Pester
  [+] Matches the pattern 2ms (1ms|1ms)
  [-] Won't match due to capitals 6ms (5ms|0ms)

...

Tests completed in 168ms
Tests Passed: 1, Failed: 1, Skipped: 0 NotRun: 0
```

These assertions work with the `-Not` parameter, too. `Should -Not -Match` -and `Should -Not -MatchExactly` produce inverse test results.

Pester Assertions Reference

You can view a complete reference for Pester assertions on the **Pester website**[14].

12.2 Using the .NET Methods

One benefit of PowerShell is the access it provides to .NET classes and methods, and this doesn't stop at regexes. The `[regex]` class, shorthand for `[System.Text.RegularExpressions.Regex]`, provides two major approaches:

1. Static methods for single use of patterns.
2. Class instances, initialized with a pattern, and instance methods to perform matching, replacing, or splitting operations as needed.

A few important points to consider when using the .NET methods are:

- Each method has several overloads. This chapter doesn't cover them all, but links to the comprehensive Microsoft documentation are available at the end of the section.

[14] https://pester.dev/docs/assertions/assertions

- None of the methods accept string arrays, so you must handle them programmatically.
- All the methods that this section describes accept `[RegexOptions]`[15] bitwise flags and, for recent .NET distributions,[16] a match time-out as a `[TimeSpan]`[17]. The Microsoft documentation describes which overloads support these.
- The methods are **case-sensitive** by default, unlike the native PowerShell operators. To use case-insensitive matching, you must use the `(?i)` inline option or `[RegexOptions]::IgnoreCase`.

12.2.1 Constructors

You can initialize regex class instances with `New-Object` or the `::new()` constructor.

Example 21: Initializing regex class instances

```
$MyRegex = New-Object Regex -ArgumentList '\[(?:ERROR|WARN *)\]'

# OR

$MyRegex = [regex]::new('\[(?:ERROR|WARN *)\]')
```

When creating regex class instances, you must pass a string pattern. You can optionally pass `[RegexOptions]` flags and a time-out.

Example 22: Initializing regex class instances with options and time-out

```
# Options equivalent to (?mi)
# Note the bitwise or operator used to combine options
$MyRegexOpts =
    [System.Text.RegularExpressions.RegexOptions]::Multiline -bor
    [System.Text.RegularExpressions.RegexOptions]::IgnoreCase

# 500 ms match time-out
$MyRegexTimeout = [timespan]::FromMilliseconds(500)

$MyRegex = [regex]::new(
    '\[(?:error|warn *)\]', $MyRegexOpts, $MyRegexTimeout
)

$MyRegex
```

```
              Options RightToLeft MatchTimeout
              -------- ----------- ------------
IgnoreCase, Multiline       False 00:00:00.5000000
```

12.2.2 IsMatch()

This method, similar to `-match`, returns a boolean value that indicates whether the pattern matched the input. Unlike `-match`, however, it doesn't capture any of the input if successful.

[15] https://learn.microsoft.com/en-us/dotnet/api/system.text.regularexpressions.regexoptions

[16] All .NET Core versions, all .NET versions (5.0+), and .NET Framework versions 4.5+.

[17] https://learn.microsoft.com/en-us/dotnet/api/system.timespan

Example 23: Using the IsMatch() method

```
$MyString = '[ERROR] /dev/sdc1 is full'
$MyPattern = '(?mi)\[(?:error|warn *)\]'

# Note the two colons (::) used below
# Use these to access a STATIC method
[regex]::IsMatch($MyString, $MyPattern)

$MyRegex = [regex]::new($MyPattern)
# Note the period (.) used below
# Use these to access an INSTANCE method
$MyRegex.IsMatch($MyString)
```

```
True
True
```

12.2.3 Match()

Unlike `IsMatch()`, this method collects match data if successful. This completes the missing functionality found in the `-match` operator and provides a more comprehensive data structure. `Match()` searches for the first pattern match in the text and returns a `[Match]`[18] object. To determine if the match was successful, use the `Success` boolean property.

Match Class in .NET

The `Match` class in .NET derives from the `Group` class, which itself derives from the `Capture` class. A `Match` instance populates itself with, and is functionally equivalent to, the first group, `Match.Groups[0]`. This represents the entire match, and its `Match.Value` property is equivalent to `Matches[0]` with the `-match` operator.

A `Group` instance populates itself with, and is functionally equivalent to, the last `Capture` of that group in the string: `Group.Captures[Group.Captures.Count - 1]`. This becomes relevant when there are many capturing groups in your pattern.

Perhaps the most important property provided by the `[Match]` object is `Groups`. This is a collection of all groups matched by the regex.

[18] https://learn.microsoft.com/en-us/dotnet/api/system.text.regularexpressions.match

Example 24: Using the Match() method with captures

```
$MyString = '[ERROR] /dev/sdc1 is full'
$MyPattern = '(?mi)^\[(?<type>error|warn *)\] *(?<msg>.+$)'

$MyMatch = [regex]::Match($MyString, $MyPattern)

$MyMatch.Success

$MyMatch.Groups | Select-Object -ExcludeProperty Groups, ValueSpan |
    Format-Table

'Message kind: {0}. Message: {1}' -f
    $MyMatch.Groups['type'].Value,
    $MyMatch.Groups['msg'].Value
```

```
True

Success Name Captures Index Length Value
------- ---- -------- ----- ------ -----
   True 0    {0}          0     25 [ERROR] /dev/sdc1 is full
   True type {type}       1      5 ERROR
   True msg  {msg}        8     17 /dev/sdc1 is full

Message kind: ERROR. Message: /dev/sdc1 is full
```

12.2.4 Match.NextMatch() Instance Method

There are two ways to look for and retrieve subsequent matches. One approach is to call the `NextMatch()` method on the latest `[Match]` object.

Example 25: Using the NextMatch() method to match all repeated letters

```
$MyMatch = [regex]::Match('aabccdde', '(?i)(\w)\1')

while ($MyMatch.Success) {
    'Found repeat: {0}' -f $MyMatch.Value
    $MyMatch = $MyMatch.NextMatch()
}
```

```
Found repeat: aa
Found repeat: cc
Found repeat: dd
```

12.2.5 Matches()

You can also look for all matches at once using the `Matches()` method and iterate over the resulting `[MatchCollection]`[19]. Each item in the collection is a `[Match]` object.

[19] https://learn.microsoft.com/en-us/dotnet/api/system.text.regularexpressions.matchcollection

Example 26: Using the Matches() method to match all repeated letters

```
$MyMatches = [regex]::Matches('aabccdde', '(?i)(\w)\1')

foreach ($match in $MyMatches) {
    'Found repeat: {0}' -f $match.Value
}
```

```
Found repeat: aa
Found repeat: cc
Found repeat: dd
```

`Matches()` uses lazy evaluation to populate the `[MatchCollection]` by default. This avoids expensive operations for complex patterns or many matches. However, if you attempt to access properties of the collection, such as `Count`, the engine populates all possible matches immediately.[20] You should therefore aim to use iterative statements such as `foreach` to process the result from a `Matches()` call.

12.2.6 Replace()

The `Replace()` method is similar in behavior to the `-replace` operator. It accepts several extra parameters, however, and these make it powerful for text processing.

Example 27: Using the Replace() method to replace lowercase 'm' words.

```
$MyString = 'May: The sunshine is mellow.'

[regex]::Replace($MyString, '\bm(\w+)\b', 'y$1')
```

```
May: The sunshine is yellow.
```

Once again observe that the .NET methods are case-sensitive by default—the word 'May' isn't replaced, unlike with `-replace` in Example 3.

12.2.6.1 Maximum Replacements and Offset

One advantage with `Replace()` is the ability to limit the number of replacements, and set a start point for them. You can achieve this with two more integer parameters, `count` and `startAt`, and they only work when `Replace()` is an instance method.

[20] Microsoft. (2021, Apr. 06). *MatchCollection Class (System.Text.RegularExpressions)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/api/system.text.regularexpressions.matchcollection#remarks. [Accessed: Nov. 15, 2021].

Example 28: Using Replace() with a max count and offset to change a log format

```
$MyString = '[21:20:06] [ERROR] Service [2952] stopped unexpectedly'

$MyRegex = [regex]::new('\[(\w+)\]')

# Max 1 replacement, starting at position 1 (2nd character)
# Skips first square brackets as this begins at position 0
$MyRegex.Replace($MyString, '$1:', 1, 1)
```

```
[21:20:06] ERROR: Service [2952] stopped unexpectedly
```

These parameters are also available, to varying degrees, in other .NET regex methods.

- `Match()` supports `beginning` (equivalent to `startAt`) and `length` (searches a specific number of characters)
- `Matches()` supports `startAt`
- `Split()` supports both `startAt` and `count`

12.2.6.2 Match Evaluators

Much like you can pass a script block to the `-replace` operator, the `Matches()` method accepts a delegate method to evaluate matches. In C#, this comes as a `MatchEvaluator` class instance, but you can cast a PowerShell script block to this type, and can therefore treat it similarly to `-replace`. The major difference is the lack of the automatic variable `$_` for the match, so you must define your script block with a parameter to receive the match. This example uses the same pattern as Example 4 to remove infixed capital letters for a limited number of words.

Example 29: Using Replace() with a script block evaluator

```
$MyString = 'This sentEnce has infixed cApital leTTErs.'

$MyRegex = [regex]::new('(?<!\A)\b[a-z]*[A-Z][A-Za-z]*\b')

$Evaluator = {
    param ($MyMatch)
    Write-Host "Run on '$MyMatch'"

    # Any matches are converted to lowercase
    ([string]$MyMatch.Value).ToLower()
}

# Max 2 replacements, starting at position 0 (1st character)
$MyRegex.Replace($MyString, $Evaluator, 2, 0)
```

```
Run on 'sentEnce'
Run on 'cApital'
This sentence has infixed capital leTTErs.
```

`Replace()` ignores the final word with infixed capitals (‘**leTTErs**’) as the call specifies a maximum of two replacements from position zero.

12.2.7 Split()

You can use the `Split()` method similarly to the `-split` operator, too. The advantage of the .NET method, however, is access to the same `count` and `startAt` parameters as `Match()`. This lets you decide the maximum number of splits and where in the input string to start the search.

Example 30: Using Split() to parse string data

```
$Headers = 'UUID', 'LastName', 'FirstName', 'TaxCode'
$MyRegex = [regex]::new('/')

$Data = @'
d8eb7ea9/9f83/4070/b989/a82025a575bd/Smith/John/1250/L
2ec4e828/449f/4f98/a182/b2980480a9eb/Doe/Jane/1250/L
785d343d/7fe0/4c29/837a/5e990826ca4b/Bloggs/Joe/290/LX
'@

# Separate data into lines with static Split()
$Lines = [regex]::Split($Data, '\r?\n')

$Lines.ForEach{
    # Create hashtable for employee data
    $entry = @{}

    # Separate into 4 columns max, starting at end of UUID
    $columns = $MyRegex.Split($_, 4, 36)

    # Iterate through columns and add to $entry in format:
    # ColumnName = Data
    for ($col = 0; $col -lt $columns.Count; $col++) {
        $entry[$Headers[$col]] = $columns[$col] -replace '/'
    }

    # Output employee data
    [PSCustomObject]$entry
}
```

```
LastName FirstName TaxCode UUID
-------- --------- ------- ----
Smith    John      1250L   d8eb7ea99f834070b989a82025a575bd
Doe      Jane      1250L   2ec4e828449f4f98a182b2980480a9eb
Bloggs   Joe       290LX   785d343d7fe04c29837a5e990826ca4b
```

Unlike with `-split`, you can’t pass a script block evaluator to `Split()`.

12.2.8 Escape() and Unescape()

It’s inevitable that you’ll have to generate regex patterns dynamically at some point. This opens up a lot of possibilities for invalid patterns and a wall of exceptions in your output. In these cases, `Escape()` becomes an important safeguard.

Example 31: Handling unknown delimiters in a regex pattern with Escape()

```
$MyData = @'
UID\Badge\Surname\Forename
f6b1b4b2-aeb9-45d7-b201-07b89065d448\15053636\Bloggs\Joe
bfb59409-5551-4863-8510-1d5cb5249294\94848749\Doe\Jane
4862a021-686e-4c40-b0b3-9c03a9c9687e\63847412\Smith\John
'@ -split '\r?\n'

if ($MyData[1] -imatch '^[\da-f\-]+(.+?)\d+\1\w+\1\w+$') {
    $Delimiter = $Matches[1]
} else {
    Write-Error -Message 'Couldn''t find delimiter'
}

Write-Host 'Will fail below here because of a bad regex pattern'
$Headers = $MyData[0] -split $Delimiter
Write-Host 'Will fail above here'

# Escape() corrects this
$Delimiter = [regex]::Escape($Delimiter)

$Headers = $MyData[0] -split $Delimiter

$MyData | Select-Object -Skip 1 | ForEach-Object {

    $entry = @{}
    $columns = [regex]::Split($_, $Delimiter)

    for ($col = 0; $col -lt $columns.Count; $col++) {

        $entry[$Headers[$col]] = $columns[$col]

    }

    [PSCustomObject]$entry

}
```

```
Will fail below here because of a bad regex pattern
OperationStopped:
Line |
  17 |  $Headers = $MyData[0] -split $Delimiter
     |  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     | Invalid pattern '\' at offset 1. Illegal \\ at end of pattern.
Will fail above here

UID                                  Badge    Forename Surname
---                                  -----    -------- -------
f6b1b4b2-aeb9-45d7-b201-07b89065d448 15053636 Joe      Bloggs
bfb59409-5551-4863-8510-1d5cb5249294 94848749 Jane     Doe
4862a021-686e-4c40-b0b3-9c03a9c9687e 63847412 John     Smith
```

The `Unescape()` method isn't as useful for dynamic regex patterns. However, it's a useful tool for restoring escaped patterns and converting regex Unicode sequences into printable text.

Example 32: Printing regex escape sequences with Unescape()

```
[regex]::Unescape('\[a\-b\]')
[regex]::Unescape('Hello\nworld')
[regex]::Unescape('\u00A9')
```

```
[a-b]
Hello
world
©
```

12.2.9 GetGroupNumbers() and GetGroupNames()

When you're working with more advanced patterns, it's sometimes necessary to enumerate the indices or names of capturing groups. This is also useful when debugging your expressions, as it provides clues on how the engine has interpreted the pattern.

Example 33: Matching an IPv4 address and displaying groups

```
$MyPattern =
    '^(?:(?<Octets>25[0-5]|2[0-4][0-9]|[01]?[0-9]{1,2})\.){3}' +
    '(?<Octets>25[0-5]|2[0-4][0-9]|[01]?[0-9]{1,2})$'

$MyRegex = [regex]::new($MyPattern)

$MyMatch = $MyRegex.Match('198.51.100.193')

$Counter = 1

$MyRegex.GetGroupNames()

$MyRegex.GetGroupNumbers()

$MyMatch.Groups['Octets'].Captures | ForEach-Object {
    "Octet $Counter = $_"
    $Counter++
}
```

```
0
Octets

0
1

Octet 1 = 198
Octet 2 = 51
Octet 3 = 100
Octet 4 = 193
```

In Example 33, the pattern uses the group name 'Octets' twice. Many regex implementations disallow this, but .NET and PowerShell permit it. The advantage here is that you can append captures from another group to those of an existing one. The example shows this, permitting a single iterative statement to retrieve all four decimal octets of the IPv4 address. Reusing a group name removes access to the original group, therefore backreferences to and replacements with the original definition aren't possible.

12.2.10 GroupNameFromNumber() and GroupNumberFromName()

It's also possible to retrieve the corresponding name or index from one another. Taking the `$MyRegex` pattern from the last example, the 'Octets' group corresponds to a group index of 1.

Example 34: Retrieving group names and indices from the other

```
$MyRegex.GroupNumberFromName('Octets')

$MyRegex.GroupNameFromNumber(1)
```

```
1

Octets
```

Group names are strings and group indices are 32-bit integers.

.NET Regex Reference

You can view a complete reference for **.NET's regex implementation**[21] and the ***regex* class**[22] at Microsoft Docs.

12.3 Regex Options

PowerShell and .NET offer a variety of mode-modifying options for regexes. This gives you extra control of how the engine interprets your patterns. You can view the available regex options by inspecting the `RegexOptions` enumeration:

```
[Enum]::GetNames([System.Text.RegularExpressions.RegexOptions])
```

Each heading below includes the enumeration for the regex option in parentheses.

12.3.1 RegexOptions.None (0)

`None` is the default regex options mode, and its enumeration is equivalent to zero.

12.3.2 RegexOptions.IgnoreCase (1)

The `IgnoreCase` option enables case-insensitive matching. This **isn't** the default mode within the .NET environment, but **is** with PowerShell's native operators, such as `-match`.

[21] https://learn.microsoft.com/en-us/dotnet/standard/base-types/regular-expressions

[22] https://learn.microsoft.com/en-us/dotnet/api/system.text.regularexpressions.regex

Example 35: Case-insensitive matching option

```
$MyString = 'Antonia is awesome!'
$MyPattern = 'an\w+'

[regex]::IsMatch($MyString, $MyPattern,
    [System.Text.RegularExpressions.RegexOptions]::None)

[regex]::IsMatch($MyString, $MyPattern,
    [System.Text.RegularExpressions.RegexOptions]::IgnoreCase)
```

```
False

True
```

12.3.3 RegexOptions.Multiline (2)

This option changes the interpretation of the caret `^` and dollar `$` anchors. By default, `^` matches the beginning of the string, and `$` matches the end, or before the final newline, of the string. In `Multiline` mode, however, these anchors match the beginning and end of the line. This enables matching within a multiline string, without first splitting the lines.

Example 36: Multiline matching option

```
$MyString = @'
52.44043425089714, -2.1832003074046074
35.838058817549914, 14.544060183946005
50.61179108261999, -3.4566678290794117
28.99748439438466, -13.489740628401105
'@
$MyPattern = '^35\.\d+, *14\.\d+$'

[regex]::IsMatch($MyString, $MyPattern, 'None')

[regex]::IsMatch($MyString, $MyPattern, 'Multiline')
```

```
False

True
```

You can use the string name of the `RegexOptions` enumeration for simpler, cleaner code.

12.3.4 RegexOptions.ExplicitCapture (4)

For longer and more complex patterns, it can be less confusing to use capturing groups `(...)` instead of non-capturing groups `(?:...)`. This is computationally expensive, however. The `ExplicitCapture` mode causes the engine to only capture **named groups**.

Example 37: Capturing only named groups with ExplicitCapture

```
function MatchWithOpts {
    param ($String, $Opts)

    Write-Output (
        [Environment]::Newline +
        $Opts + ':' + [Environment]::Newline +
        'Name Offset Value' + [Environment]::Newline +
        '---- ------ -----'
    )

    $MyMatch = [regex]::Match($String,
        '([A-Za-z]:)?(\\+(?<dir>[^\\/:*?"<>|]*))+', $Opts)

    $MyMatch.Groups.ForEach{
        $Group = $_.Name
        $_.Captures.ForEach{
            Write-Output (
                '{0,-5}{1,-7}{2}' -f $Group, $_.Index, $_.Value
            )
        }
    }
}

$MyString = 'C:\Program Files\PowerShell\7\Modules'

MatchWithOpts $MyString 'None'

MatchWithOpts $MyString 'ExplicitCapture'
```

```
None:
Name Offset Value
---- ------ -----
0    0      C:\Program Files\PowerShell\7\Modules
1    0      C:
2    2      \Program Files
2    16     \PowerShell
2    27     \7
2    29     \Modules
dir  3      Program Files
dir  17     PowerShell
dir  28     7
dir  30     Modules

ExplicitCapture:
Name Offset Value
---- ------ -----
0    0      C:\Program Files\PowerShell\7\Modules
dir  3      Program Files
dir  17     PowerShell
dir  28     7
dir  30     Modules
```

12.3.5 RegexOptions.Compiled (8)

When you initialize a regex, the engine parses the pattern and generates a sequence of operations. It then stores the sequence and later interprets it to perform matching operations. The `Compiled` option can improve performance by compiling the sequence into Common Intermediate Language (CIL) bytecode.[23] The runtime can then directly execute the instructions.[24]

The caveat to this approach is that compilation is expensive. Compiled regexes improve runtime performance at the cost of initialization. If you intend to use a regex lots of times, you may find performance improvements with the `Compiled` option.

12.3.5.1 Static Regex Methods

It's also possible to use the `Compiled` option with static regex methods. The regex engine caches the patterns you use in static method calls. This means that the `Compiled` option can offer a performance improvement for static method calls, too. This option causes the engine to cache the compiled CIL bytecode instead of the interpreted opcodes.

You can get or set the static cache size for the engine with `[regex]::CacheSize`.

12.3.6 RegexOptions.Singleline (16)

The `Singleline` option changes the interpretation of the period `.` character class. Normally, this class matches any character except the newline `\n`. With this mode enabled, the engine treats the input string as a single line, so `.` also matches a newline. This mode is useful when matching patterns across many lines.

Example 38: Capturing accross lines with Singleline

```
$MyString = @'
This string has a "quoted section
across two lines."
'@
$MyPattern = '".+?"'

[regex]::IsMatch($MyString, $MyPattern)

[regex]::IsMatch($MyString, $MyPattern, 'singleLINE')
```

```
False

True
```

Not passing any value to the regex options parameter is equivalent to passing `None`. This parameter is also case-insensitive when passing a string instead of `[System.Text.RegularExpressions.RegexOptions]`.

[23] Common Intermediate Language (CIL) was originally called Microsoft Intermediate Language (MSIL) before the Common Language Infrastructure (CLI) was standardized.

[24] Microsoft. (2021, Sep. 15). *Regular expression options - Compiled regular expressions*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-options#compiled-regular-expressions. [Accessed: Jan. 10, 2022].

12.3.7 RegexOptions.IgnorePatternWhitespace (32)

This option is useful for debugging or annotating your patterns, and for long patterns that would otherwise be hard to read. It causes the engine to ignore any white space in your patterns. You can still match white space or the space character within character classes, such as `\s`, `[ ]`. You can also match space characters using a backslash escape.

You can rewrite the pattern from Example 37 in a more human-readable format. The `IgnorePatternWhitespace` option also supports comments with a hash `#` symbol. The engine ignores all text after a `#` until the end of the line. You can learn more about regex comments and deconstructing your patterns for readability in the Regex Deep Dive chapter.

Example 39: IgnorePatternWhitespace option

```
$NoWhiteSpace = [regex]::new('([A-Za-z]:)?(\\+(?<dir>[^\\/:*?"<>|]*))+')
$WithWhiteSpace = [regex]::new(@'
(               # Unnamed capturing group
    [A-Za-z]        # Match uppercase/lowercase letter a-z
    :               # Match colon
)?              # Match this group 0/1 times (optional)
(               # Unnamed capturing group
    \\+             # Match one or more backslashes
    (?<dir>         # Named capturing group 'dir'
        [^              # Match anything but these characters
        \\/:*?"<>|      # NTFS disallowed chars
        ]*              # Match this class zero or more times
    )               # Match group 'dir' exactly once
)+              # Match this group 1 or more times
'@, 'IgnorePatternWhitespace')

$MyString = 'C:\Program Files\PowerShell\7\Modules'

$NoWhiteSpace.IsMatch($MyString)

$WithWhiteSpace.IsMatch($MyString)
```

```
True

True
```

There are some limitations to this mode. You can't put white space between characters that make up language elements, including:

- Quantifiers `{min,max}`
- Group initializers `(?<name>`, `(?:`, `(?<!`, `(?#` etc.
- Unicode classes `\p{name}`
- Backslash escapes `\...`

12.3.8 RegexOptions.RightToLeft (64)

In `RightToLeft` mode, the engine searches the input string backwards, from the last character to the first. It's important to note that the engine still builds pattern matches the same way. The forwards direction has just changed from the perspective of the engine. Therefore, last-to-first mode is a more appropriate description.

You should construct your patterns for this mode in the same way as you would for the standard mode. The two most significant differences you'll observe are:

- The order in which the engine finds matches
- The behavior in response to greedy and lazy quantifiers

Example 40: Matching from the end of the string wth RightToLeft

```
$Text = 'a1b1c1a2b2c2'
$TwoChars = '.{2}'
$Greedy = '(?<L>.+)(?<R>.+)'

Write-Host (
    'First pair, no options: ' +
    [regex]::Match($Text, $TwoChars, 'None').Value
)

Write-Host (
    'First pair, right-to-left: ' +
    [regex]::Match($Text, $TwoChars, 'RightToLeft').Value
)

$GreedyLtr = [regex]::Match($Text, $Greedy, 'None')
Write-Host (
    'Greedy sharing, no options:' + [Environment]::NewLine +
    '  $1 = ' + $GreedyLtr.Groups[1].Value + [Environment]::NewLine +
    '   L = ' + $GreedyLtr.Groups['L'].Value + [Environment]::NewLine +
    '  $2 = ' + $GreedyLtr.Groups[2].Value + [Environment]::NewLine +
    '   R = ' + $GreedyLtr.Groups['R'].Value
)

$GreedyRtl = [regex]::Match($Text, $Greedy, 'RightToLeft')
Write-Host (
    'Greedy sharing, right-to-left:' + [Environment]::NewLine +
    '  $1 = ' + $GreedyRtl.Groups[1].Value + [Environment]::NewLine +
    '   L = ' + $GreedyRtl.Groups['L'].Value + [Environment]::NewLine +
    '  $2 = ' + $GreedyRtl.Groups[2].Value + [Environment]::NewLine +
    '   R = ' + $GreedyRtl.Groups['R'].Value
)
```

```
First pair, no options: a1
First pair, right-to-left: c2
Greedy sharing, no options:
  $1 = a1b1c1a2b2c
   L = a1b1c1a2b2c
  $2 = 2
   R = 2
Greedy sharing, right-to-left:
  $1 = a
   L = a
  $2 = 1b1c1a2b2c2
   R = 1b1c1a2b2c2
```

The engine processes both the tokens and the string in last-to-first order. Since both are reversed, matches are still coherent. For instance, in Example 40, the `RightToLeft` match is '**c2**' as found in the string, not '**2c**'.

Because the engine is searching from the end of the string to the beginning, a greedy token later in your pattern matches first. Therefore, two greedy tokens capable of matching the same characters will share these differently in `RightToLeft` mode.

The index assignment for captures is still left-to-right, however. The left `L` group receives a group number of 1, and the `R` group, 2, in both modes. Lookarounds don't change their direction, either. A lookahead will always match characters *after* it, and a lookbehind will always match characters *before* it. The final important point is that the start index, available in some .NET methods, is still an offset from the **start** of the string. The difference is that the engine searches backwards from this index in `RightToLeft` mode, so it produces different behavior.

12.3.9 RegexOptions.ECMAScript (256)

The .NET and PowerShell regex engine supports an alternate operating mode designed to imitate ECMAScript, the standardized language underpinning JavaScript. In this mode, the engine behaves differently in three major ways.[25]

12.3.9.1 No Unicode Support

In contrast with many regex implementations, .NET regex supports Unicode matching in its default state. This isn't the case in `ECMAScript` mode, and several character classes differ in what they match:

- `\w` matches only `[A-Za-z0-9_]`
- `\s` matches only `[ \f\n\r\t\v]`
- `\d` matches only `[0-9]`
- `\p{...}` isn't a valid language element

[25] Microsoft. (2021, Sep. 15). *Regular expression options - ECMAScript matching behavior.* Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-options#ecmascript-matching-behavior. [Accessed: Nov. 20, 2021].

12.3.9.2 Interpretation of Numeric Backslash Escapes

Usually, when a single decimal digit follows a backslash, it's *always* interpreted as a backreference. If a capture with the numeric name doesn't exist, the engine throws an exception. In `ECMAScript` mode, a nonexistent capture results in the interpretation of a literal digit, instead. For more than one digit following a backslash, the engine interprets a decimal backreference. If a capture with that index doesn't exist, it assumes an octal character code up to `\377`, with trailing digits interpreted literally. In `ECMAScript` mode, the engine attempts to find a backreference with as many octal digits as possible and convert them to decimal. If this doesn't exist, it assumes an octal character code up to `\377`, with trailing digits interpreted literally.

12.3.9.3 Self-Referencing Capturing Groups

In `ECMAScript` mode, the engine updates any captures that include backreferences to themselves on each iteration. This enables a self-backreference to match part of a capture in the first iteration of the capture.

The following example shows the difference in behavior between `ECMAScript` mode and the canonical regex mode.

Example 41: Matching self-backreferencing captures in ECMAScript mode

```
$MyPattern = '(\d\1)+'

'NORM, 1:   ' + [regex]::Match('1', $MyPattern).Value
'ECMA, 1:   ' + [regex]::Match('1', $MyPattern, 'ECMAScript').Value

'NORM, 11:  ' + [regex]::Match('11', $MyPattern).Value
'ECMA, 11:  ' + [regex]::Match('11', $MyPattern, 'ECMAScript').Value

'NORM, 111: ' + [regex]::Match('111', $MyPattern).Value
'ECMA, 111: ' + [regex]::Match('111', $MyPattern, 'ECMAScript').Value
```

```
NORM, 1:
ECMA, 1:   1

NORM, 11:
ECMA, 11:  1

NORM, 111:
ECMA, 111: 111
```

12.3.10 RegexOptions.CultureInvariant (512)

This option changes the behavior of the `IgnoreCase` option. Under normal circumstances, the engine uses the current culture to determine which uppercase and lowercase characters are equivalent.[26] For some comparisons, such as file paths or URIs, differences between culture modes

[26] Microsoft. (2021, Sep. 15). *Regular expression options - Comparison using the invariant culture.* Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-options#compare-using-the-invariant-culture. [Accessed: Nov. 19, 2021].

can be problematic. The *invariant culture* circumvents this, using a predetermined character set indifferent to the current culture.

To retrieve the current culture in a PowerShell session, use `$Host.CurrentCulture`.

12.3.11 Combining Regex Options

You can combine regex options in several ways. Some options are only effective in combination with others, and some are invalid when others are present. The Microsoft documentation describes these comprehensively, but a couple of major points are:

- Only `IgnoreCase` and `Multiline` are valid with `ECMAScript`
- `CultureInvariant` only applies when `IgnoreCase` is present

To combine bitwise `[System.Text.RegularExpressions.RegexOptions]` flags, use the `-bor` operator. Applying bitwise logic, you can also remove an option by creating a filter mask with `-bnot`. You may also use the numeric result, and cast this to the `RegexOptions` type. Finally, you can cast a string with comma-separated names (case-insensitive) to the `[RegexOptions]` type.

Example 42: Combining bitwise regex options

```
[System.Text.RegularExpressions.RegexOptions]::Multiline -bor
    [System.Text.RegularExpressions.RegexOptions]::IgnoreCase

# Regex options use 10 bits, with bit 3 (128) unused
# IgnoreCase = 0000000001 = 1
# Multiline  = 0000000010 = 2
# Both       = 0000000011 = 3
[System.Text.RegularExpressions.RegexOptions]3

[System.Text.RegularExpressions.RegexOptions]3 -band -bnot
    [System.Text.RegularExpressions.RegexOptions]::IgnoreCase

[System.Text.RegularExpressions.RegexOptions]'multiline, IGNORECASE'
```

```
IgnoreCase, Multiline

IgnoreCase, Multiline

Multiline

IgnoreCase, Multiline
```

12.3.12 Inline Options

You can turn five of the options discussed in this chapter on or off from within a regex pattern, using inline options. The Regex Deep Dive chapter covers inline options.

Regex Options Reference

You can view a complete reference for **Regex Options**[27] at Microsoft Docs.

[27] https://learn.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-options

13. Regex Deep Dive

After reading Accessing Regexes, you should have a general understanding of how to use regexes in PowerShell. This chapter takes you deeper into the topic, with advanced syntax, replacement patterns, and debugging.

13.1 Debugging Your Regex Patterns

So you've designed a regex pattern. Great! But does it *actually* do what you want it to?

This section covers the regex-related errors you can expect from PowerShell and provides some tips on deconstructing your patterns from the perspective of the regex engine.

Example 1: A common regex error shown in PowerShell

```
'$12.00 ($12.99 inc. tax)' -match '(\$(\d+\.\d{2}) inc. tax'
#                                  ^
# Mistake: Literal bracket should be escaped: \(
```

```
OperationStopped: Invalid pattern '(\$(\d+\.\d{2}) inc. tax'
at offset 23. Not enough )'s.
```

OperationStopped: Invalid pattern

When you see the *Invalid pattern* error, it means there's something wrong with your regex. Common causes of invalid patterns include:

- Not escaping metacharacters such as brackets `()[]`, anchors `^$`, quantifiers `?*+`, or the backslash `\`.
- Not closing a group with `)` or character class with `]`.
- Referencing a nonexistent capturing group name or index `(a).+\2`.
- Reversing a range reference in a character class `[z-a]`.
- Using an escape where you don't need one `he\x digits`.
- Not escaping special PowerShell characters in expandable strings `"\$(\d)"`.

However, not all mistakes cause invalid patterns. Many erroneous patterns are *valid* regexes but don't behave as intended.

Example 2: Regex mistakes don't always generate errors

```
'$12.00 ($12.99 inc. tax)' -match '\($(\d+\.\d{2}) inc. tax'
#                                    ^
# Mistake: Literal dollar sign should be escaped: \$

'$12.00 ($12.99 inc. tax)' -match '\(\$(\d+\.\d{2}) inc. tax'
$Matches[1]
```

```
False

True
12.99
```

In Example 2, an intended literal dollar sign `$` isn't escaped, and the engine interprets this as an end of string anchor. This is still a valid pattern, but it'll never succeed because the tokens after the `$` are unable to match the pattern. Therefore, it's important to think from the perspective of the regex engine to construct effective patterns.

Other than those previously mentioned, causes of valid but erroneous patterns include:

- Not escaping tokens with special meanings in regex, for example `$3.50` (anchor)
- Not escaping special PowerShell characters in expandable strings, for example `"\$3.50"` (`$3` PowerShell variable)
- Trying to escape special PowerShell characters in literal strings, for example ``'\`$(\d)'`` (literal backtick in regex pattern)
- Inserting commas between ranges in a character class, for example `[a-z,0-9]`
- Not using the correct capitalization when case-sensitivity is on
- Unintentionally inserting white space into a pattern
- Not considering line breaks and their interactions with multiline mode
- Zero-length matches (zero-or-more quantifier within alternation)
- Confusing the numerical order of capturing groups

13.1.1 Regex Through the Eyes of an NFA Engine

The regex engine in PowerShell and .NET is of the nondeterministic finite automaton (NFA) type.[1] In practical terms, this means that the operations of the engine are driven by the regex pattern, not the text itself. This is in contrast to deterministic regex engines (DFA), which are text-controlled. The mathematical and computational theory behind these differences is beyond the scope of this book, but a practical understanding of the implications suffices to help you navigate .NET regex.

Watch a regex engine step through the pattern and string in Example 2 from the Regex 101 chapter at the Regex 101 Website[2]. This uses a different regex engine but is sufficiently similar for this example. At each step, the debugger highlights the current token in the regex pattern, along with the current match.

[1] Microsoft. (2021, Sep. 15). *Details of regular expression behavior*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/standard/base-types/details-of-regular-expression-behavior. [Accessed: Jan. 10, 2022].

[2] https://regex101.com/debugger?flags=i&flavor=pcre2&regex=Monday%7CTuesday%7CWednesday%7CThursday%7CFriday%7CSaturday%7CSunday&testString=It%20rained%20on%20Friday,%20but%20Monday%20will%20be%20clear.

13.1.2 Backtracking and Branching

So, what’s backtracking? Consider the pattern in Example 2 from the Regex 101 chapter. The pattern contains alternation for each day of the week, and the engine can decide which alternate to use for the match attempt.

Regex 101, Example 2

```
$MyString = 'It rained on Friday, but Monday will be clear.'
$MyPattern = 'Monday|Tuesday|Wednesday|Thursday|Friday|Saturday|Sunday'

$MyString -match $MyPattern
```

The regex engine steps through `$MyPattern` one token at a time, attempting to match each to the current *target* in the input text.

1. The engine finds no matches for any alternatives on the first character ‘I’, so it *backtracks* to the beginning of the pattern and moves on.
2. When the engine reaches the second character ‘t’, it matches `T` in the `Tuesday` alternative, as the `-match` operator is case-insensitive.
3. The next (third) character, a space (0x20), doesn’t match `u` in `Tuesday`, however, so the engine continues to the other alternatives.
4. The same happens when the engine tries the `Thursday` alternative.
5. The engine finds no matches for any alternatives on the following characters (3 to 13) ‘ rained on ‘ so *backtracks* to the beginning of the pattern for each.
6. When the engine reaches the ‘F’ in ‘Friday’, it matches the `F` in the `Friday` alternative, and so it tries the next element `r` against the next character ‘r’.
7. This matches too, and the engine continues matching elements to characters until `Friday` in the pattern matches ‘Friday’ in `$MyString`.
8. The `-match` operator stops at the first match, so the engine stops and returns `$true`.

Backtracking occurs in this scenario because the pattern contains an alternation construct (which is a fancy way to say “either/or patterns”). These decisions are the basis of backtracking in NFA engines. Unlike a DFA engine, which tracks all matches as it moves along the text, an NFA engine processes each token sequentially and remembers these decision points (backtracking positions). If the pattern can’t match at a later point, the engine will backtrack to this point and try the next decision.[3]

This effectively creates a new *branch* of possibilities, and the engine follows this branch until either it makes a match or reaches the end of the pattern. The engine only gives up when it has exhausted these avenues by trying each decision in turn. This way, it tries every permutation at least once. You may think that this has the potential to take a lot of time. You’d be right.

13.1.3 Catastrophic Backtracking

While this exhaustive approach to matching ensures that the engine finds any viable matches, many backtracking points can create an exponentially large number of permutations.

[3] Microsoft. (2021, Sep. 15). *Backtracking in Regular Expressions*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/standard/base-types/backtracking-in-regular-expressions. [Accessed: Jan. 10, 2022].

Example 3: Catastrophic backtracking with crude GUID matching

```
$Opts = [System.Text.RegularExpressions.RegexOptions]::None
$Time = [timespan]::FromSeconds(5)

$Guid = '87db9d39-ddc8-413c-84ac-0be925a8230a'
$Pattern = '([0-9a-f]+-?)+\Z'

Write-Host "'$Guid'"
Write-Host (
    [regex]::IsMatch($Guid, $Pattern, $Opts, $Time)
)

$GuidSpace = $Guid + ' '

Write-Host "'$GuidSpace'"
Write-Host (
    [regex]::IsMatch($GuidSpace, $Pattern, $Opts, $Time)
)
```

```
'87db9d39-ddc8-413c-84ac-0be925a8230a'
True

'87db9d39-ddc8-413c-84ac-0be925a8230a '
MethodInvocationException: Exception calling "IsMatch" with "4" argument(s):
"The RegEx engine has timed out while trying to match a pattern to an input
string. This can occur for many reasons, including very large inputs or
excessive backtracking caused by nested quantifiers, back-references and
other factors."
```

The RegEx engine has timed out while trying to match a pattern to an input string.

When the input string is a valid GUID, the engine finds a complete match in less than a millisecond. However, an errant space changes things, and the 5-second time-out has to save the day.

The issue here is that the match fails. When this happens, the nested one-or-more `+` quantifiers cause catastrophic backtracking. Consider what happens when the engine has matched all but the space. In the examples below, the caret `^` symbol represents the current match target of the engine, and parentheses `()` represent match components of the outer group with its `+` quantifier.

```
'(87db9d39-)(ddc8-)(413c-)(84ac-)(0be925a8230a) '
      1        2      3      4         5       ^
```

The next token in the pattern is the end-of-string `\Z` anchor. Instead, the next character of the input string is a space. The match has failed, so the engine backtracks. The inner `+` quantifier, which was greedy and initially matched the entire hexadecimal block, gives up one character from its fifth match component (the last '*a*').

```
'(87db9d39-)(ddc8-)(413c-)(84ac-)(0be925a8230)a '
      1        2      3      4         5      ^
```

However, the entire group can match one or more hexadecimal digits, so it does.

```
'(87db9d39-)(ddc8-)(413c-)(84ac-)(0be925a8230)(a) '
      1        2      3      4         5       6 ^
```

Can you see where this is going? The engine is now effectively back where it was before, but with another permutation of match components. The fifth match component of the inner + quantifier gives up another character. These two unmatched characters now match as a repeat of the outer + quantifier.

```
'(87db9d39-)(ddc8-)(413c-)(84ac-)(0be925a823)(0a) '
      1        2      3      4         5      6  ^
```

Yet again, the match fails at the space character. This time, however, the last match component can also give up part of its match. The outer + quantifier can then match the final two characters in a separate component. Therefore, **two** new permutations have emerged.

```
'(87db9d39-)(ddc8-)(413c-)(84ac-)(0be925a823)(0)(a) '
      1        2      3      4         5      6  7 ^
```

When the fifth match component gives up another character, there are now **four** new permutations.

```
'(87db9d39-)(ddc8-)(413c-)(84ac-)(0be925a82)(30a) '
      1        2      3      4         5      6  ^
'(87db9d39-)(ddc8-)(413c-)(84ac-)(0be925a82)(30)(a) '
      1        2      3      4         5     6   7 ^
'(87db9d39-)(ddc8-)(413c-)(84ac-)(0be925a82)(3)(0a) '
      1        2      3      4         5     6  7  ^
'(87db9d39-)(ddc8-)(413c-)(84ac-)(0be925a82)(3)(0)(a) '
      1        2      3      4         5     6  7  8 ^
```

When the fifth match component gives up another character (‘*2*’), it creates another **eight** permutations. Each character given up doubles the number of new permutations for match components in 2^n fashion. Apply this logic to the whole input string, and it’s easy to see why the engine ran out of time.

Watch a regex engine step through this pattern without the space[4] and with the space[5] at the Regex 101 website.

Catastrophic backtracking is often the result of nested quantifiers. By thinking about what happens when a match fails, you can spot these scenarios and correct for them. So, what’s the fix in this case?

[4] https://regex101.com/debugger?flags=i&flavor=pcre2&regex=(%5B0-9a-f%5D%2B-%3F)%2B%5CZ&testString=87db9d39-ddc8-413c-84ac-0be925a8230a

[5] https://regex101.com/debugger?flags=i&flavor=pcre2&regex=(%5B0-9a-f%5D%2B-%3F)%2B%5CZ&testString=87db9d39-ddc8-413c-84ac-0be925a8230a%20

13.1.4 Atomic Groups

Assuming you want the same behavior, which is matching an unlimited number of hexadecimal blocks, the fix is to use atomic groups `(?>...)`. You may also see these referred to as nonbacktracking groups. When the engine matches one of these groups to some input text, it doesn't give up any part of its match.

Example 4: Avoiding catastrophic backtracking with atomic groups

```
$Opts = [System.Text.RegularExpressions.RegexOptions]::None
$Time = [timespan]::FromSeconds(5)

$Guid = '87db9d39-ddc8-413c-84ac-0be925a8230a'
$Pattern = '(?>[0-9a-f]+-?)+\Z'

Write-Host "'$Guid'"
Write-Host (
    [regex]::IsMatch($Guid, $Pattern, $Opts, $Time)
)

$GuidSpace = $Guid + ' '

Write-Host "'$GuidSpace'"
Write-Host (
    [regex]::IsMatch($GuidSpace, $Pattern, $Opts, $Time)
)
```

```
'87db9d39-ddc8-413c-84ac-0be925a8230a'
True

'87db9d39-ddc8-413c-84ac-0be925a8230a '
False
```

This time, when the engine can't match the space in the second string, none of the match components of the outer `+` quantifier give up characters. This only leaves stepping through the input string and attempting to start the match at later positions. The engine soon runs out of permutations to try, and processing stops.

Watch a regex engine step through this pattern with the trailing space at Regex 101[6].

13.2 Functionality to Consider

13.2.1 No Subroutines

A subroutine in regex is the ability to reuse a subexpression at a different point in the input string. This differs from backreferencing, which only reuses the capture from a capturing subexpression.

[6] https://regex101.com/debugger?flags=i&flavor=pcre2&regex=(%3F%3E%5B0-9a-f%5D%2B-%3F)%2B%5CZ&testString=87db9d39-ddc8-413c-84ac-0be925a8230a%20

Subroutines can significantly shorten patterns by reusing repetitive elements. .NET regex doesn’t have this capability, as is clear with the repetition in Example 33 from the Accessing Regexes chapter.

13.2.2 No Recursion

Recursive matching, available in some regex implementations, makes it possible to “step into” a new recursion level. Here, the engine reapplies either the entire pattern or a captured group at the current point in the input string. If this level reaches the recursive token again, the engine steps into a deeper recursion level. This functionality adds a lot of possibilities for regex matching, at the cost of simplicity and efficiency. .NET regex doesn’t support recursion but offers an alternative solution with *balancing groups*. You can learn more about balancing groups in the Advanced Subexpressions and Backreferences section of this chapter.

13.2.3 Possessive Quantifiers vs. Atomic Groups

In many regex implementations, you can prevent backtracking with possessive greedy quantifiers. Usually, you would specify this with an extra plus + sign after the quantifier. .NET doesn’t support them, but since they’re functionally equivalent to wrapping the token in an atomic group, there is no real disadvantage to this. Wherever you might use a possessive quantifier with another regex engine `\w++`, simply replace it with an atomic group, keeping the quantifier **inside** `(?>\w+)`. Atomic groups also have the advantage of working with lazy quantifiers `(?>\w+?)`.

13.2.4 Variable-Length Lookbehinds

One uncommon feature in .NET regex is the support for full regex syntax in all lookarounds. Lookbehind size variability is often limited to alternation, finite quantifiers, or both. While this is useful for improving the specificity of your patterns, it can also be inefficient. If the lookbehind can extend backwards indefinitely, the engine will have to check back to the beginning of the input string, and this can add a lot of processing time. You can learn more about lookarounds in the Advanced Subexpressions and Backreferences section of this chapter.

13.3 Deconstructing a Pattern

At some point, you’ll need to use a regex pattern you didn’t create or can’t remember creating. To understand how it works and troubleshoot matching problems, it’s useful to know how to break up a pattern and interpret it.

The first step is to make your pattern a bit more human-readable.

- Split subexpression/group definitions and custom classes into blocks
    - Indent the contents of blocks
- Put alternatives and pipes on individual lines

  - Indent the alternatives but not the pipes
  - Keep the pipes at the same indentation as the group opening/closing brackets, or the margin if outside any group

- Keep quantifiers together with their associated tokens where possible
- You can place each token-quantifier pair either on an individual line or group them together with related tokens

  - Break long sequences of tokens into individual lines with the same indentation

Example 5 shows the pattern from Example 33 of the Accessing Regexes chapter, rewritten in extended form. You could use this string as a regex pattern, so long as you pass the *extended/ignore pattern white space* flag, either inline as `(?x)` or with `[RegexOptions]::IgnorePatternWhitespace`. Example 39 from *Accessing Regexes* used this feature.

Example 5: Extended regex form for easier interpretation

```
^                       # Beginning of string
(?:                     # Noncapturing group
    (?<Octets>              # Capturing group named "Octets"
                            # First alternative, matches 250-255
        25[0-5]                 # Match "25" then 0-5
    |                       #
                            # Second alternative, matches 200-249
        2[0-4][0-9]             # Match "2" then 0-4 then 0-9
    |                       #
                            # Third alternative, matches 0-199
        [01]?                   # Match "1" or "0" optionally
        [0-9]{1,2}              # Match 0-9, 1 or 2 times
    )                       #
    \.                      # Match period/full-stop
){3}                    # Match this group 3 times
                        #
(?<Octets>              #     Exactly the same "Octets" group for 4th octet,
    25[0-5]                 # since .NET regex doesn't support
|                       #     subroutines/recursion
    2[0-4][0-9]             #
|                       #
    [01]?[0-9]{1,2}         #
)                       #
$                       # End of string
```

Once you've deconstructed your pattern, you can interpret it. As in Example 5, it's helpful to annotate each token or group of tokens. This gives you a set of human-readable instructions to step through. Think about how the engine is going to interpret each token from start to finish, and how different inputs could change this.

The engine tries alternatives in the order they appear in a pattern. The third alternative in the "Octets" group, which matches an optional 1/0 and two digits from zero through nine, can match the first two digits of blocks starting with 2xx, with the `[0-9]{1,2}` token. If this alternative appeared first, it could match 2xx numbers incompletely, leaving the last digit unmatched. For the first three blocks, a mandatory period character must follow. This would cause the match to fail and the engine would backtrack, trying the other alternatives and leading to the expected behavior. For the fourth block, however, the match of only two characters could stand, but only if no anchors or lookaheads were checking for further digits. In Example 5, the end of string anchor `$` prevents this unexpected behavior. If the pattern needed to extract IP addresses from the middle of a string, however, a different approach is necessary.

When using alternatives, consider how you order them. You can also use lookarounds or anchors to ensure an alternative is specific to the target match. Adding a negative lookahead for an extra digit `(?![0-9])` or a word boundary `\b` prevents the last block matching when extracting an IP address from a sentence.

Example 6: Avoiding phantom matches with alternatives

```
# The alternatives are reversed in this example,
# so the tokens matching 0-199 come first
$Template = '(?:(?<Octets>[01]?[0-9]{{1,2}}|2[0-4][0-9]|25[0-5])\.){{3}}' +
    '(?<Octets>[01]?[0-9]{{1,2}}{0}|2[0-4][0-9]|25[0-5])'

# Create 3 patterns using a format string,
# with "{" and "}" doubled up to escape them
$Nothing = $Template -f ''
$Lookahead = $Template -f '(?![0-9])'
$WordBoundary = $Template -f '\b'

$IpsToMatch = @(
    'Address: 198.51.100.42'
    'Address: 198.51.100.193'
    'Address: 198.51.100.254'
)

foreach ($IpAddress in $IpsToMatch) {
    [pscustomobject]@{
        Input        = $IpAddress
        Nothing      = [regex]::Match($IpAddress, $Nothing).Value
        Lookahead    = [regex]::Match($IpAddress, $Lookahead).Value
        WordBoundary = [regex]::Match($IpAddress, $WordBoundary).Value
    }
}
```

```
Input                   Nothing        Lookahead      WordBoundary
-----                   -------        ---------      ------------
Address: 198.51.100.42  198.51.100.42  198.51.100.42  198.51.100.42
Address: 198.51.100.193 198.51.100.193 198.51.100.193 198.51.100.193
Address: 198.51.100.254 198.51.100.25  198.51.100.254 198.51.100.254
```

Notice that for the final IP address, no anchor results from the last octet being ‘**25**’. Both a word boundary and a negative lookahead prevent this.

13.4 Advanced Syntax

13.4.1 Unicode Categories and Blocks

As discussed in the Accessing Regexes chapter, .NET regex supports Unicode out of the box. Many of the shorthand classes already match Unicode equivalents, including other writing and numeric systems. Matching Unicode characters directly is also possible with the `\uHHHH` escape, where HHHH is the hexadecimal UTF-16 code point (big-endian). For example, `\u00A9` matches the copyright © symbol (**U+00A9**) and `\u2021` matches the double dagger ‡ symbol (**U+2021**). You can match any character encoded by UTF-16 in the Basic Multilingual Plane (BMP, **U+0000–U+FFFF**).

You can also match Unicode blocks and categories with the `\p{...}` escape. To match a block, use `\p{Is...}` where ... is a block name. For example, `\p{IsLatin-1Supplement}` matches all characters in the Latin-1 Supplement block (**U+0080–U+00FF**), including the copyright symbol.

To match a category, use `\p{X}` or `\p{Xy}` where **X** and **Xy** are category and subcategory shorthands. For example, `\p{Po}` matches characters from the *Other Punctuation* subcategory, including the double dagger symbol. By extension, `\p{P}`, which matches characters from the *Punctuation* category, also matches the double dagger.

Example 7: Matching Unicode characters, blocks, and categories

```
$BEUnicode = [System.Text.Encoding]::BigEndianUnicode
$CopyrightSymbol = $BEUnicode.GetString(@(0x00, 0xA9))
$DDaggerSymbol = $BEUnicode.GetString(@(0x20, 0x21))

$CopyrightSymbol -match '\u00A9'
$CopyrightSymbol -match '\p{IsLatin-1Supplement}'

$DDaggerSymbol -match '\u2021'
$DDaggerSymbol -match '\p{Po}'
$DDaggerSymbol -match '\p{P}'
```

```
True
True

True
True
True
```

As with the word `\w`, decimal `\d`, and white space `\s` class shorthands, the Unicode class shorthand has an inverse `\P{...}`, which matches all but that category or block.

13.4.1.1 UTF-16

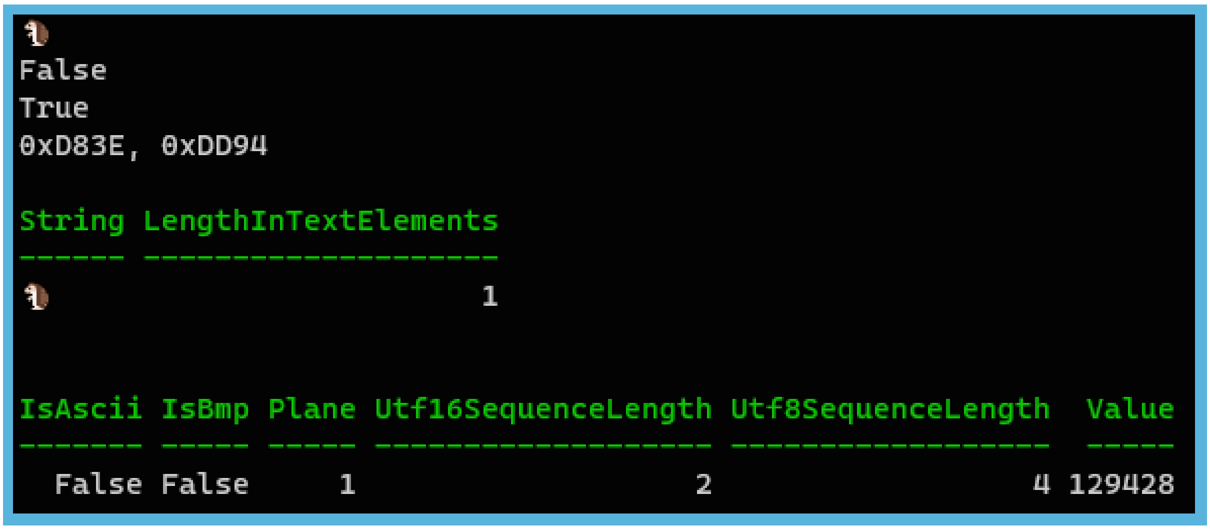
One important consideration when you are working with Unicode characters is that PowerShell and .NET operate with UTF-16 encoding. This applies to the regex engine and matching, too. The engine stores characters beyond the Basic Multilingual Plane (BMP, **U+0000–U+FFFF**) as UTF-16 surrogate pairs. These are special Unicode code points enabling two 16-bit code points to represent one beyond **U+FFFF**. Any Unicode code point that isn't a surrogate is known as a *scalar*. Each surrogate pair consists of:

- A high surrogate **U+D800–U+DBFF** (high 10 bits, 1024 characters)
- A low surrogate **U+DC00–U+DFFF** (low 10 bits, 1024 characters)

To determine the code point from a surrogate pair, the encoding system gets the high index from the high surrogate character by subtracting 0xD800, multiplying by 1024 (0x400), and adding 0x100000. This represents the code point range **U+100000–U+10FC00** in steps of 1024. The encoding system then gets the low index from the low surrogate by subtracting 0xDC00. This represents the additional range 0x0–0x3FF (0–1023). By adding these two values together, any code points in the range **U+100000–U+10FFFF** are possible.

When attempting to make a match, the regex engine will not examine the Unicode code point, instead it will examine the two UTF-16 surrogates. This means you can't use regex patterns for surrogate pairs in the way you would for BMP characters.

```
$HedgehogBytes = @(0xF0, 0x9F, 0xA6, 0x94)
$HedgehogEmoji = [System.Text.Encoding]::UTF8.GetString($HedgehogBytes)

# Show emoji
$HedgehogEmoji

# U+1F994 is in category Other Symbols (So), but match fails
$HedgehogEmoji -match '\p{So}'

# Matching high surrogate followed by low surrogate succeeds
$HedgehogEmoji -match '\p{IsHighSurrogates}\p{IsLowSurrogates}'

# Show surrogate codepoints
$HedgehogEmoji.ToCharArray().ForEach{
    '0x' + [convert]::ToString([int]$_, 16).ToUpper()
} -join ', '

# Interpret actual text elements from UTF-16 string
[System.Globalization.StringInfo]::new($HedgehogEmoji) | Format-Table

# Get Unicode scalar (non-surrogate code point)
$HedgehogEmoji.EnumerateRunes() | Format-Table
```



```
🦔
False
True
0xD83E, 0xDD94

String LengthInTextElements
------ --------------------
🦔                        1


IsAscii IsBmp Plane Utf16SequenceLength Utf8SequenceLength  Value
------- ----- ----- ------------------- ------------------  -----
  False False     1                   2                  4 129428
```

Output from Example 8

The `[System.Text.Rune]` class and `String.EnumerateRunes()` method is only present in PowerShell 6.0 (.NET 3.0) and later. You can use the `[System.Globalization.StringInfo]` class in Windows PowerShell to process individual graphemes.

Unicode Categories and Blocks Reference

You can view a complete reference for supported Unicode blocks[7] and categories[8] at Microsoft Docs.

.NET Character Encoding Reference

You can view a complete reference for Character Encoding in .NET[9] at Microsoft Docs.

13.4.2 Character Class Subtraction

Character class subtraction is a feature available in only a handful of regex implementations. With it, you can filter your custom character classes to restrict what they match.

Example 8: Character class subtraction

```
$Array = 'a' .. 'p'
-join $Array # Input string
-join ($Array -match '[a-k]') # Match a-k
-join ($Array -match '[a-k-[d]]') # Match a-k except d
```

```
abcdefghijklmnop
abcdefghijk
abcefghijk
```

The range operator, `..`, only works with characters beginning in PowerShell 6.0. In prior versions, the range operator only works with integers.

When subtracting character classes:

- Positive subtractions `-[...]` remove characters or ranges from the initial class.
- Inverse subtractions `-[^...]` remove **all** characters or ranges from the initial class except those in the subtraction class.
- You can subtract within a subtraction (nested subtractions work).
- You can't add characters that aren't in the initial class by including them in the subtraction (double-negatives don't work).
- The engine ignores characters or ranges not in the initial class (no errors for out-of-range subtractions).

[7] https://learn.microsoft.com/en-us/dotnet/standard/base-types/character-classes-in-regular-expressions#SupportedNamedBlocks

[8] https://learn.microsoft.com/en-us/dotnet/standard/base-types/character-classes-in-regular-expressions#SupportedUnicodeGeneralCategories

[9] https://learn.microsoft.com/en-us/dotnet/standard/base-types/character-encoding-introduction

Example 9: Limitations of character class subtraction

```
$Array = 'a' .. 'p'
-join ($Array -match '[a-k-[h-z]]') # Matches a-k except h-k, l-z ignored
-join ($Array -match '[a-k-[^j-z]]') # Matches nothing but j-k, l-z ignored
-join ($Array -match '[a-k-[b-g-[cd]]]') # Matches a-k, except b and e-g
```

```
abcdefg
jk
acdhijk
```

Character Class Subtraction Reference

You can view a complete reference for Character Class Subtraction[10] at Microsoft Docs.

13.4.3 Using Inline Options

.NET provides a range of regex option modifiers, and the Accessing Regexes chapter discusses them under the Regex Options heading. You can alter five of them within your patterns to change the behavior from that point forward, or only for a span. A helpful mnemonic for these is *msnix* ("MS-nix").

Table 1: Regex inline option modifiers

`[RegexOptions]` flag	Modifier	Name
`Multiline`	`m`	Multiline mode
`Singleline`	`s`	Singleline mode
`ExplicitCapture`	`n`	Explicit captures only
`IgnoreCase`	`i`	Case-insensitive mode
`IgnorePatternWhitespace`	`x`	Ignore pattern whitespace

There are two ways to apply these options to your patterns. The first applies them to the rest of the pattern after an option modifier. Turn an option on with a letter from Table 1, and off with a hyphen (minus sign) before that letter. The general form for this is `(?msnix-msnix)`.

[10] https://learn.microsoft.com/en-us/dotnet/standard/base-types/character-classes-in-regular-expressions#character-class-subtraction-base_group---excluded_group

Example 10: Inline option modifiers

```
$MyRegex = [regex]::new(@'
(?xm-i)     # Ignore white space and multiline ON, case insensitivity OFF
^           # Start of line because multiline is on
[a-z]{2}    # 2 lowercase letters
(?i)        # Turn on case insensitivity
-[a-z]{2}   # Hyphen followed by 2 letters of any case
'@, 'IgnoreCase')

$MyRegex.Matches(@'
en-US
en-us
EN-us
en-GB
'@).ForEach{ $_.Value }
```

```
en-US
en-us
en-GB
```

In Example 10, the regex option flags turn on `IgnoreCase`, as it would be with the `-match` operator. However, the `-i` modifier overrides this and disables case insensitivity within the pattern. The `m` modifier turns on multiline mode, allowing the pattern to match a culture code from each line, not just the one immediately following the beginning of the string.

The modifier takes effect at its current position in the pattern, so the modifier doesn't alter the behavior of any tokens that come before it.

Example 11: Specificity of inline option modifiers

```
$MyRegex = [regex]::new('(\w)(?n)(\w)(?-n)(\w)')

$MyRegex.Match('abc').Groups.ForEach{
    Write-Host $_.Name $_.Value
}
```

```
0 abc
1 a
2 c
```

In Example 11, the engine doesn't capture the second group but captures the first and third. Explicit capture is only on between `(?n)` and `(?-n)`, meaning the engine can capture the other unnamed groups.

13.4.4 Using Option Spans

You can also apply option modifiers within a confined span using subexpression syntax with modifiers and a colon. The general syntax for this is `(?msnix-msnix:...)`. Using this approach, the changed options only apply to the subexpression.

Example 12: Option spans (subexpressions)

```
$MyRegex = [regex]::new('(\w)(?-n:(\w))(\w+)', 'ExplicitCapture')

$MyRegex.Match('abc').Groups.ForEach{
    Write-Host $_.Name $_.Value
}
```

```
0 abc
1 b
```

In Example 12, `ExplicitCapture` is present, but the engine captures the second unnamed group. This is because the group is inside a subexpression with `ExplicitCapture` turned off.

Inline Options Reference

You can view a complete reference for Inline Options[11] at Microsoft Docs.

13.4.5 Comments in Regex

Like any programming language, many regex implementations support comments, also called *remarks*. .NET regex is no exception, and there are two forms of comments supported.

The first kind, the comment span `(?#...)`, is available regardless of which regex options are present. The second, the end of line comment `#...` applies only when `IgnorePatternWhitespace` is present.

Example 13: Comments in .NET regex

```
if ('abcdefg hijklmn' -match '(?:(\w)(?# Matches letter pairs)(\w))+') {
    $Matches
}
```

```
Name    Value
----    -----
2       f
1       e
0       abcdef
```

Remember that with `-match`, you only get the first match. You only have access to the last capture for each group, too. In Example 13, the captures for *ab* and *cd* aren't accessible. The second match for *hijklmn* and its captures isn't, either.

When the `IgnorePatternWhitespace` flag is present, you can use end of line comments. The engine ignores all text after a hash symbol `#` until the end of the line.

[11] https://learn.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-options#specifying-the-options

Example 14: End of line comments in .NET regex

```
$MyString = 'SWdlbCBzaW5kIHRvbGwh'

# Matches valid base-64 strings
$MyPattern = @'
(?nx)                   # Ignore pattern white space, explicit captures
(                       # Unnamed group
    [A-Za-z0-9+/]{4}        # Match 4 B64 chars (A-Z, a-z, 0-9, +, and /)
)+                      # Match one or more instances of group
(                       # Unnamed group
                        # 1st alternative
    [A-Za-z0-9+/]{3}=       # Match 3 B64 chars and "="
|                       #
                        # 2nd alternative
    [A-Za-z0-9+/]{2}==      # Match 2 B64 chars and "=="
)?                      # Match group optionally
'@

if ($MyString -match $MyPattern) {
    Write-Host ('Match: {0}' -f
        -join ([System.Convert]::FromBase64String($Matches[0]) -as [char[]]))
}
```

```
Match: Igel sind toll!
```

In Example 14, the `(?x)` inline option turns on `IgnorePatternWhitespace`.

13.5 Advanced Replacement Patterns

The previous chapters haven't discussed replacement patterns in depth. This is because the engine treats them differently and they use different language elements.

Replacement patterns are instructions for how the regex engine should substitute text in replacement operations. You may find them referred to as *substitution patterns* for this reason. The only metacharacter in replacement patterns is the dollar sign `$`. The engine interprets all other text literally. The character following the dollar sign changes the nature of the substitution.

13.5.1 Named and Numeric Captures

Any group in the regex pattern that led to a capture from the input string can be substituted in replacement operations. The syntax for substituting a capture by its numeric index is `$NNN`, where *NNN* is the ordered index of the capturing group in the pattern from left-to-right.

The syntax for named captures is a little different from backreferences. Use `${name}` where *name* is the group name of the named capture `<name>`.

Example 15: Reformatting log data using substitution in replacement patterns

```
$MyString = @'
[2020-07-16T19:50:31] [PATCH] Service "xrdp" installed by "apt"
[2020-07-16T20:25:23] [INFO ] Service [2896] started
[2020-07-16T20:25:26] [DEBUG] Service [2896] ready
'@

$Patterns = @(
    '(?m)^\[([^\]T]+)T([^\]]+)\] ' +
        '\[(INFO|PATCH|DEBUG|WARN|ERROR|FATAL) ?\] (?<msg>.+)$'
    '(?m)^(WARN|PATCH)(?= )'
    '(?m)^INFO(?= )'
    '(?m)^DEBUG(?= )'
)

$Replacements = @(
    '$3 message at $2 on $1{0}    ${{msg}}{0}' -f [Environment]::NewLine
    '$1ING'
    'INFORMATIONAL'
    'DEBUGGING'
)

for ($i = 0; $i -lt $Patterns.Count; $i++) {
    $MyString = $MyString -replace $Patterns[$i], $Replacements[$i]
}

$MyString
```

```
PATCHING message at 19:50:31 on 2020-07-16
    Service "xrdp" installed by "apt"

INFORMATIONAL message at 20:25:23 on 2020-07-16
    Service [2896] started

DEBUGGING message at 20:25:26 on 2020-07-16
    Service [2896] ready
```

Unlike backreferences in regex patterns, a nonexistent capture doesn't result in interpretation as an octal character code. Instead, the engine substitutes the literal text of the token. This is also the case with nonexistent named captures.

Example 16: Nonexistent captures in replacement patterns

```
$MyString = '$5.23 (March)'

$NoCaptures = '\$\d+\.\d{2} \(\w+\)'
$Captures = '\$(\d+)\.(?<discount>\d{2}) \((\w+)\)'

$Replacement = '$$$1.00 ($$0.${discount} off until ${2})'

$MyString -replace $NoCaptures, $Replacement

$MyString -replace $Captures, $Replacement
```

```
$$1.00 ($0.${discount} off until ${2})

$5.00 ($0.23 off until March)
```

Note that in the pattern, the *until* month group is the third capturing group from left-to-right, but is actually `$2` in the replacement pattern. This is because the regex engine only assigns numeric indexes to named captures **after** all unnamed ones.

In Example 16, the `$$` token inserts a literal dollar sign. The protected numeric capture reference `${NNN}` prevents the engine from interpreting any following digits as part of the group index. For instance, the replacement pattern `$10` represents the tenth capture, but `${1}0` represents the first capture, followed by a literal zero.

13.5.2 Entire Match

You can insert the value of the pattern's entire match, equivalent to capture `0`, using a dollar sign followed by an ampersand `$&`.

Example 17: Substituting the entire match

```
'Nice!' -replace 'ice', 'oice $&'
```

```
Noice ice!
```

13.5.3 Match Span Prefixes and Postfixes

What about the text before and after that matched by the pattern? The engine stores that, too. To insert all the text before the first character of the match (prefix), use a dollar sign followed by a backtick `` $` ``. For all the text after the last character of the match (postfix), use a dollar sign followed by a single quotation mark `$'`.

Example 18: Substituting before and after the match

```
'Nice!' -replace 'ice', '$`$`$`oice $&$''$''$''' # Literal

'Nice!' -replace 'ice', "$``$``$``oice $&$'$'$'" # Expandable
```

```
NNNNoice ice!!!!

NNNNoice ice!!!!
```

You must escape the backtick in expandable strings, and the single quotation mark in literal strings. You can achieve both escapes by doubling the relevant character.

13.5.4 Entire Input

You can substitute a copy of the entire input string before any replacement operations using `$_`. This isn't the same as the `$_` PowerShell automatic variable. You must pass a dollar sign followed by an underscore. Either use a literal string, `'$_'`, or escape the dollar sign in an expandable string, ``"`$_"``.

This is effectively equivalent to the prefix, match, and postfix tokens combined ``$`$&$'``.

13.5.5 Last Capture

The last token available for replacement patterns substitutes the last capture. This is the highest-numbered capture in the match.

Example 19: Substituting the last capture

```
$MyPattern = '(?m)^(?<First>[\w-[\d]]+)((?: [\w.-[\d]]+)+)? ([\w--[\d]]+)$'
$MyReplacement = '"$2, $+$1"'

$Names = @'
John Smith
Jane Luisa Doe
Joe D. Bloggs
Joe Smith-Bloggs
'@

$Names -replace $MyPattern, $MyReplacement
```

```
"Smith, John"
"Doe, Jane Luisa"
"Bloggs, Joe D."
"Smith-Bloggs, Joe"
```

In Example 19, note how the last name is capture 2, and the first name is capture 3. This is for the same reason as in Example 16. The .NET regex engine indexes named captures **after** all unnamed ones. The *first name* group is therefore the last capture, and accessed with `${First}`, `$3` or `$+`.

13.6 Advanced Subexpressions and Backreferences

13.6.1 Backreferences in Depth

The previous regex sections have mentioned backreferences already, but there's a little more to them with .NET regex. You can match captures from earlier in your regex patterns using backreferences, both by number and name. These aren't the same as subroutines, where the *pattern* gets reapplied, instead of the *capture*. Subroutines don't exist in .NET regex.

For numeric backreferences, use `\NNN` where *NNN* is an integer **starting at 1**, representing the ordered index of the capture from the pattern from left-to-right. For the pattern `(\d+)(%)` and input string '*15%*', `\1` would match '*15*' and `\2` would match '*%*'. For named backreferences, use `\k<name>` or `\k'name'` where *name* is the group name `(?<name>...)`.

The engine gives **all** captures a numeric index regardless of whether the group is named, and you can access these with `\NNN` or `\k<...>`. When accessing a nonexistent backreference using a numeric index, the engine will treat it as an octal character code instead. The Accessing Regexes section discusses backreference interpretation under the ECMAScript Mode heading.

Example 20: Named and numeric backreferences

```
$Numbers = '12230'

# Matches repeating digits
$Numbers -match '(\d)\1'

# Tries to match capture 130, which doesn't exist,
# so matches octal 130 (capital "X")
$Numbers -match '(\d)\130'

# Use an index enclosed with triangular bracket or
# single quotation mark to prevent the above issue
$Numbers -match '(\d)\<1>30'
$Numbers -match "(\d)\'1'30"

# Passing numeric indexes to \k results in the indexed capture,
# unless a capturing group was explicitly named with that number
$Numbers -match '(\d)\k<1>30'
$Numbers -match "(\d)\k'1'30"

# You can use either enclosed form for both
# the capturing group and the backreference
$Numbers -match "(?'repeat'\d)\k<repeat>"
$Numbers -match "(?<repeat>\d)\k'repeat'"
```

```
True

False

True
True

True
True

True
True
```

If you use a numeric name `(?<NNN>)` for your capturing group, it overrides the internally assigned index. Doing this means the capture names will no longer reflect the order of capturing groups in the pattern. You probably shouldn't do this.

Example 21: Named numeric captures and their consequences

```
'5:5:' -match "(?'2'\d)\1" # Capture 1 doesn't exist, throws error

'5:5:' -match "(?'2'\d)(:)\1\2" # Expected order of captures
'5:5:' -match "(?'2'\d)(:)\2\1" # Actual order of captures
```

```
OperationStopped: Invalid pattern '(?'2'\d)\1' at offset 10.
Reference to undefined group number 1.

False
True
```

Capturing group names are always **case-sensitive**. This applies whether the *IgnoreCase* flag is present (such as with `-match`) or not.

Backreference Constructs Reference

You can view a complete reference for Backreference Constructs[12] at Microsoft Docs.

13.6.2 Lookarounds in Depth

As described earlier, lookarounds can determine the success or failure of a match without becoming a part of it. In other words, they're zero-width assertions. Lookarounds are also atomic, and the engine can't backtrack into them once processing moves on.

Example 22: Using a negative lookaround to isolate line comments

```
$MyPattern = '(?nm)^(?!\s*#).+(\r?\n)*'
$MyString = @'
# Create the RNG
$CryptoRandGen = [System.Security.Cryptography.RNGCryptoServiceProvider]::new()

# Buffer to store random bytes
[byte[]]$bufferByte = [byte[]]::new(1)
'@

$MyString -replace $MyPattern
```

```
# Create the RNG
# Buffer to store random bytes
```

The function of the negative lookahead here is to exclude all lines beginning with zero or more space characters followed by a hash symbol `#`. All other lines match the `.+` token, as do any following newlines. The `-replace` operator replaces these matches, leaving only the comment lines and adjacent newlines.

You can also use anchors in lookarounds, as they're already zero-width. You can negate a start-of-line anchor `(?!^)` in multiline mode, for example.

[12] https://learn.microsoft.com/en-us/dotnet/standard/base-types/backreference-constructs-in-regular-expressions

Example 23: Finding nouns and articles with lookarounds

```
$MyPattern = '(?im)(?<=(?<article>a|the) )(?>(?<noun>\w+))(?<!ing)(?!$)'
$MyString = @'
The cat sat on the mat eating a hat
The knitting needles fell on the floor again
'@

$MyMatches = [regex]::Matches($MyString, $MyPattern)

$MyMatches.ForEach{
    'Match "{0}"' -f $_.Value
    $_.Groups.Where{
        $_.Name -ne '0' # Exclude the $0 'whole match' group
    }.ForEach{
        ' ', $_.Name, '=', $_.Value -join ' '
    }
}
```

```
Match "cat"
  article = The
  noun = cat
Match "mat"
  article = the
  noun = mat
Match "floor"
  article = the
  noun = floor
```

Don’t get spooked! This pattern is a lot simpler than it looks. Breaking it down:

- `(?im)`: Option modifiers; `IgnoreCase` and `Multiline`
- `(?<=(?<article>a|the) )`: Positive lookbehind containing named capturing group ‘article’. Matches ‘a’ or ‘the’ and a space. This means ‘*a* ‘ or ‘*the* ‘ **must** precede the following noun (note the space that follows both).
- `(?>(?<noun>\w+))`: Atomic group containing named capturing group ‘noun’. Matches one or more word characters.
- `(?<!ing)`: Negative lookbehind. Causes match failure if it matches ‘ing’ before the current point.
- `(?!$)`: Negative lookahead. Causes match failure if it matches the end of a line (since multiline mode is on).

Of the five article-noun pairs in the string, two don’t match:

- ‘*a hat*’ doesn’t match because the ‘t’ in ‘*hat*’ is the end of the line. The negative lookahead `(?!$)` matches this and causes failure.
- ‘*The knitting*’ doesn’t match because ‘*knitting*’ ends with ‘ing’. The negative lookbehind `(?<!ing)` matches this and causes failure.

Notice that the overall match values (‘*cat*’, ‘*mat*’, ‘*floor*’) don’t include the articles. This is because the tokens that match the article are inside a lookbehind, which is zero-width. However, the capturing group ‘noun’ could still capture these as the engine processed the lookbehind. The ability to capture text without consuming it, coupled with lookaround size variability, provides a lot of room for creative regexes.

Example 23: Thoughts

The `(?!$)` lookahead contains only zero-width tokens, so `(?<!$)` would be functionally equivalent. What would happen if you changed the other lookarounds to their directional opposites? What about if you removed the atomic group?

Try these changes in PowerShell. It reveals a lot about how the regex engine is processing the lookarounds.

Lookarounds Reference

You can view a complete reference for lookarounds on the Grouping Constructs[13] page at Microsoft Docs.

13.6.3 Conditional Logic

This feature is something that the regex parts of this book haven’t covered at all so far. That isn’t to say it’s not useful. Conditional matching constructs introduce *if-then-else* logic into regexes. The *if* part can be a subexpression, and the engine treats this as a zero-width assertion, much like a lookaround. It can also be a numeric index or group name and, in this case, the engine checks if the group has any captures. You can then tailor your pattern’s response by using different subexpressions for the *then* and *else* parts. The standard syntax for expression-based conditional matching is `(?(subexpression)yes|no)`.

Example 24: Conditional logic with subexpressions

```
$WithId = '<div id="some-id" class="styleA styleB">'
$WithoutId = '<div class="styleA styleB">'

$MyRegex = [regex]::new('(?mi)^<div(?([^>]+id="[^"]+")(?:\s*(?<name>[^"]+)=' +
    '"(?<value>[^"]+)")*\s*|(?<attribs>[^>]+))>$')

$ProccessMatches = {
    param($Match)
    Write-Host '  Attributes:'
    $Names = $Match.Groups['name'].Captures
    $Values = $Match.Groups['value'].Captures
    for ($i = 0; $i -lt $Names.Count; $i++) {
        Write-Host (
            '    {0} = {1}' -f $Names[$i].Value,
                ($Values[$i].Value -split ' ' -join ', ')
        )
```

[13] https://learn.microsoft.com/en-us/dotnet/standard/base-types/grouping-constructs-in-regular-expressions#zero-width-positive-lookahead-assertions

```
    }
    Write-Host (
        '  Raw attributes: [' + $Match.Groups['attribs'].Value + ']'
    )
}

Write-Host 'With Id:'
& $ProccessMatches -Match $MyRegex.Match($WithId)

Write-Host 'Without Id:'
& $ProccessMatches -Match $MyRegex.Match($WithoutId)
```

```
With Id:
  Attributes:
    id = some-id
    class = styleA, styleB
  Raw attributes: []

Without Id:
  Attributes:
  Raw attributes: [ class="styleA styleB"]
```

Try deconstructing the pattern from Example 24 yourself. See if you can interpret what it's doing before reading the explanation ahead.

The pattern in Example 24 captures text from HTML tags differently, depending on whether an *id* attribute is present. If so, the pattern extracts each attribute's name and value. Otherwise, it collects the attribute span in one lump.

Deconstructing the pattern reveals the distinct *if*, *then*, and *else* parts of the conditional.

```
(?mix)
^<div            # Literal "<div" after start of line
(?(              # Conditional statement (if)
    [^>]+            # One or more characters that aren't ">"
    id="             # Literal 'id="'
    [^"]+            # One or more characters that aren't '"'
    "                # Literal '"'
)                # Subexpression if condition matches (then)
    (?:              # Noncapturing group
        \s*              # Zero or more space characters
        (?<name>         # Capturing group "name"
            [^"]+            # One or more characters that aren't '"'
        )                # Match group once
        ="               # Literal '="'
        (?<value>        # Capturing group "value"
            [^"]+            # One or more characters that aren't '"'
        )                # Match group once
        "                # Literal '"'
    )*               # Match group zero or more times
    \s*              # Zero or more space characters
|                # Subexpression if condition fails (else)
    (?<attribs>      # Capturing group "attribs"
        [^>]+            # One or more characters that aren't ">"
    )                # Match group once
)                # End of conditional construct
>$               # Literal ">" before end of line
```

The engine treats the *if* part of the conditional as zero-width. The *yes* (*then*) part therefore needs to be capable of matching the same text, just as with a positive lookahead `(?=...)`.

As mentioned at the beginning of this section, you can also use the same conditional construct to evaluate the success of an earlier named or unnamed capturing group. The standard syntax for capture-based conditional matching is `(?(NNN)yes|no)` where `NNN` is a group index, or `(?(name)yes|no)` where `name` is a group name.

Example 25: Matching GUIDs with capture-based conditional logic

```
$MyPattern = '^(\{)?[a-f0-9]{8}-[a-f0-9]{4}-[0-4][a-f0-9]{3}-' +
    '[ab89][a-f0-9]{3}-[a-f0-9]{12}(?(1)\})$'

'96d30676-14d2-411c-b3f7-78f1708221e2' -imatch $MyPattern
'{96d30676-14d2-411c-b3f7-78f1708221e2' -imatch $MyPattern
'96d30676-14d2-411c-b3f7-78f1708221e2}' -imatch $MyPattern
'{96d30676-14d2-411c-b3f7-78f1708221e2}' -imatch $MyPattern
```

```
True
False
False
True
```

The pattern in Example 25 matches a GUID (UUID) with or without curly braces `{}`. If it finds an opening brace, a closing one must be present or the match fails. The example also shows that the *no* (*else*) group is optional. You can omit the pipe `|` and only include a *yes* (*then*) group. It's possible to leave the *yes* group blank, too. This is true for both expression and capture-based conditional constructs.

Example 25: Thoughts

What would happen if the `?` quantifier was **inside** the first capturing group? Try it out in PowerShell.

The capturing group matches, but with an empty capture. The conditional still treats this as *true*, so is always expecting a closing brace ‘}’, regardless of whether an opening brace ‘{‘ is present.

Conditionals Reference

You can view a complete reference for conditionals on the Alternation Constructs[14] page at Microsoft Docs.

[14] https://learn.microsoft.com/en-us/dotnet/standard/base-types/alternation-constructs-in-regular-expressions#conditional-matching-with-an-expression

13.6.4 Balancing Groups

The chapter has already mentioned the absence of recursive matching in .NET regex. So how do you match nested constructs? The answer is a feature found only in a handful of regex implementations: balancing groups. Instead of building a recursion stack, this construct *pops* captures from the capture stack of an existing capturing group. If the input string contains balanced nested constructs, there should be no remaining captures for the existing group, and you can test for this with conditionals.

The standard syntax for a balancing group is `(?<Closing-Opening>...)`, where an existing capturing group `(?<Opening>...)` is present. If the contents of the balancing group `Closing-Opening` match, the engine *pops* (removes) the last capture of the group `Opening`. It then stores all the text between that capture and the balancing group in a new capture of the `Closing` group.

Example 26: Capturing intermediate text with balancing groups

```
$MyPattern = '(?<Open>\()[^\(\)]+(?<Close-Open>\))'
$MyString = 'Some of this sentence is (enclosed in parentheses).'

$Result = [regex]::Match($MyString, $MyPattern)

if ($Result.Success) {
    $Result.Groups.ForEach{
        Write-Host ('Group {0}:' -f $_.Name)
        $_.Captures.ForEach{
            Write-Host ('  Capture: "{0}"' -f $_.Value)
        }
    }
}
```

```
Group 0:
  Capture: "(enclosed in parentheses)"
Group Open:
Group Close:
  Capture: "enclosed in parentheses"
```

Notice that the opening parenthesis, ‘(‘, is absent for the `Open` group capture. The engine *pushed* this capture to the stack when the `Open` group matched, but *popped* it when the `Close-Open` balancing group matched. You can see that the engine captured the text between the `Open` capture and `Close-Open` construct, and placed this in a capture of the `Close` group. Notice that the closing parenthesis, ‘)’, is absent from any captures, too. The engine doesn’t capture the contents of a balancing group construct, but **does consume them**.

You can omit the closing group name `(?<-Opening>...)` to prevent the capture of the intermediate text. You can also use an empty balancing group construct `(?<Closing-Opening>)` to guarantee the removal of the latest `Opening` capture.

Example 27: Nested balanced constructs with balancing groups

```
$MyPattern = '^[^"]*(?:(?:(?<StartQuote>(?=[\p{P} ]|^)")[^"]*)+' +
    '(?:(?<Quotes-StartQuote>"(?=[\p{P} ]|$))[^"]*)+)*(?(StartQuote)(?!))$'

$BalancedQuotes = '"Hello?", I queried. The stranger replied, ' +
    '"Why is "Hello" a question?".'

$UnbalancedQuotes = 'Hello?", I queried. The stranger replied, ' +
    '"Why is "Hello" a question?".'

$ResultBalanced = [regex]::Match($BalancedQuotes, $MyPattern)
$ResultUnbalanced = [regex]::Match($UnbalancedQuotes, $MyPattern)

Write-Host 'Balanced match:' $ResultBalanced.Success
Write-Host 'Unbalanced match:' $ResultUnbalanced.Success

Write-Host (
    'Balanced StartQuote capture count: ' +
    $ResultBalanced.Groups['StartQuote'].Captures.Count
)

Write-Host 'Quoted phrases from balanced string:'
$ResultBalanced.Groups['Quotes'].Captures | Format-Table
```

```
Balanced match: True
Unbalanced match: False

Balanced StartQuote capture count: 0

Quoted phrases from balanced string:

Index Length Value                       ValueSpan
----- ------ -----                       ---------
    1      6 Hello?
   52      5 Hello
   44     26 Why is "Hello" a question?
```

The balancing group `Quotes` captures nested quoted phrases from the balanced string. The empty `StartQuote` group is the indicator; the pattern uses a conditional `(?(StartQuote)(?!))` to assert this. The `(?!)` part is simply an empty negative lookahead, which always fails, and this acts as a breakpoint if `StartQuote` still has captures. Therefore, when an uneven number of quotes breaks the balance, `StartQuote` has leftover captures and the conditional causes match failure.

To understand what's happening in Example 27, the next example displays what each group matches.

Example 28: Breaking down a pattern with a balancing group

```
$MyPattern = @'
(?x)^                          # Ignore white space. Match start of string
[^"]*                          # Text before first quotation mark (none)
(?<QuoteSpans>                 # Rest of string
    (?<StartQuotePlusContents>     # '"' and level contents BEFORE deeper level
        (?<StartQuote>                 # Group stack used to count levels
            (?=[\p{P} ]|^)"                # '"' preceded by punctuation/space
        )                              #
        [^"]*                      # Level contents BEFORE deeper level
    )+                         # One or more, to ENTER multiple levels
    (?<EndQuoteAndPostfix>       # '"' and level contents AFTER deeper level
        (?<Quotes-StartQuote>      # Captures deeper level contents
            "(?=[\p{P} ]|$)            # '"' followed by punctuation/space/end
        )                          #
        [^"]*                      # Level contents AFTER deeper level
    )+                         # One or more, to EXIT multiple levels
)*                         # Zero or more, for multi-instances of same level
(?(StartQuote)(?!))        # If "StartQuote" has leftover captures, fail match
$                          # Match end of string
'@
```

```
$BalancedQuotes = '"Hello?", I queried. The stranger replied, ' +
    '"Why is "Hello" a question?".'

$Result = [regex]::Match($BalancedQuotes, $MyPattern)

if ($Result.Success) {
    $Result.Groups.ForEach{
        Write-Host ('Group "{0}":' -f $_.Name)
        $captureCount = -1
        $_.Captures.ForEach{
            Write-Host (
                '  Capture {0} (pos {1}) = {2}' -f
                ++$captureCount, $_.Index, $_.Value
            )
        }
    }
} else { Write-Host 'Match failed' }
```

```
Group "0":
  Capture 0 (pos 0) = "Hello?", I queried. The stranger replied, "Why is
  "Hello" a question?".
Group "QuoteSpans":
  Capture 0 (pos 0) = "Hello?", I queried. The stranger replied,
  Capture 1 (pos 43) = "Why is "Hello" a question?".
Group "StartQuotePlusContents":
  Capture 0 (pos 0) = "Hello?
  Capture 1 (pos 43) = "Why is
  Capture 2 (pos 51) = "Hello
Group "StartQuote":
Group "EndQuoteAndPostfix":
  Capture 0 (pos 7) = ", I queried. The stranger replied,
  Capture 1 (pos 57) = " a question?
  Capture 2 (pos 70) = ".
Group "Quotes":
  Capture 0 (pos 1) = Hello?
  Capture 1 (pos 52) = Hello
  Capture 2 (pos 44) = Why is "Hello" a question?
```

Navigating through the input string, the quotation level changes:

```
"Hello?", I queried. The stranger replied, "Why is "Hello" a question?".
^      ^                                   ^       ^     ^            ^
1      2                                   3       4     5            6
```

1. Open level 1 quote
2. Close level 1 quote
3. Open level 1 quote
4. Open level 2 quote (quote within a quote)
5. Close level 2 quote
6. Close level 1 quote

This means that the pattern has to enter and exit several deeper levels recursively, before exiting the current level. In the list, this occurs when “Open” or “Close” happen consecutively (3 → 4 and 5 → 6). The one-or-more `+` quantifier on the two quote groups (`StartQuotePlusContents` and `EndQuoteAndPostfix`) supports this, as many *opening* captures can occur successively before the balancing groups. The balancing groups can pop these captures to *close* those quotes successively, too.

The pattern also has to handle entering and exiting many instances of the same quote level. In the list, this occurs when “Open” follows a “Close” of the same level (2 → 3). The zero-or-more `*` quantifier on the `QuoteSpans` group supports this, as many capture push-pop cycles can occur for the inner groups. There are two instances of the first quotation level, which is why the `QuoteSpans` group has two captures. Each quote span contains quotation marks, quote contents, and the text that follows until the next quote.

The `StartQuotePlusContents` group reveals the text that `StartQuote` captures before `Quotes-StartQuote` removes it. Captures `0` and `1` are the level 1 quotes, while Capture `2` is the level 2 quote, both with starting quotation marks. The `EndQuoteAndPostfix` group shows what the `Quotes-StartQuote` group would have captured if it wasn’t a balancing group, along with the text following the quotes.

The pattern in Example 27 behaves exactly the same as Example 28, but uses noncapturing groups in place of `StartQuotePlusContents`, `EndQuoteAndPostfix`, `QuoteSpans`.

Balancing Groups Reference

You can view a complete reference for balancing groups on the Grouping Constructs[15] page at Microsoft Docs.

[15] https://learn.microsoft.com/en-us/dotnet/standard/base-types/grouping-constructs-in-regular-expressions#balancing-group-definitions

14. Regex Best Practices

That's it! You've now covered all the major topics on regexes in PowerShell and .NET. This chapter rounds off the regex part of the book with some important concepts that'll aid you in crafting effective regexes.

14.1 Constrained and Unconstrained Input

A topic that's never far from the spotlight in the realm of security is user input. This risk also applies to regexes. When tackling a pattern-matching problem, always ask yourself where the input is coming from. If you're generating your pattern dynamically, also consider the source of the information that plugs into your generator.

Constrained input is data from a reliable source that follows an expected format. This could be output from a known application or data from a database where you've already checked the contents for validity. *Unconstrained* input, on the other hand, is data that may differ from what you expect. This includes user input, but also data where a user could affect its format, such as log files that directly record user input.

If you can't be reasonably sure that the input text is constrained, construct your pattern as if it isn't. This is an aspect of defensive coding and you should always apply it to your regexes, too. It's much easier to craft your regex patterns defensively from the start rather than adapt an existing pattern to handle unexpected input.

14.2 Backtracking and Exponential Operations

The earlier regex chapters cover backtracking and its consequences extensively, but it's worth repeating a couple of points. Each time you use a quantifier that permits an indefinite number of matches (such as `+`, `?`, and `*` and their lazy counterparts), you create branching opportunities for the regex engine. The engine must save every starting point where branching can occur as these quantifiers consume input text, so that it can backtrack to one of these states if it needs to.

There are many situations where backtracking isn't beneficial or could degrade performance for nonmatching input. One example is catastrophic backtracking, a phenomenon the Regex Deep Dive chapter covers. However, saving each of these branching states has a performance penalty, and this becomes more obvious with large inputs. Therefore, wherever preventing backtracking won't affect your pattern's ability to match input text, you should prevent it. You can achieve this with atomic groups or lookarounds.

14.3 Preventing ReDoS with Regex Time-Outs

Following on from the discussion about unconstrained input and backtracking, it's possible to imagine an intentional input that could exploit these weaknesses. The .NET regex constructor and static methods support a *matchTimeout* parameter, which aborts the match process and throws an exception if the operation takes longer than the time-out period. This becomes important in production environments where unconstrained input coupled with vulnerable patterns could become a regex denial-of-service (ReDoS).

All .NET and .NET Framework versions since 4.5 support regex time-outs.

You can see a regex time-out in action in Example 3 from the Regex Deep Dive chapter. Here, it prevents catastrophic backtracking from continuing unchecked. The native PowerShell regex operators don't support a match time-out, so use this feature of the .NET methods in scenarios with unsupervised commands or in production environments.

14.4 Capturing Just Enough

Returning to the topic of regex performance, captures are another useful but costly feature to consider. Since each capturing group requires the engine to build a stack of captures, this can become both computationally expensive and memory-intensive for large inputs. Remember, a regex match always creates at least one capture—the entire match. Only use capturing groups if you need additional substrings from the match span.

Example 1: The performance penalty of excessive captures

```
$LongString = ('This is a single sentence. ' * 1e5).Trim()

Write-Host ('String length: {0:n0} chars' -f $LongString.Length)

# Matches multiple sentences
# WARNING: This pattern backtracks catastrophically with invalid input
$OneCapture = [regex]::Matches($LongString,
    '(?m)\b(?:([\w"''\(\)/-]+)[;,]?\s*)+[.?!]+')
$TwoCaptures = [regex]::Matches($LongString,
    '(?m)\b(([\w"''\(\)/-]+)[;,]?\s*)+[.?!]+')

Write-Host (
    'One capture: {0} ms' -f
    (Measure-Command { $OneCapture.Count }).TotalMilliseconds
)

Write-Host (
    'Two captures: {0} ms' -f
    (Measure-Command { $TwoCaptures.Count }).TotalMilliseconds
)
```

```
String length: 2,699,999 chars
One capture: 120.4441 ms
Two captures: 244.4458 ms
```

Both patterns in the example capture one sentence in each match and each word of the sentence as extra captures of that match. The second pattern generates excessive captures, however. It captures both the individual words and those words followed by any spaces. This means that each match contains two copies of almost identical text.

It's therefore important to use non-capturing `(?:...)` groups wherever you don't need to extract text, such as grouping constructs for repetition. You can also use the `ExplicitCaptures` regex option or inline option `(?n)` to disable capturing for all unnamed groups.

14.5 Static vs. Instance Methods and Caching

The Accessing Regexes chapter covered both the static and instance methods of the .NET `Regex` class. Caching has received only a brief mention and only in the context of interpreted versus compiled patterns.

By default, the regex engine caches the last fifteen patterns you use with the static methods.[1] If you use the `Compiled` option with a static call, the compiled CIL bytecode gets cached instead. You can set or retrieve how many patterns the engine stores with the `[Regex]::CacheSize` static property. This improves the efficiency of looping statements that contain static regex method calls. When you instantiate a `Regex` object, the engine bypasses this cache and creates a newly interpreted or compiled regex pattern.

Consider where this occurs in the execution of your code to avoid degraded performance. You can use static methods anywhere, with or without the `Compiled` flag. This takes advantage of static caching and removes the need to consider the cost of instantiation. However, it could be less efficient if you use lots of unique patterns in one session.

With class instances, there is never a need for the engine to reinterpret or recompile the processed pattern. It will be available for as long as it remains in scope. Aim for a single instantiation at the outermost scope that's appropriate. Some scenarios and ideal locations are:

- **PowerShell modules**: In the module scope, outside of any functions
- **PowerShell classes**: As a static property
- **Script functions**: In the `begin {}` block

Of course, if your pattern is going to change, you might need to create a new object in each method or function call. In these circumstances, apply the same rule and minimize the number of instances of the **same pattern**.

14.6 No More CompileToAssembly()

In older Windows-only editions of PowerShell and .NET, the `Regex` class provided a static method for compiling several regular expressions and exporting them to an assembly. This method now throws an exception, and you should instead use the `Compiled` regex option.

[1] Microsoft. (2021, Sep. 15). *Compilation and Reuse in Regular Expressions - The Regular Expressions Cache*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/standard/base-types/compilation-and-reuse-in-regular-expressions#the-regular-expressions-cache. [Accessed: Mar. 01, 2022].

14.7 Getting the Scope Right

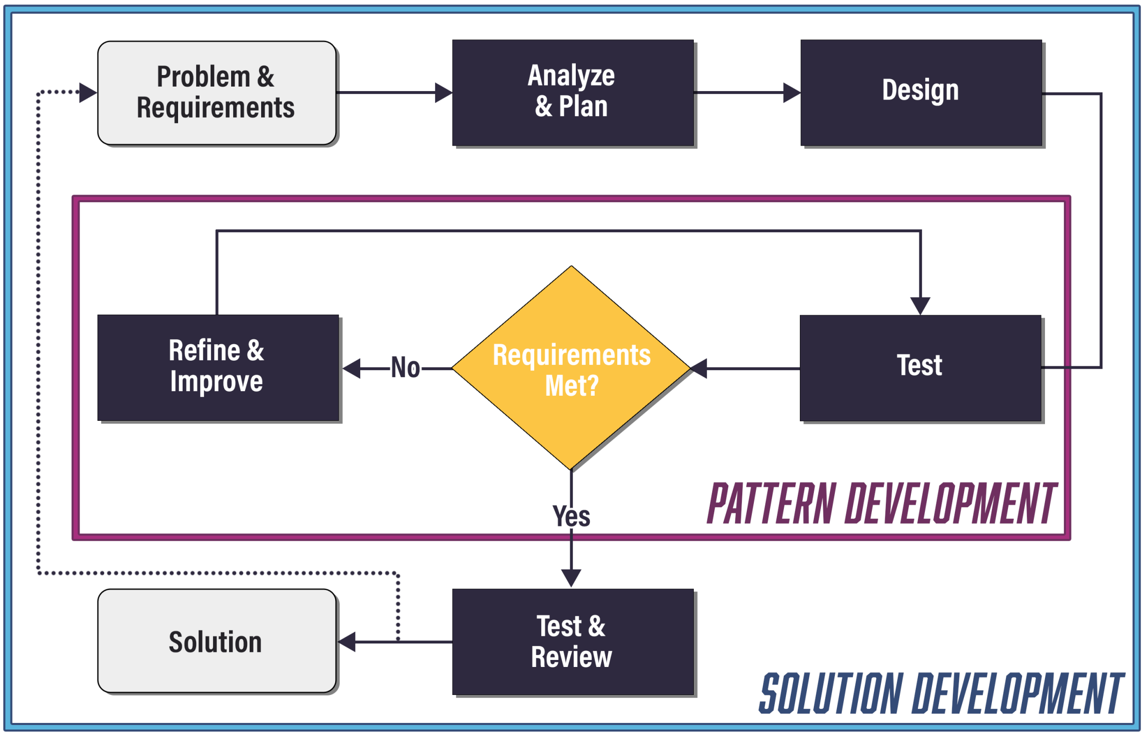
Regexes are powerful—but so is PowerShell! It can be too easy to get tunnel vision when tackling a problem and to try to solve it using regex alone. Always assess the different approaches you could take. Ask yourself questions such as:

- Should you process the text line-by-line, or as a whole? Either can be more or less efficient than the other in different contexts. Looping statements with a pattern for single lines are often simpler but don't work for target matches that span many lines.
- Could you achieve the same result with several simpler patterns and PowerShell statements? Simpler patterns are much easier to debug. Complex patterns can be more efficient than switching between programmatic and regex logic several times.
- Should there be some preprocessing? Filtering and formatting the input text (with more regexes or otherwise) before passing it on to the extraction pattern can simplify the problem. This is especially important with unconstrained input.
- What's faster to create? What's faster to run? Some solutions might be quick to put together, but at the cost of performance. Consider how efficient your pattern-matching needs to be, and balance it against how much time you have to deliver a solution.

Before you craft a regex pattern to solve a problem, prepare a strategy. This could involve testing pattern fragments with various inputs, or looking at the consistency of sample data. It may involve evaluating the performance of one approach versus another, such as line-by-line and whole-text matching. An idea for the solution will often emerge during this planning step, and it's simply a matter of putting all the parts together and refining them.

14.8 Iterative Development

Seldom is a first-attempt regex pattern ready for deployment, even with a prior strategy. There's often a handful of overlooked cases that'll cause unwanted behavior. This means you'll likely need to improve and retest your pattern several times before it's fit for purpose. How far you go to refine your pattern depends on the requirements for the solution and how quickly you must deliver it. Iterative development is an effective approach for creating your patterns and the tools you build around them.



Iterative development applied to regex patterns and pattern-matching tools

This section considers the pattern from Example 1 and discusses potential refinements. The requirement is that the pattern captures individual sentences and isolates each word within a sentence, without accompanying punctuation. After overcoming the catastrophic backtracking with an atomic group, the pattern looks like this:

```
(?m)\b(?>([\w"'\(\)/-]+)[;,]?\s*)+[.?!]+
```

It's now more well-behaved, but there's still needless computation when at least one sentence doesn't end with a valid terminator (`.?!:`). This is because the engine backtracks and tries to find a match starting at each subsequent word boundary. For the string 'This has no end', it would try the following.

```
(This) (has) (no) (end)
(has) (no) (end)
(no) (end)
(end)
```

In each instance, the match fails because of the lack of sentence termination, but the engine keeps trying shorter matches until it runs out of whole words. This *isn't* catastrophic backtracking, but affects the pattern's efficiency.

There are a variety of approaches that could improve the efficiency of this pattern. You can remove the need for a word boundary altogether by using a more advanced beginning-of-sentence assertion. The positive lookbehind `(?<=^\s*|[.?!]\s+)` asserts that before the current position, there must be either:

- A sentence terminator and at least one space
- The start of the string/line and zero or more spaces

This change improves performance by excluding some match candidates before the engine processes the rest of the pattern, effectively providing a short-circuit. Since .NET supports variable-length lookbehinds, you can even match a variable number of spaces before the sentence. Integrating this into the pattern gives:

```
(?m)(?<=^\s*|[.?!]\s+)(?>([\w"'\(\)/-]+)[;,]?\s*)+[.?!]+
```

Example 2: Improving efficiency with a short-circuit assertion

```powershell
$BadSentence = 'lots of words ' * 100
$OldPattern = '(?m)\b(?>([\w"''\(\)/-]+)[;,]?\s*)+[.?!]+'
$NewPattern = '(?m)(?<=^\s*|[.?!]\s+)(?>([\w"''\(\)/-]+)[;,]?\s*)+[.?!]+'

Write-Host (
    'Old pattern: {0} ms' -f (
        Measure-Command { [regex]::Matches($BadSentence, $OldPattern) }
    ).TotalMilliseconds
)

Write-Host (
    'New pattern: {0} ms' -f (
        Measure-Command { [regex]::Matches($BadSentence, $NewPattern) }
    ).TotalMilliseconds
)
```

```
Old pattern: 5.7201 ms
New pattern: 0.4184 ms
```

The pattern still matches no sentences that contain unexpected characters. Using a negative custom character class solves this problem. The class `[^.?!;,\s]` matches anything **except** spaces (`\s`), sentence terminators (`.?!`), and sentence dividers (`:,`). Assuming this is the desired behavior, the pattern now looks like this:

```
(?m)(?<=^\s*|[.?!]\s+)(?>([^.?!;,\s]+)[;,]?\s*)+[.?!]+
```

Excluding only a few characters instead of an inclusive (allowlist) approach introduces an additional problem. The word group now captures word-adjacent characters, such as quotation marks and brackets. The pattern would capture ‘*(Hello world)*’ as ‘*(Hello*’ and ‘*world)*’. This doesn’t fulfill the requirements, which specify no punctuation with the captured words. You could extend the pattern further to account for characters that may appear before or after words and exclude them from captures.

Since the pattern is now quite long, the following snippets use the `IgnorePatternWhitespace` option, with patterns spread over several lines.

```
(?mx)
(?<=^\s*|[.?!]\s+)          # 1. Beginning-of-sentence assertion
(?>                         # Word matching group, no backtracking
    [\('"/:.]*                  # 2. Any characters before a word
    ([^.?!;,\s\(\)"'/]+)        # 3. Word characters (capture)
    [;,\)'"/]*                  # 3. Any characters after a word
    \s*                         # 4. Zero or more spaces
)+                          # Match one or more words
[.?!]+                      # 5. One or more sentence terminators
```

The pattern now handles characters that appear before words (part ***2***), and after words (part ***4***). Part ***3*** in the pattern is the capture, which now excludes these extra characters. This is necessary to prevent greedy matching from consuming characters that part ***4*** would capture, or part ***2*** from giving up characters to part ***3*** in any backtracking scenarios. Another benefit is that the pattern now matches decimal points within numbers (such as *6.47* and *.39*).

Is the pattern fit for purpose now? The only way to discover this is to run tests.

Example 3: Testing a more advanced pattern for words in sentences

```
$MyString = "The customer's name is Jane E. Doe."
$MyPattern = '(?mx)(?<=^\s*|[.?!]\s+)(?>[\(''"/:.]*([^.?!;,\s\(\)"''/]+)' +
    '[;,\)''"/]*\s*)+[.?!]+'

[regex]::Matches($MyString, $MyPattern).ForEach{
    Write-Host ('Sentence: "{0}"' -f $_.Value)
    $_.Groups[1].Captures.ForEach{
        Write-Host ('  Word: "{0}"' -f $_.Value)
    }
}
```

```
Sentence: "The customer's name is Jane E."
  Word: "The"
  Word: "customer"
  Word: "s"
  Word: "name"
  Word: "is"
  Word: "Jane"
  Word: "E"
Sentence: "Doe."
  Word: "Doe"
```

Immediately, two further problems are apparent: The pattern interprets the apostrophe `'` in "*customer's*" as a boundary between two words. It also interprets the period `.` in '*Jane E. Doe*' as a sentence terminator.

Tackling the apostrophe issue first, one solution is to remove the exclusion from the main word group and use a negative lookbehind to ensure an apostrophe doesn't appear at the end of a word. While it would be possible to use conditional logic and balancing groups to differentiate single quotation marks from apostrophes, it's important to weigh pattern complexity and efficiency against covering *every* edge case. The pattern now looks like this:

```
(?mx)
(?<=^\s*|[.?!]\s+)          # 1. Beginning-of-sentence assertion
(?>                         # Word matching group, no backtracking
    [\('"/:.]*                  # 2. Any characters before a word
    ([^.?!;,\s\(\)"/]+)         # 3. Word characters (capture)
    (?<!')                      # 6. Must not end with apostrophe
    [;,\)'"/]*                  # 3. Any characters after a word
    \s*                         # 4. Zero or more spaces
)+                          # Match one or more words
[.?!]+                      # 5. One or more sentence terminators
```

Part ***6*** in the snippet is the new negative lookbehind, and part ***3*** no longer excludes apostrophes.

The second issue is more complex than it may seem. Accounting for a period as part of name initials is relatively easy. You could add an earlier alternative inside the word group `[A-Z]\.|...` to cover those scenarios.

```
(?mx)
(?<=^\s*|[.?!]\s+)          # 1. Beginning-of-sentence assertion
(?>                         # Word matching group, no backtracking
    [\('"/:.]*                  # 2. Any characters before a word
    ([A-Z]\.|[^.?!;,\s\(\)"/]+) # 3. Word characters (capture)
    (?<!')                      # 6. Must not end with apostrophe
    [;,\)'"/]*                  # 3. Any characters after a word
    \s*                         # 4. Zero or more spaces
)+                          # Match one or more words
[.?!]+                      # 5. One or more sentence terminators
```

Part ***3*** now accounts for single-letter name initials. However, this introduces a new edge case. If a sentence ends with a single uppercase letter and a period, the pattern will continue to match the next sentence.

Consider the sentences "*Smith takes Vitamin D. Jones doesn't take anything.*". How would you tell between two sentences and one sentence containing the name '*Vitamin D. Jones*'? To a human, it's obvious that *Vitamin* is unlikely to be a name, but without a contextual dictionary of names and nouns, or the ability to parse sentence structure, it isn't possible for a computer to differentiate between them.

This is an edge case, but names with initials are more common than sentences that end with single uppercase letters. Therefore, the modification is justifiable in this instance. These decisions and considerations appear often when developing regexes. Deciding what to account for and what's beyond the scope of a reasonable solution is an important aspect of regex iterative development.

There are several further considerations that could drive pattern evolution. Should the pattern:

- Capture hyphenated compound words (such as *cross-platform*) independently? The existing pattern treats them as single words.

- Handle clause separators, such as em dashes (—), as word separators? The existing pattern treats these as one hyphenated word if there aren't spaces around the dash.
- Exclude more opening/closing punctuation around words? The existing pattern includes punctuation, such as curly braces `{}`, with word captures. Unicode categories such as opening punctuation `\p{Ps}` and closing punctuation `\p{Pe}` exist for this.
- Treat numbers with decimal points as single words? The existing pattern treats them as two.
- Handle abbreviations such as '*Mrs.*' and '*Prof.*'? The existing pattern breaks a sentence here.
- Account for a colon followed by a newline? The existing pattern treats the text following as a continuing sentence.
- Handle punctuation from other languages with Unicode categories? The existing pattern aims to match English sentences only.

These are just a handful of modifications that could be necessary depending on the scenario. The refinements made in this section aren't the only possible ones either. The examples have demonstrated a single development pathway. How you tackle a pattern-matching problem will be unique to each scenario.

Example 4: A pattern that meets the requirements but doesn't cover all edge cases

```
$Sentences = @'
This sentence contains "quotes", (brackets), and the number 42.01.
Mr. E. Smith's [first] name is John-E is for example.
PowerShell is cross-platform.
This sentence isn''t valid
'@

$MyPattern = '(?m)(?<=^\s*|[.?!]\s+)(?>[\(''"/:.]*([A-Z]\.|' +
    '[^.?!;,\s\(\)"/]+)(?<!'')[;,\)''"/]*\s*)+[.?!]+'

[regex]::Matches($Sentences, $MyPattern).ForEach{
    Write-Host ('Sentence: "{0}"' -f $_.Value)
    Write-Host (
        '   Words: {0}{1}' -f ($_.Groups[1].Captures.Value -join '/'),
        [Environment]::NewLine
    )
}
```

```
Sentence: "This sentence contains "quotes", (brackets), and the number 42.01."
   Words: This/sentence/contains/quotes/brackets/and/the/number/42/01

Sentence: "Mr."
   Words: Mr

Sentence: "E. Smith's [first] name is John-E is for example."
   Words: E./Smith's/[first]/name/is/John-E/is/for/example

Sentence: "PowerShell is cross-platform."
   Words: PowerShell/is/cross-platform
```

The take-away is that it's important to test your patterns in a variety of contexts. You may find that one development path isn't working, and you may have to return to the analysis and planning stage. That's OK—humans can backtrack too, just like regex engines!

14.9 Edge Cases and Near Matches

Adversarial testing is an often-overlooked idea with regexes. When you're developing and testing a pattern, you're primarily thinking about what it *should* match. It's equally important to consider what it *could* match, though. There are three kinds of input that your pattern will need to handle:

- Input that matches your pattern
- Input that **almost** matches your pattern
- Input that doesn't match your pattern

It's the second case that often causes headaches. As well as testing valid and invalid inputs to your pattern-matching solution, test for near-matches. There are two kinds of near-matches that cause unexpected and unwanted behavior:

- Input that **should** match your pattern, but **doesn't**. This is a **false negative**.
- Input that **shouldn't** match your pattern, but **does**. This is a **false positive**.

Consider the pattern for matching IP addresses in Example 33 from Accessing Regexes. If the zero from `[01]?` was missing, most valid IP addresses would still match. Likewise, if no anchors or lookarounds were present, most invalid IPs would still not match. However, false positives and negatives arise without them.

Example 5: Near matches to a regex pattern

```
$MyPattern =
    '(?:(?<Octets>25[0-5]|2[0-4][0-9]|1?[0-9]{1,2})\.){3}' +
    '(?<Octets>25[0-5]|2[0-4][0-9]|1?[0-9]{1,2})'

$Tests = [ordered]@{
    'Valid 1'   = '198.51.100.255'  # Valid
    'Valid 2'   = '198.051.100.255' # Valid
    'Invalid 1' = '198.51.256.255'  # Block 3 invalid >255
    'Invalid 2' = '198.51.100.256'  # Block 4 invalid >255
}

$Tests.Keys.ForEach{
    $Result = [regex]::Match($Tests[$_], $MyPattern)
    $Match = $Result.Success ? $Result.Value : '-'
    [pscustomobject]@{
        Test  = $_
        Input = $Tests[$_]
        Match = $Match
    }
}
```

```
Valid 1    Input: 198.51.100.255   Match: 198.51.100.255
Valid 2    Input: 198.051.100.255  Match: -
Invalid 1  Input: 198.51.256.255   Match: -
Invalid 2  Input: 198.51.100.256   Match: 198.51.100.25
```

Line 14 in Example 5 uses a ternary operator instead of an if-then-else statement. This feature is available starting with PowerShell 7. See the Advanced Conditions chapter for more on this.

Valid 2 is a false negative because the pattern wasn't equipped to handle the zero in '*051*'. ***Invalid 2*** is a false positive because the pattern can still match without consuming the final '*6*'. Anchors such as `^` and `$` overcome the false positive, while using `[01]?` instead of `1?` overcomes the false negative.

Example 5 demonstrates the importance of testing your regex patterns and pattern-matching solutions. Throw as many inputs at them as you can. Ask yourself, "How can I break this?." Your pattern should be able to handle any input you could **reasonably expect**.

Best Practices Reference

You can view some regex Best Practices[2] at Microsoft Docs.

14.10 Thread Safety

This section focuses only on thread safety in the context of regex objects. In general, you won't need to worry about this aspect of regex objects, but scenarios such as accessing regex results across PowerShell runspaces will make this necessary.

A `[Regex]` class instance is both immutable and thread-safe. This means you're able to create and use regex objects across many threads. Single result objects such as `[Match]`, `[Group]`, and `[Capture]` **aren't** automatically thread-safe, however. Neither are their associated collections, `[MatchCollection]`, `[GroupCollection]`, and `[CaptureCollection]`. This is because some of these classes use *lazy evaluation* to improve performance.

Where possible, aim to access regex result objects within a single thread.

If you must share singular result objects across threads, use the static `Synchronized()` method to retrieve a thread-safe instance of the object. This method causes the engine to compute each capture of every group, resulting in an immutable object that you can use between threads.[3]

[2] https://learn.microsoft.com/en-us/dotnet/standard/base-types/best-practices

[3] Microsoft. (2020, Jul. 08). *System.Text.RegularExpressions - Group.cs.* L42-L57. dotnet/runtime on GitHub. [Online]. Available: https://github.com/dotnet/runtime/blob/main/src/libraries/System.Text.RegularExpressions/src/System/Text/RegularExpressions/Group.cs. [Accessed: Jan. 30, 2022].

Example 6: Using synchronized regex results

```
$MyString = 'F09FA694'
$HexValues = [regex]::Match($MyString, '(?i)([a-f0-9]{2})+')
$SyncMatch = [System.Text.RegularExpressions.Match]::Synchronized($HexValues)

$ApiUrl = 'https://ucdapi.org/unicode/latest/codepoint/hex/{0}'
[ref]$Counter = 0

$SyncMatch.Groups[1].Captures | ForEach-Object -Parallel {
    $Index = [System.Threading.Interlocked]::Increment($using:Counter)

    Write-Verbose "[Thread $Index]: Processing match '$_'" -Verbose

    $Data = Invoke-Restmethod -Uri ($using:ApiUrl -f $_.Value)

    '[Thread {0} Result] 0x{1} ({2}) = {3}' -f
        $Index, $_.Value.ToUpper(), [Convert]::ToInt32($_.Value, 16),
        ($Data.name ? $Data.name : $Data.name1)
} -ThrottleLimit 3 -Verbose
```

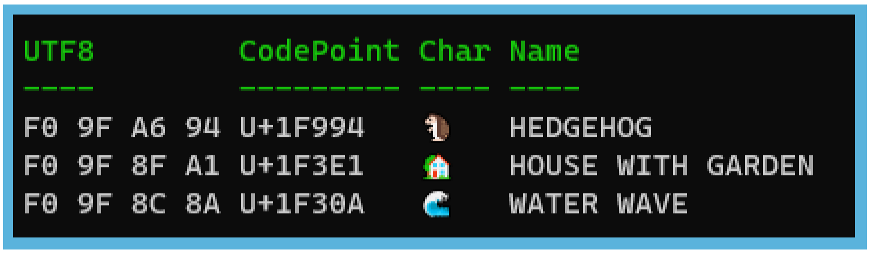
```
VERBOSE: [Thread 2]: Processing match '9F'
VERBOSE: [Thread 1]: Processing match 'F0'
VERBOSE: [Thread 3]: Processing match 'A6'
VERBOSE: [Thread 4]: Processing match '94'
[Thread 1 Result] 0xF0 (240) = LATIN SMALL LETTER ETH
[Thread 3 Result] 0xA6 (166) = BROKEN BAR
[Thread 2 Result] 0x9F (159) = APPLICATION PROGRAM COMMAND
[Thread 4 Result] 0x94 (148) = CANCEL CHARACTER
```

Note how the order of execution doesn't match the order of the captures. `ForEach-Object -Parallel` uses PowerShell runspaces in parallel and the order in which PowerShell processes the input collection isn't guaranteed. A thread-safe increment of the counter shows the order in which processing starts for each capture. Because of the variability in web API response times, the order in which processing completes varies from both the input and the starting order. The order of the results differs from the order of the processed matches.

If you must enumerate result collections across threads, such as `[MatchCollection]`, `[GroupCollection]`, and `[CaptureCollection]`, use the `SyncRoot` property of the instance to synchronize access by locking the object during access. Since PowerShell doesn't have a `lock` statement, use `[System.Threading.Monitor]` to ensure synchronized access to the collection.

Example 7: Synchronizing access to regex result collections

```
$MyString = '0xF0 0x9F 0x8C 0x8A 0xF0 0x9F 0xA6 0x94 0xF0 0x9F 0x8F 0xA1'
$MyPattern = '(?i)(0xF[0-4])(?: (0x[89a-f][0-9a-f])){3}'
$Utf8FourByte = [regex]::Matches($MyString, $MyPattern)
$ApiUrl = 'https://ucdapi.org/unicode/latest/chars/{0}'

$Utf8FourByte | ForEach-Object -Parallel {
    $LockTaken = $false
    try {
        [System.Threading.Monitor]::Enter($_.Groups.SyncRoot, [ref]$LockTaken)
        $LeadingByte = $_.Groups[1].Captures[0].Value
        $ContinuationBytes = $_.Groups[2].Captures.ForEach{ $_.Value }
        $UTF8 = $_.Groups[0].Value -ireplace '0x([0-9a-f]{2})', '$1'
    }
    catch { Write-Error -ErrorRecord $_; return }
    finally {
        if ($LockTaken) {
            [System.Threading.Monitor]::Exit($_.Groups.SyncRoot)
        }
    }

    $Bytes = [byte[]]::new(4)
    $Bytes[0] = $LeadingByte -as [byte]
    for ($i = 0; $i -lt $ContinuationBytes.Count; $i++) {
        $Bytes[$i + 1] = $ContinuationBytes[$i] -as [byte]
    }

    $Char = [System.Text.Encoding]::UTF8.GetString($Bytes)
    $Data = Invoke-Restmethod -Uri ($using:ApiUrl -f $Char)

    [pscustomobject]@{
        UTF8      = $UTF8
        CodePoint = 'U+' +
            [System.Convert]::ToString($Data.codePoint, 16).ToUpper()
        Char      = $Char
        Name      = $Data.name ? $Data.name : $Data.name1
    }
}
```



```
UTF8        CodePoint Char Name
----        --------- ---- ----
F0 9F A6 94 U+1F994   🦔   HEDGEHOG
F0 9F 8F A1 U+1F3E1   🏡   HOUSE WITH GARDEN
F0 9F 8C 8A U+1F30A   🌊   WATER WAVE
```

Output from Example 7

Once again, the order in which processing takes place isn't guaranteed.

`[System.Threading.Monitor]::Enter(...)` attempts to get an exclusive lock of an object. If another thread has already locked the object, `Enter()` waits until it's released. `Exit()` releases a locked object locked with `Enter()`.

Regex Thread Safety Reference

You can view a complete reference for regex Thread Safety[4] at Microsoft Docs.

14.11 Next Steps

You've now reached the end of the regex part of this book! All that's left is to suggest some helpful resources for continued learning.

[4] https://learn.microsoft.com/en-us/dotnet/standard/base-types/thread-safety-in-regular-expressions

First, take a look at the Modern IT Automation with PowerShell Extras[5] repository on GitHub. There are some more complex regex patterns there with real-world applications, complete with breakdowns and explanations. You'll find these in the Edition-01/Regex[6] folder.

If you'd like a more in-depth journey into the world of regexes, look no further than *Mastering Regular Expressions*[7] by Jeffrey Friedl. While the latest edition of this book is from 2006, it's a truly comprehensive course. It also contains language-specific chapters, including for .NET. Since .NET's regex implementation hasn't changed in recent years, the content still applies to current versions of .NET and PowerShell at the time of writing.

On the other hand, if you're looking for a more direct guide on creating regexes to solve specific problems, *Regular Expressions Cookbook*[8] by Jan Goyvaerts and Steven Levithan is a good place to start. This book takes you through common pattern-matching problems and introduces solutions in the context of popular programming environments. .NET is amongst these, so the presented solutions need no modifications to work in PowerShell.

Another great resource with plug-and-play solutions for .NET is *Regular Expression Pocket Reference*[9] by Tony Stubblebine. This handbook contains both at-a-glance syntax reference and regex solutions.

For a more academic take on regexes and regular expression theory, try Rex[10]. Rex is a tool written by Microsoft Research's RiSE Group[11] that efficiently generates matching inputs for one or more .NET regex patterns using symbolic finite automata (SFA). It has resulted in several publications[12], but is also useful for evaluating your own patterns. It provides valuable insight into how the engine is interpreting your pattern. The original online version is no longer available, but you can still download the original Rex binary[13] from the Microsoft website.

For more recent developments, Rex is a part of the Automata .NET Library[14] on GitHub, alongside lots of other tools related to finite state automata (FSA) and transducers (FST). While there, take a look at the Symbolic Regex Matcher[15] library. This is an efficient alternative approach to interpreting regular expressions using SFA.

Don't forget to check out the further reading section below, which includes many of the resources discussed in the regex part of this book, plus more.

14.12 Further Reading

14.12.1 Official Reference Materials

- **Microsoft .NET Regex Reference**[16]

[5] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras
[6] https://github.com/devops-collective-inc/Modern-IT-Automation-with-PowerShellExtras/tree/main/Edition-01/Regex
[7] https://www.oreilly.com/library/view/mastering-regular-expressions/0596528124/
[8] https://www.oreilly.com/library/view/regular-expressions-cookbook/9781449327453/
[9] https://www.oreilly.com/library/view/regular-expression-pocket/9780596514273/
[10] https://www.microsoft.com/en-us/research/project/rex-regular-expression-exploration/
[11] https://www.microsoft.com/en-us/research/group/research-software-engineering-rise/
[12] https://www.microsoft.com/en-us/research/project/rex-regular-expression-exploration/publications/
[13] https://www.microsoft.com/en-us/download/details.aspx?id=52296
[14] https://github.com/AutomataDotNet/Automata
[15] https://github.com/AutomataDotNet/srm
[16] https://learn.microsoft.com/en-us/dotnet/standard/base-types/regular-expressions

- **Microsoft PowerShell Regex Reference**[17]
- **Microsoft PowerShell Comparison Operators**[18]
- **Microsoft PowerShell `-split` Operator**[19]
- **Microsoft PowerShell `switch` Statement**[20]
- **Microsoft PowerShell `Select-String` Cmdlet**[21]
- **Microsoft .NET Regex Behavior**[22]
- **Microsoft .NET Regex Backtracking**[23]
- **Microsoft .NET Regex Object Model**[24]
- **Microsoft .NET Regex Compilation and Reuse**[25]
- **Microsoft .NET Regex Thread Safety**[26]
- **Microsoft .NET Regex Best Practices**[27]

14.12.2 Other Materials

- **Mastering Regular Expressions**—The comprehensive regex book[28]
- **Regular Expressions Cookbook**—Comprehensive regex solutions for common pattern-matching problems[29]
- **Regular Expression Pocket Reference**—A regex handbook with syntax and solutions for quick reference[30]
- **regular-expressions.info**—General regex reference[31]
- **Regex101**—General regex tester and debugger (no .NET support)[32]
- **RegExr**—Open source general regex tester (no .NET support)[33]
- **rextester.com/Tester**—.NET regex tester[34]
  - You can also visit the rextester Homepage[35] to test-compile C#
- **RegexPlanet**—Regex tester including .NET[36]
  - Use `(?'...')` instead of `(?<...>)` for named captures
- **regexstorm.NET**—.NET regex tester[37]

[17] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_regular_expressions
[18] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_comparison_operators#-match-and--notmatch
[19] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_split
[20] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_switch
[21] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/select-string
[22] https://learn.microsoft.com/en-us/dotnet/standard/base-types/details-of-regular-expression-behavior
[23] https://learn.microsoft.com/en-us/dotnet/standard/base-types/backtracking-in-regular-expressions
[24] https://learn.microsoft.com/en-us/dotnet/standard/base-types/the-regular-expression-object-model
[25] https://learn.microsoft.com/en-us/dotnet/standard/base-types/compilation-and-reuse-in-regular-expressions
[26] https://learn.microsoft.com/en-us/dotnet/standard/base-types/thread-safety-in-regular-expressions
[27] https://learn.microsoft.com/en-us/dotnet/standard/base-types/best-practices
[28] https://www.oreilly.com/library/view/mastering-regular-expressions/0596528124/
[29] https://www.oreilly.com/library/view/regular-expressions-cookbook/9781449327453/
[30] https://www.oreilly.com/library/view/regular-expression-pocket/9780596514273/
[31] https://www.regular-expressions.info/dotnet.html
[32] https://regex101.com
[33] https://regexr.com/
[34] https://rextester.com/Tester
[35] https://rextester.com/
[36] https://www.regexplanet.com/advanced/dotnet/index.html
[37] http://regexstorm.net/tester

  - HTTP-only website

- **Optimizing Regex Performance I**—2010 blog entry from the Microsoft Base Class Library team[38]
- **Optimizing Regex Performance II**—2010 blog entry from the Microsoft Base Class Library team[39]
- **Rex Project**—RiSE Group Rex landing page[40]
- **Rex Introduction Video**—Margus Veanus from RiSE explains SFA and Rex (archive)[41]

[38] https://learn.microsoft.com/en-us/archive/blogs/bclteam/optimizing-regular-expression-performance-part-i-working-with-the-regex-class-and-regex-objects-ron-petrusha

[39] https://learn.microsoft.com/en-us/archive/blogs/bclteam/optimizing-regular-expression-performance-part-ii-taking-charge-of-backtracking-ron-petrusha

[40] https://www.microsoft.com/en-us/research/project/rex-regular-expression-exploration/

[41] https://web.archive.org/web/20210411024653/https://channel9.msdn.com/Blogs/Peli/Margus-Veanes-Rex-Symbolic-Regular-Expression-Exploration/

V PowerShell Security

> *"Sticking your head in the sand might make you feel safer, but it's not going to protect you from the coming storm."* — Barack Obama

PowerShell security has always been an afterthought in organizations, putting users and administrators at risk. Because of the scope of its complexity, malware often uses PowerShell in its payload or as a launchpad. PowerShell has four security pillars comprising different aspects of itself. They are:

1. **Script Development**: Educating the development of scripts using best security practices.
2. **Script Execution**: To reduce unauthorized script execution, enforce policies on Script Execution.
3. **Console Execution**: To reduce single-liner attacks, implement policies on Console Execution.
4. **PowerShell Remoting**: To reduce lateral attacks, implement policies on PowerShell remoting sessions.

All tiers must be understood, implemented, and configured correctly to minimize the risk. The topics in this section target these tiers with:

- Script Signing.
- Script Execution Policies.
- PowerShell Constrained Language Mode.
- PowerShell Just Enough Administration (JEA).

15. Script Signing

PowerShell allows you to protect your scripts from tampering by signing them with a digital signature.[1] Signing a script with a digital signature requires that you have a code signing certificate. This chapter discusses why you should sign your scripts and what options are available to you. You'll learn about working with digital signatures and code signing certificates, as well as how to implement a script signing solution in your organization using a Public Key Infrastructure (PKI).

15.1 What Is Script Signing and How It Protects You

Digital signatures, like signatures on paper, are a way to ensure authenticity. Digital signing uses a cryptographic process to ensure, with high probability, that signed data:

1. Hasn't been modified after being signed
2. Originates from an identified source

This doesn't necessarily mean the source is trustworthy or that the signed data is of high quality! It gives you some confidence that the data hasn't changed since it was signed, and that the creator and signer are the same entity.

Almost all the code you run on a Windows machine is signed. Running signed code significantly reduces the probability of executing malicious code. Most Windows software vendors sign their compiled binaries before passing it on to you, the user. Microsoft signs all of the compiled binaries which are a part of Windows. Doing this establishes trust that the underlying binaries have not changed between publication and installation. The digital signature also allows administrators to configure security software such that users can only run approved software releases.

Script signing allows you to add these protections to any script from any source. It also allows you to have a higher degree of trust in scripts provided to you.

However, code signing doesn't guarantee complete protection against malware. Here are two examples:

- Microsoft signed a malicious driver[2] in June 2021.
- Attackers sometimes sign malware[3] with stolen certificates.

You should include a code signing policy as part of your organization's security strategy, but you can't rely on it alone.

[1] Microsoft. (2022, Mar. 18), *about Signing*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_signing. [Accessed: Sep. 15, 2022].

[2] https://www.gdatasoftware.com/blog/microsoft-signed-a-malicious-netfilter-rootkit

[3] https://www.welivesecurity.com/2018/07/09/certificates-stolen-taiwanese-tech-companies-plead-malware-campaign/

15.1.1 How Digital Signing Works

To sign a file, you need data you want to sign and a cryptographic key pair (in common words: a certificate). A cryptographic key pair is a pair of large numbers which were calculated in such a way that:

1. You can encrypt data with one of them and then use the other to decrypt this data. This is *asymmetric* or *public-key* cryptography.[4]
2. It isn't possible to calculate the value of one from the other, even with access to encrypted data.

Store one of these numbers in a safe place, like a password—this is your *private key*. The other—distribute freely to those who must be able to decrypt data encrypted with the first one—this is your *public* key. You can use a private key to encrypt data and your recipients can use the corresponding public key to decrypt it. Usually, this key pair comes in the form of a digital certificate, which allows you to add more information, such as who issued it, what it's for, and the intended user.

However, the signed data isn't *encrypted* because any recipient can read it clearly. What **is** encrypted is the *hash sum* of that data. "What's a hash sum?" you might ask.

A hash sum is a cryptographically calculated string, fixed in length and alphanumeric, which is supposed to be unique for any unique piece of data. An important property of any hash sum is that you can't derive the original data from it; having only a hash sum, you **can't** reconstruct the original data.

PowerShell uses the SHA-256 algorithm to calculate hash sums for script signing, but there are other algorithms available[5].[6]

When you send a digitally signed message to someone, you calculate a hash sum for the content, encrypt the hash sum with your private key and send both the message and the encrypted hash sum. The recipient calculates a hash sum for the message again, decrypts the received hash sum using **your** public key, and compares them. If both hash sums are the same, it means the message remained unchanged during transmission. It also means that it's **you** who sent it, because no one else has access to your private key (or at least no one should have access to it).

15.1.2 How Code Signing Works in Modern Windows Systems

In Windows, all digital signature trusts are based on certificates.[7] A digital certificate is a public key + some additional information, like:

[4] Wikipedia. (2022, Sep. 10). *Public-key cryptography.* Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Public-key_cryptography. [Accessed: Sep. 15, 2022].

[5] https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptography.hashalgorithmname#properties

[6] Microsoft. (2022, Aug. 23) *Get-FileHash.* Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/get-filehash. [Accessed: Sep. 15, 2022].

[7] Microsoft. (2022, Jun. 02) *Cryptography and Certificate Management.* Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/windows/security/cryptography-certificate-mgmt. [Accessed: Sep. 17, 2022].

- Who created (*issued*) this certificate (Issuer);
- Dates of validity (Valid from, Valid to);
- Purposes for which this certificate is valid (Key Usage, Enhanced Key Usage);
- Who the intended user is (Subject).

A digital certificate can be self-signed or signed by another certificate (usually the certificate's parent, in the case of a certificate authority hierarchy). In case of a self-signed certificate, the person (or a machine) which created it, certifies by itself that it's the signer that made this certificate. In self-signed certificates, `Issuer` and `Subject` fields are the same.

When a certificate is signed by another entity, we say that the other entity issued the certificate. In that case, you'll see who issued the certificate in the `Issuer` field of the certificate.

Usually, certificates are issued by *Certification Authorities*, which are special services *trusted* by people and organizations worldwide. We trust their policies and processes to issue certificates only for legitimate purposes and only to subjects whose identity is established.

To trust a Certification Authority (CA) in Windows, you take its *root certificate* and place it into a designated certificate store—Trusted Root Certification Authorities. A CA's root certificate is a certificate self-signed by that root CA. A CA uses this certificate to sign all other certificates issued by it.

In the case of a single-tier CA in Windows, all certificates issued by the root CA will be valid. However, a best-practice implementation requires that there be at least two tiers of CAs in a Windows environment. In that case, the root CA will issue one certificate to a subordinate CA. The subordinate CA will issue all other certificates.

Windows has many root certificates already installed in the Trusted Root Certification Authorities store, out of the box. However, you're free to remove CAs you don't trust from that store or add other CAs (or self-signed certificates) there.

Microsoft supports a list of all Certification Authorities[8] which are trusted by Windows by default.

As soon as the Windows OS trusts a CA's root certificate, all other certificates issued by this and subordinate CAs will be trusted automatically. However, there are conditions. Today's date must be within the certificate's validity period (between "Valid from" and "Valid to" dates), and the certificate itself must not be *revoked*.

A revoked certificate is a certificate whose serial number (a unique identifier) was added into a *Certificate Revocation List* (CRL). A CRL is just a list of certificates' serial numbers along with the date/time when each was added to the list. This list is periodically published by a CA. The CA adds a link to that list to every certificate it issues (CRL Distribution Point). When a client checks a certificate's validity, it uses the `CRL Distribution Points` attribute of the certificate to download the CRL and then searches for the certificate's serial number on that list. If nothing is found, you're ready!

[8] https://learn.microsoft.com/en-us/security/trusted-root/participants-list

Windows performs these checks every time it sees a certificate. That’s not all that’s checked for signed code, however. Code signing certificates must also have a “Code Signing” purpose in their Enhanced Key Usage attribute.

15.2 The Anatomy of a Signed Script

Here’s an example of a signed PowerShell script:

Example 1: A signed script contains a signature block

```
Write-Host "Signed Script"
# SIG # Begin signature block
# MIIIPwYJKoZIhvcNAQcCoIIIMDCCCC0CAQEfCzAJBgUrDgMCGgUAMGkGCisGAQQB
# gjcCAQSgWzBZMDQGCisGAQQBgjcCAO,wJgIDAQAABBAfzDtgWUsITrck0sYpfvNR
# AgEAAgEAAgEAAgEAAgEAMCEwCQYFKD1DAhoFAAQUHVVG73a+TMJVHFMfMllHWt0T
# o+igggZTMIIC4TCCAkOgAwIBAgITae1AAAKkII0Kbir2EQAAAAAAAjAKBggqhkjO
# PQQDBDAVMRMwEQYDVQQDEwpNeSBSbS50IENBMB4fDTIfMDgyOTIfMTE0OVofDTI2
# MDgyOTIfMjE0OVowWjETMBEGCgmSJo,T8ifkARkWA25ldDEfMBUGCgmSJomT8ifk
# ARkWB2V4YW1wbGUfEjAQBgoJkiaJkL0sZAEZFgJhZDEWMBQGA1UEAfMNTfkgSfNz
# dWluZyBDQTB2MBAGByqGSM49AgEGBV,BBAAiA2IABIK0kBR1YTFZUoaDW9i9IMIG
# o3G2g14oqK9nGZ6ZW9uRW83HFMos8E,liB6eaA+kgbi4VEYgG0E9A2zYe7fzSlBN
# ViFFPLBPlhMiYLn9YNZknraKkGmkpI2pz12cswvOCKOCAQ4wggEKMBAGCSsGAQQB
# gjcVAQQDAgEAMB0GA1UdDgQWBBQQBLnf9jwBGLe8KrJnheSaLJKhVDAZBgkrBgEE
# AYI3FAIEDB4KAFMAdQBiAEMAQTALBUNVHQ8EBAMCAYYwDwYDVR0TAQH/BAUwAwEB
# /zAfBgNVHSMEGDAWgBRf9WiwId5L3Vdd2EHIFUw1gm53PDA4BgNVHR8EMTAvMC2g
# K6AphidodHRwOi8vcGtpLmV4YW1wbPUuY29tL0NEUC9NeVJvb3RDQS5jcmwwQwYI
# KwYBBQUHAQEENzA1MDMGCCsGAQUFBWAChidodHRwOi8vcGtpLmV4YW1wbGUuY29t
# L0FJQS9NeVJvb3RDQS5jcnQwCgYIKHZIzj0EAwQDgYsAMIGHAkIBgHL2AfA9vAFu
# 8/cUZ/s4JbP8SvIm3GzotkiMYb68G49sCJWo+ZRfcQjRyGrOlAP5gFu7BbjoAtTy
# OgzbnFwfOAwCQSfIDPWwi8n93qBLXE4/+WjEEbBFNY4OBnskcEF7DMKJPr3HnaKU
# /8Bp1lnfchc2/ccHRBHNff/lDGeyMVLvHjr5MIIDajCCAvCgAwIBAgITOgAAAAUl
# cN1zaYSZ9wAAAAAABTAKBggqhkjOaAQDAzBaMRMwEQYKCZImiZPyLGQBGRYDbmV0
# MRcwFQYKCZImiZPyLGQBGRYHZfhhR,BsZTESMBAGCgmSJomT8ifkARkWAmFkMRYw
# FAYDVQQDEw1NeSBJc3N1aW5nIENBk44fDTIfMDkyNzE5MzUyMVofDTIyMDkyNzE5
# MzUyMVowajETMBEGCgmSJomT8ifkS,kWA25ldDEfMBUGCgmSJomT8ifkARkWB2V4
# YW1wbGUfEjAQBgoJkiaJk/IsZAEZT1JhZDEOMAwGA1UEAfMFVfNlcnMfFjAUBgNV
# BAMTDUFkbWluafN0cmF0b3IwWTATh3cqhkjOPQIBBggqhkjOPQMBBwNCAAQzTyMe
# AKfiFVSY0ynrrEFQQY4M+EzSTDhTE,dDT+SWqNa2iS/QouR7Yw4H0I9b/uKFodbh
# 8Jo9Jv1MbgSga9Y5o4IBgzCCAf8wS2YJKwYBBAGCNfUHBDAwLgYmKwYBBAGCNfUI
# g5SvIoTbuBCElY04h63cIoOYmE5aP,iLEpr8vfkCAWQCAQIwEwYDVR0lBAwwCgYI
# KwYBBQUHAwMwDgYDVR0PAQH/BAQD06eAMBsGCSsGAQQBgjcVCgQOMAwwCgYIKwYB
# BQUHAwMwHQYDVR0OBBYEFJHSpOs/t,3v4SmW/Tj5Ugcnhjj+MB8GA1UdIwQYMBaA
# FBAHefH2PAEYt7wqsmeF5JoskqFUF7sGA1UdHwQ0MDIwMKAuoCyGKmh0dHA6Ly9w
# a2kuZfhhbfBsZS5jb20vQ0RQL015r,NzdWluZ0NBLmNybDBGBggrBgEFBQcBAQQ6
# MDgwNgYIKwYBBQUHMAKGKmh0dHA6e49wa2kuZfhhbfBsZS5jb20vQUlBL015SfNz
# dWluZ0NBLmNydDA3BgNVHREEMDAue,wGCisGAQQBgjcUAgOgHgwcQWRtaW5pc3Ry
# YfRvckBhZC5leGFtcGflLm5ldDAKP0gqhkjOPQQDAwNoADBlAjB83uKYtBkduS94
# I9Ihv9Lwtkff3T27q7f5SJThW7blS,vwmFfbgEfHg8sjTsKWRpsCMQDrTW/EqyJb
# af4KMIzN3e31f86Bqp9T+WN9BWjTC64m8CA6KFTefNbTVEQFAfgIg6AfggFWMIIB
# UgIBATBfMFofEzARBgoJkiaJk/Iso,EZFgNuZfQfFzAVBgoJkiaJk/IsZAEZFgdl
# eGFtcGflMRIwEAYKCZImiZPyLGQBN1YCYWQfFjAUBgNVBAMTDU15IElzc3Vpbmcg
# Q0ECEzoAAAAFJfDdc2mEmfcAAAAAf1UwCQYFKw4DAhoFAKB4MBgGCisGAQQBgjcC
# AQwfCjAIoAKAAKECgAAwGQYJKoZIB7cNAQkDMQwGCisGAQQBgjcCAQQwHAYKKwYB
# BAGCNwIBCzEOMAwGCisGAQQBgjcCO,UwIwYJKoZIhvcNAQkEMRYEFGYfKlHDtCoj
# fISNIf3qAON9ff1TMAsGByqGSM49O1EFAARIMEYCIQC8Q9FdL/GyWCTyabocOrmr
# Y1BzEny+K7az9TL2WzaKJgIhAI3/K0KWepFOWrYKDsWNsfDJedlA3SbwSczCLLNN
# # koIi
# SIG # End signature block
```

In a signed PowerShell script, you have your code first and after it, at the end of the file, a signature block.

The signature block consists of:[9]

1. A beginning line: `# SIG # Begin signature block`.
2. The signature itself: multi-line, base64 encoded.
3. An ending line: `# SIG # End signature block`
4. A newline MUST occur after the ending line.

A script is considered to have been signed only when it has a full, unmodified signature block. Therefore, if you either remove the signature block completely or just tamper with it a bit, by removing or adding a symbol to any line, PowerShell will treat this file as an ordinary unsigned script.

No code is allowed after the signature block. If you put anything but comments at the end of a signed .ps1 file, you'll get this error:

```
Executable script code found in signature block.
    + CategoryInfo          : ParserError: (:) [], ParseException
    + FullyQualifiedErrorId : TokenAfterEndOfValidScriptText
```

This is actually a security feature: it prevents you from unknowingly running potentially malicious code. A malicious actor might insert a block that looks just like a signature, but isn't one. For example, they might remove the last letter "k" from the first line and it will look like this `# SIG # Begin signature bloc`. As mentioned already, because the signature is now malformed, PowerShell won't treat it as one and will consider the file unsigned. After that fake signature, the attacker may then insert more code and sign the resulting script with a real signature block at the actual end of the file. Since signature blocks are really long and it's the last element of a file, a person reviewing the script might not notice that there is more code after the fake signature.

Thankfully, PowerShell's got your back here. It searches for comments which look similar to the signature block start line and then, if it finds any code after them, raises that error. What's important is that the check for code after the signature block executes **before** the check that decides if the script is signed or not.

You can see the actual implementation of it in the PowerShell code[10]. If it wasn't for this check, the first signature, which is fake, would be ignored. The result would be that **all** code in the file would be executed as a signed script, including the malicious addition.

15.3 How to Sign a Script

In order to sign your scripts, you need a certificate with the "Code Signing" Enhanced Key Usage (OID 1.3.6.1.5.5.7.3.3). Then, with the power of `Set-AuthenticodeSignature`[11], you apply a

[9] Microsoft. (2021, Jul. 29) *Lexical Structure (PowerShell Language Specification 3.0).* Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/scripting/lang-spec/chapter-02. [Accessed: Sep. 17, 2022].

[10] https://github.com/PowerShell/PowerShell/blob/8d017d8a752cf960996c8c23c0f323a9cbe0f905/src/System.Management.Automation/engine/parser/tokenizer.cs#L1696

[11] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.security/set-authenticodesignature

digital signature to your script and et voilà! It's really that easy. The hardest part is to make sure your clients trust the certificate which you used to sign the scripts (this chapter covers this detail a little later).

15.3.1 Acquiring a Code Signing Certificate

You have several ways to get your own certificate for signing code. You can either generate it by yourself (self-signed), buy one from a publicly trusted provider, or issue one from your own internal Public Key Infrastructure (PKI).

15.3.1.1 Self-Signed

Generating a new self-signed certificate is actually pretty easy:

Example 2: Creating a new self-signed certificate

```
New-SelfSignedCertificate -Type CodeSigningCert -Subject MySelfCodeSigningCert
```

This will give you a new code signing certificate in the local computer Personal store. Note that your Thumbprint (and other attributes of the certificate) will be different than the value shown here.

Example 3: Displaying the new certificate

```
Get-ChildItem -Path 'Cert:\LocalMachine\My' |
    Where-Object -FilterScript {$_.Subject -eq 'CN=MySelfCodeSigningCert'}
```

```
   PSParentPath: Microsoft.PowerShell.Security\Certificate::LocalMachine\My

Thumbprint                                Subject
----------                                -------
F45E5297DA01A97E527F5AF262F29B4A8CCF2083  CN=MySelfCodeSigningCert
```

You can specify another store with the `-CertStoreLocation` parameter. For example, you can use "Cert:\CurrentUser\My" for your user's Personal store.

If you look at the certificate closely, you'll see that its `EnhancedKeyUsageList` property contains `Code Signing` as its value:

Example 4: The EnhancedKeyUsageList property shows that this is a code signing certificate

```
Get-Item -Path Cert:\LocalMachine\My\F45E5297DA01A97E527F5AF262F29B4A8CCF2083 |
    Select-Object -Property 'EnhancedKeyUsageList'
```

```
EnhancedKeyUsageList
--------------------
{Code Signing (1.3.6.1.5.5.7.3.3)}
```

To check if you can use the certificate for code signing, use `Get-ChildItem` with the `-CodeSigningCert` parameter:

Example 5: The -CodeSigningCert parameter is a useful feature of the Certificate provider

```
Get-ChildItem -Path 'Cert:\LocalMachine\My' -CodeSigningCert
```

```
   PSParentPath: Microsoft.PowerShell.Security\Certificate::LocalMachine\My

Thumbprint                                Subject
----------                                -------
F45E5297DA01A97E527F5AF262F29B4A8CCF2083  CN=MySelfCodeSigningCert
```

All returned certificates could be used to sign your scripts.

15.3.1.2 Self-Hosted PKI

Self-signed certificates are fine for testing, but have a major downside: no one trusts the certificates. You must install the certificates on every computer where they might be needed. Doing so is time-consuming and difficult to manage. The preferred method to issue certificates in an organization is to use a Public Key Infrastructure (PKI). A Public Key Infrastructure is a hierarchy of Certification Authorities, where you have usually one top-level CA (*root CA*) and several subordinates.

By trusting the root CA, you effectively trust every certificate issued by it. You'll deploy the certificate of that root CA to your machines (and the certificates of any intermediate CAs) and the computers in the organization will trust your code signing certificates (and other certificates issued by these CAs) automatically.

The Use Your Own PKI section below talks about how you can set up a PKI.

15.3.1.3 Third Party PKI

An alternative to hosting a PKI by yourself is to hire somebody to do that for you. You have two options here:

1. Order code signing certificates one by one from a commercial certificate provider, like Digicert, GlobalSign, Sectigo, etc. This is a relatively cheap, quick, and easy option, but you'll pay for every certificate you issue—this might not be the most cost-effective choice if you plan to issue a lot of certificates.
2. Sign-up for a managed PKI service. This type of service is a full-scale PKI, where you have all the flexibility, but don't have to worry about the management of this infrastructure. Many security companies offer PKI-as-a-Service: SecureW2, HydrantID, Entrust, just to name a few.

15.3.2 How to Install Code Signing Certificates Properly

For the whole signature process to work, a code signing certificate and its certificate chain must be properly installed in the system:

- On the signer's computer:
    1. The certificate itself must be in two certificate stores:
        1. Personal (either user or computer)—it's for you (the signer) to sign code using the certificate.
        2. Trusted Publishers—without having the certificate in this location, you won't be able to validate your signatures.
    2. The root certificate of the certificate chain must be in the Trusted Root Certification Authorities store. That's usually the topmost certificate you see in the "Certification Path" tab of a certificate.
- On the user's computer:
    1. The code signing certificate must be in Trusted Publishers.
    2. The root certificate must be in the Trusted Root Certification Authorities.
    3. Any intermediate certificates must be in the Intermediate Certification Authorities.

The PowerShell Certificate provider[12] has slightly different names for these stores:

GUI Name	Certificate Provider Name
Personal	My
Trusted Publishers	TrustedPublisher
Trusted Root Certification Authorities	Root

For the sake of demonstration, the following examples use a self-signed certificate, which means both `$RootCert` and `$SigningCert` are the same. In a real environment, you'll most likely have a certificate from a proper chain with a separate root certificate. Ensure that you put the correct objects in `$RootCert` and `$SigningCert` variables.

[12] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.security/about/about_certificate_provider

A self-signed certificate generated with the `New-SelfSignedCertificate` cmdlet will most likely already be installed in your Personal store, so it won't be covered in the following example.

First, you need to save your code signing certificate in a variable for easy access:

```
$Cert = Get-ChildItem -Path 'Cert:\LocalMachine\My' -CodeSigningCert
```

The above (and the examples below) assumes that there is only one code signing certificate in the Personal store. If there are more, then the `$Cert` object needs to be indexed in the examples below.

Next, install it as a Trusted Publisher certificate:

Example 6: Adding a certificate to the machine's trusted publisher store

```
$SigningCert = $Cert
$TrustPubStor = [System.Security.Cryptography.X509Certificates.X509Store]::new(
  [System.Security.Cryptography.X509Certificates.StoreName]::TrustedPublisher,
  [System.Security.Cryptography.X509Certificates.StoreLocation]::LocalMachine
)
$TrustPubStor.Open(
    [System.Security.Cryptography.X509Certificates.OpenFlags]::ReadWrite
)
$TrustPubStor.Add($SigningCert)
$TrustPubStor.Close()
```

Since it's a self-signed certificate, you must install it as a root certificate too:

Example 7: Adding a certificate to the machine's root certificate store

```
$RootCert = $Cert
$RootStore = [System.Security.Cryptography.X509Certificates.X509Store]::new(
    [System.Security.Cryptography.X509Certificates.StoreName]::Root,
    [System.Security.Cryptography.X509Certificates.StoreLocation]::LocalMachine
)
$RootStore.Open(
    [System.Security.Cryptography.X509Certificates.OpenFlags]::ReadWrite
)
$RootStore.Add($RootCert)
$RootStore.Close()
```

The above example doesn't use `Import-Certificate` because that cmdlet requires a certificate to be in a file. Using the .NET class instead allows skipping the step of exporting the certificate to a file on disk.

That's it! You're ready to begin signing your scripts.

15.3.3 Signing Process

Regardless of how you acquired a code signing certificate, the signing process is the same. Start by retrieving your code signing certificate from the store. Next, use the `Set-AuthenticodeSignature` cmdlet, passing the certificate and the path to the script that you want to sign:

Example 8: Signing a script with a code signing certificate from the user's personal store

```
$ScriptPath = Join-Path -Path $env:Temp -ChildPath 'test1.ps1'
Set-Content -Path $ScriptPath -Value 'Write-Host "Signed Script"'
$Cert = Get-ChildItem -Path 'Cert:\CurrentUser\My' -CodeSigningCert
Set-AuthenticodeSignature -Certificate $Cert -FilePath $ScriptPath
```

`Set-AuthenticodeSignature` supports files no smaller than 4 bytes. Thankfully, spaces and line breaks do the trick for ultra-small scripts.

Reminder: The above example assumes that only one code signing certificate is in the personal store.

15.3.4 How to Prevent Your Signatures from Expiring

When you have a code signing certificate, it has a finite period of validity. What will happen after that certificate expires? The code you signed with it effectively becomes unsigned again! How can you solve this problem you might ask? You could issue a new certificate and sign the code again, but that's time consuming. You would also need to deliver the new signed release to your users somehow.

Thankfully, there's an easier solution—a time stamp server. The purpose of time stamping is to certify that you signed some code while the code signing certificate was still valid. This will allow your code to continue to be signed even after the certificate's expiration date.

There are several publicly available time stamp servers:

- http://timestamp.digicert.com
- http://timestamp.sectigo.com
- http://timestamp.verisign.com/scripts/timstamp.dll
- https://www.freetsa.org/

Others exist—check out this gist[13] by @Manouchehri[14]!

If you want to run your own time stamp server, there are several commercial solutions available along with these open-source implementations:

[13] https://gist.github.com/Manouchehri/fd754e402d98430243455713efada710

[14] https://github.com/Manouchehri

- SignServer[15]
- uts-server[16]

Just choose one of the time stamp servers and use it in the `-TimestampServer` parameter:

Example 9: Including a time stamp when signing a script

```
$Params = @{
    Certificate = $Cert
    FilePath = $ScriptPath
    TimestampServer = 'http://timestamp.digicert.com'
}
Set-AuthenticodeSignature @Params
```

15.3.5 What Else Can You Sign

15.3.5.1 Functions

In PowerShell, you can't sign a function itself, because PowerShell only supports signing files. What you can do is to have one function per .ps1 file and therefore, when you sign that file, you effectively sign the function.

The content of your .ps1 files will look like this:

```
function Do-Stuff {
    'stuff done'
}
# SIG # Begin signature block
# ...
# SIG # End signature block
```

When dot-sourcing a file to import the function from it into your current session, you're still executing that script. When you execute a script, the system checks its signature and, if the signature check doesn't pass, the function won't import.

15.3.5.2 Modules

Signing a module is very easy: just sign every .ps1, .psm1, .psd1, and .ps1xml file using the `Set-AuthenticodeSignature` cmdlet.

Now, why sign all these files, when you could just sign the main .psm1? It's because you sign individual files to prevent tampering.

Suppose your module consists of several .ps1 files plus a .psd1, and a .psm1, which loads all the .ps1 files. In that case, a malicious actor could change those .ps1 files and the module would still load, so **you need to sign everything**.

[15] https://www.signserver.org/
[16] https://github.com/kakwa/uts-server

15.4 How to Verify a Signature

To make sure a PowerShell file is correctly signed, you can use several tools:

1. PowerShell (Get-AuthenticodeSignature)[17]
2. Sysinternals sigcheck[18]
3. signtool.exe[19]
4. And of course you can open file properties in Explorer and look at the Digital Signatures tab.

15.4.1 Get-AuthenticodeSignature

This is what a proper output from `Get-AuthenticodeSignature` looks like for a trusted certificate:

Example 10: Displaying a signature with Get-AuthenticodeSignature

```
Get-AuthenticodeSignature -FilePath 'C:\test1' | Select-Object -Property *
```

```
SignerCertificate      : [Subject]
                           CN=MySelfCodeSigningCert

                         [Issuer]
                           CN=MySelfCodeSigningCert

                         [Serial Number]
                           1B599557458628A54D3D9EA33D15C160

                         [Not Before]
                           13/06/2021 23:38:18

                         [Not After]
                           13/06/2022 23:58:18

                         [Thumbprint]
                           F45E5297DA01A97E527F5AF262F29B4A8CCF2083

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

And this is what you get when the root certificate of the certificate chain is missing from the Trusted Root Certification Authorities store:

[17] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.security/get-authenticodesignature?view=powershell-7.1

[18] https://learn.microsoft.com/en-us/sysinternals/downloads/sigcheck

[19] https://learn.microsoft.com/en-us/dotnet/framework/tools/signtool-exe

```
SignerCertificate      : [Subject]
                           CN=MySelfCodeSigningCert

                         [Issuer]
                           CN=MySelfCodeSigningCert

                         [Serial Number]
                           1B599557458628A54D3D9EA33D15C160

                         [Not Before]
                           13/06/2021 23:38:18

                         [Not After]
                           13/06/2022 23:58:18

                         [Thumbprint]
                           F45E5297DA01A97E527F5AF262F29B4A8CCF2083

TimeStamperCertificate :
Status                 : UnknownError
StatusMessage          : A certificate chain processed, but terminated in
                a root certificate which is not trusted by the trust provider.
Path                   : C:\test.ps1
SignatureType          : Authenticode
IsOSBinary             : False
```

15.4.2 Sigcheck

This is what a proper output from *sigcheck* looks like for a trusted certificate:

Example 11: Displaying a signature with sigcheck

```
sigcheck64.exe C:\test.ps1
```

```
Sigcheck v2.82 - File version and signature viewer
Copyright (C) 2004-2021 Mark Russinovich
Sysinternals - www.sysinternals.com

c:\test.ps1:
        Verified:       Signed
        Signing date:   00:03 14/06/2021
        Publisher:      MySelfCodeSigningCert
        Company:        n/a
        Description:    n/a
        Product:        n/a
        Prod version:   n/a
        File version:   n/a
        MachineType:    n/a
```

And this is what you get when the root certificate of the certificate chain is missing from the Trusted Root Certification Authorities store:

```
Sigcheck v2.82 - File version and signature viewer
Copyright (C) 2004-2021 Mark Russinovich
Sysinternals - www.sysinternals.com

c:\test.ps1:
        Verified:       A certificate chain processed, but terminated in a root
                        certificate which is not trusted by the trust provider.
        File date:      00:03 14/06/2021
        Publisher:      MySelfCodeSigningCert
        Company:        n/a
        Description:    n/a
        Product:        n/a
        Prod version:   n/a
        File version:   n/a
        MachineType:    n/a
```

15.4.3 Signtool

This is what a proper output from *signtool* looks like for a trusted certificate:

Example 12: Displaying a signature with signtool

```
$Params = @{
    Path = ${Env:ProgramFiles(x86)}
    ChildPath = 'Windows Kits\10\bin\10.0.22000.0\x64\signtool.exe'
}
$SignToolPath = Join-Path @Params
& $SignToolPath verify /pa C:\test.ps1
```

```
File: C:\test.ps1
Index  Algorithm  Timestamp
========================================
0      sha1       None

Successfully verified: C:\test.ps1
```

Below is what you'll see when the root certificate of the certificate chain is missing from the Trusted Root Certification Authorities store:

```
File: C:\test.ps1
Index  Algorithm  Timestamp
========================================
SignTool Error: A certificate chain processed, but terminated in a root
        certificate which is not trusted by the trust provider.

Number of errors: 1
```

15.4.4 Execution Errors

All these tools will be silent if you do have the certificate that you used to sign the code in the Trusted Publishers store on the machine. They will only ask questions if the certificate is not present. You'll see a prompt when you try to run that script:

Example 13: Attempting to run a script signed with an untrusted certificate

```
C:\test.ps1
```

```
Do you want to run software from this untrusted publisher?
File C:\test.ps1 is published by CN=MySelfCodeSigningCert and is not trusted on
                         your system. Only run scripts from trusted publishers.
[V] Never run  [D] Do not run  [R] Run once  [A] Always run  [?] Help (default
                                                                    is "D"):
```

And of course, if your system doesn't have the proper root certificate installed, you'll receive an error about that too:

Example 14: Attempting to run a signed script where the root of the certificate chain is untrusted

```
C:\test.ps1
```

```
.\test.ps1 : File C:\test.ps1 cannot be loaded. A certificate chain processed,
but terminated in a root certificate which is not trusted by the trust provider
At line:1 char:1
+ .\test.ps1
+ ~~~~~~~~~~~~~~
    + CategoryInfo          : SecurityError: (:) [], PSSecurityException
    + FullyQualifiedErrorId : UnauthorizedAccess
```

15.5 Scaling Out

In this section, you'll learn what to do when you need to run signed scripts on more than one machine. If you want to issue signing certificates by yourself, first read Use Your Own PKI, which explains how to build your own Public Key Infrastructure. Otherwise, you can skip right to Deploy Code Signing Certificates in a Corporate Environment.

15.5.1 Use Your Own PKI

15.5.1.1 Why Would You Want Your Own PKI?

You might have different reasons to want your own Public Key Infrastructure. Also, your company may already have its own PKI. You should ask the proper parties about this. Here are some examples as to why you may desire your own PKI:

- **Compliance**. Some regulations might require you to have all sensitive/security information on your premises.
- **Security**. You might not trust how a service provider will manage your PKI. Then, of course, you must do it by yourself.

- **Availability**. With your own servers, you can achieve any availability level, which isn't always an option with managed services.
- **Flexibility**. Only you define which certificates, how, and to whom you'll issue them. SaaS (Software as a Service) isn't always that flexible.
- **Cost**. In general, doing things by yourself is cheaper. However, note that this is not a universal rule. Sometimes the opposite is true, because supporting your own infrastructure requires good engineers and hardware—these cost money.

15.5.1.2 How to Build a PKI with Just PowerShell

In this example, you'll build a *two-tiered* Public Key Infrastructure:

- *Public* doesn't mean it will be available to anybody—it just means that this system uses *public-key cryptography.*
- *Two-tiered* means that the PKI hierarchy will have two tiers: a root certification authority and an issuing certification authority—this is sufficient for most organizations.

You'll need at least two servers with Windows Server 2019 or later, which have no other software installed on the server. One of them must be a member of an AD DS (Active Directory Domain Services) domain (the issuing CA), the other should be just a workgroup machine (the root CA). You will also need a web server: you'll put CA certificates and CRLs there (this can even be a non-Windows machine). This is often the issuing CA. Using the issuing CA as a web server is simpler than using a separate server as a PKI web server. Also, you must be an Enterprise Administrator in this AD DS forest to install the proper Windows roles and serveices.

It often makes sense for the same group of people to manage both AD DS and ADCS (Active Directory Certificate Services—the PKI) infrastructures: both of them are usually company-wide and both issue cryptographic assertions about identities. Basically, you want to protect your PKI servers as carefully as you protect domain controllers. All this also applies to AD FS (Active Directory Federation Services) as well.

15.5.1.2.1 Root CA

The first step in building your own PKI is to configure a root certification authority. For best practices, this **must** be a VM which can be normally offline (powered off) or a physical machine that can be turned off.[20] Even though the VM or physical hardware is normally powered-off, this does NOT reduce the need that the instance (physical or virtual) must be properly backed up. This VM or physical hardware must not be a member of the domain/forest.

[20] TechNet wiki contributors. (2016, Aug. 12). *Active Directory Certificate Services (AD CS) Public Key Infrastructure (PKI) Design Guide.* Microsoft TechNet Wiki. [Online]. Available: https://social.technet.microsoft.com/wiki/contents/articles/7421.active-directory-certificate-services-ad-cs-public-key-infrastructure-pki-design-guide.aspx#Use_Offline_CAs. [Accessed: Sep. 16, 2022].

For the best protection of the root CA's private key, the industry standard is to store the private key on a smart card or a USB HSM (Hardware Security Module); but of course, that also has its own downsides. If that smart card stops working, or if you lose the PIN code for it, you'll have to reinstall your PKI from scratch (not immediately, but eventually, because you won't be able to renew subordinate CAs certificates anymore or issue new CRLs ([Certificate Revocation Lists]). The benefit of having a private key on a smart card is that it's very difficult to extract the key from there. But for the next base case, you can store the backup of the root CA's private key on another medium, with proper security.

The first step in ADCS CA installation is to install the Windows feature containing all of the required bits:

Example 15: Installing the Active Directory Certificate Services CA feature on the root CA

```
Install-WindowsFeature -Name ADCS-Cert-Authority -IncludeManagementTools
```

```
Success Restart Needed Exit Code      Feature Result
------- -------------- ---------      --------------
True    No             Success        {Active Directory Certificate Services...
```

Then you need to perform an initial configuration of the certificate authority:

Example 16: Configuring the root Certificate Authority

```
$Params = @{
  CACommonName = 'My Root CA'
  ValidityPeriod = 'Years'
  ValidityPeriodUnits = 10
  CryptoProviderName = 'ECDSA_P521#Microsoft Software Key Storage Provider'
  KeyLength = 521
  HashAlgorithmName = 'SHA512'
  CAType = 'StandaloneRootCA'
  Force = $true
}
Install-AdcsCertificationAuthority @Params
```

This example specifies that the root CA's name will be "My Root CA" (the `-CACommonName` parameter), and the validity period of its certificate will be 10 years (`-ValidityPeriod` and `-ValidityPeriodUnits` parameters). The certificate will use elliptic curves cryptography instead of classic RSA and its key length will be 521 bits (`-CryptoProviderName` and `-KeyLength` parameters). For the hash algorithm, the example uses SHA-2 with a length of 512 bits (`-HashAlgorithmName`). Also, note that for a root CA, which is not to be a member of a domain/forest, you must pass "StandaloneRootCA" to the `-CAType` parameter. Therefore, the VM or the standalone server must not be a member of the domain/forest!

How were these cryptographic parameters chosen? The example uses the highest available for the chosen Key Storage Provider[21] because this is a root CA: it will rarely issue certificates, so

[21] https://learn.microsoft.com/en-us/windows/win32/seccertenroll/cng-key-storage-providers

there are no performance concerns, and the key must be as secure as possible. As for the hash length, it's usually recommended to have the hashlength to be of the same size (or close to it) as your elliptic curve. Note also, that while for demonstration purposes the example uses "Microsoft Software Key Storage Provider," this isn't a best practice for production environments—if one is available, you should really use a smart card or an HSM. Consult the documentation for your key storage provider to choose the most secure set of parameters. Also note, most companies do not have a HSM, so they use the available Microsoft Key Storage Providers.

Also check out Cryptographic Key Length Recommendations[22] by BlueCrypt—they allow you to conveniently compare recommendations from different organizations.

There are two more important concepts when you deal with certificate authorities: *CDP* and *AIA*.

CDP (*CRL Distribution Point*) is a location where the certificate authority will store a Certificate Revocation List (CRL). Your clients periodically download this list and use it to check to see whether a certificate presented to them is revoked. When a certificate is revoked, that means that the certificate should no longer be trusted. The main part of the revocation process is to put the serial number (a unique identifier) of a certificate into a CRL. If a certificate's serial number is in a CRL, it means the certificate is revoked.

To get a list of CDPs on a CA, use `Get-CACrlDistributionPoint`:

Example 17: Retrieving a list of CDPs for the root CA

```
Get-CACrlDistributionPoint
```

```
PublishToServer      : True
PublishDeltaToServer : True
AddToCertificateCdp  : False
AddToFreshestCrl     : False
AddToCrlCdp          : False
AddToCrlIdp          : False
Uri                  : C:\Windows\system32\CertSrv\CertEnroll\
                      <CAName><CRLNameSuffix><DeltaCRLAllowed>.crl

PublishToServer      : False
PublishDeltaToServer : False
AddToCertificateCdp  : False
AddToFreshestCrl     : False
AddToCrlCdp          : True
AddToCrlIdp          : False
Uri                  : ldap:///CN=<CATruncatedName><CRLNameSuffix>,
                      CN=<ServerShortName>,CN=CDP,CN=Public Key Services,
                        CN=Services,<ConfigurationContainer><CDPObjectClass>

PublishToServer      : False
PublishDeltaToServer : False
AddToCertificateCdp  : False
AddToFreshestCrl     : False
AddToCrlCdp          : False
```

[22] https://www.keylength.com/en/

```
AddToCrlIdp          : False
Uri                  : http://<ServerDNSName>/CertEnroll/
                      <CAName><CRLNameSuffix><DeltaCRLAllowed>.crl

PublishToServer      : False
PublishDeltaToServer : False
AddToCertificateCdp  : True
AddToFreshestCrl     : True
AddToCrlCdp          : False
AddToCrlIdp          : False
Uri                  : file://<ServerDNSName>/CertEnroll/
                      <CAName><CRLNameSuffix><DeltaCRLAllowed>.crl
```

If the root CA is not a member of the domain, then the LDAP CDP does nothing and can be deleted. If the root CA is offline, then the HTTP CDP does nothing and can be deleted. Therefore, in general, for an offline CA, the CDP should be copied to an intermediate or terminal (issuing) CA, so that certificates can access those CRLs.

Authority Information Access (AIA) is a place from which clients can download the certificate of a Certification Authority. When a client doesn't have a certificate's parent certificate, they'll use the AIA defined in a certificate to download the certificate's parent.

This isn't that important for a root CA: clients **must** have the root certificate installed locally to trust the PKI that uses that root CA. But AIA is tremendously important for issuing and intermediate CAs, because clients rarely have those certificates installed. This is especially important for internal CAs. Intermediate and issuing CAs should have their certificates published to endpoint devices via GPO or via an endpoint management system.

To get a list of AIA locations on a CA, use `Get-CAAuthorityInformationAccess`:

Example 18: Retrieving a list of AIA locations for the root CA

```
Get-CAAuthorityInformationAccess
```

```
AddToCertificateAia AddToCertificateOcsp Uri
------------------- -------------------- ---
              False                False C:\Windows\system32\CertSrv\CertEnroll\
                                  <ServerDNSName>_<CAName><CertificateName>.crt

              False                False ldap:///CN=<CATruncatedName>,CN=AIA,
    CN=Public Key Services,CN=Services,<ConfigurationContainer><CAObjectClass>

              False                False http://<ServerDNSName>/CertEnroll/
                                  <ServerDNSName>_<CAName><CertificateName>.crt

               True                False file://<ServerDNSName>/CertEnroll/
                                  <ServerDNSName>_<CAName><CertificateName>.crt
```

If the root CA is not a member of the domain, then the LDAP AIA does nothing and can be deleted. If the root CA is offline, then the HTTP AIA does nothing and can be deleted. Therefore, in general, for an offline CA, the AIA should be copied to an intermediate or terminal (issuing) CA, so that certificates can access the necessary files.

You can see that, by default, a Windows CA has a fair number of CDP and AIA locations configured. Often, you may not need all of them. Usually, a single location for all CAs in an organization is enough. This is dependent upon the access which specific endpoints may have to various resources within the organization.

Both CDP and AIA support several protocols which can be used to access these files. For maximum interoperability, however, you should just use HTTP. This also decreases the administrative load of updating multiple endpoints and monitoring them. However, in an Active Directory environment, publishing the certificate via LDAP makes the certificate easily available to all AD endpoints.

You might also note that the `Uri` properties of all these objects contain strings in angle brackets `<>`. These are variables specific to the Windows CA service. You can read about their meaning in the documentation[23], but in this section you won't use them.

Your clients learn which URIs to use as CDP or AIA locations by looking into a certificate's properties: as you can see in the output above, CDPs and AIAs have several `AddToCertificate...` properties—these define which of them will be included in a certificate issued by this CA.

You don't need any of the CDPs already defined on this Root CA: since this is an offline server, all those paths will be unavailable, anyway. You'll use a separate HTTP server for that.

⚠ This chapter doesn't discuss the installation of a web server to store CRLs and CA certificates—you must set it up by yourself.

Example 19: Removing the default CDPs from the root CA

```
Get-CACrlDistributionPoint | Remove-CACrlDistributionPoint -Force
```

```
RestartCA
---------
     True
     True
     True
     True
```

ℹ This section assumes that your Windows VM or server is named ROOTCA01. This section also assumes that you named your root CA "My Root CA."

If you take a look in a folder where a Windows CA publishes its CRL and CRT files by default, you'll see that the file name for the certificate is "ROOTCA01_My Root CA.crt" and the CRL is called "My Root CA.crl":

[23] https://learn.microsoft.com/en-us/powershell/module/adcsadministration/add-cacrldistributionpoint?view=windowsserver2019-ps

Example 20: Inspecting the default root certificate and CRL location

```
$Path = Join-Path -Path $env:SystemRoot -ChildPath 'system32\CertSrv\CertEnroll'
Get-ChildItem -Path $Path
```

```


    Directory: C:\Windows\system32\CertSrv\CertEnroll


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         7/11/2021  11:11 PM            534 My Root CA.crl
-a----         7/11/2021  11:11 PM            517 ROOTCA01_My Root CA.crt
```

For AIA, remove every entry except for the one pointing to the local file system, because it's not possible to add it back using the standard cmdlet `Add-CAAuthorityInformationAccess`. Therefore, the name of the certificate file will remain "ROOTCA01_My Root CA.crt."

Example 21: Removing all AIA locations from the root CA except the one for the local file system

```
Get-CAAuthorityInformationAccess |
    Where-Object -FilterScript {$_.Uri -notlike ('{0}*' -f $env:SystemRoot)} |
    Remove-CAAuthorityInformationAccess -Force
```

```


RestartCA
---------
     True
     True
     True
```

Next, you need to create new AIA and CDP locations. You can define several CDP locations, of different types. For CDP, it will be an HTTP URI and a local file system location. For AIA, you already have a file system location, so only an HTTP URI is left. You must have at least one local file system location, so you can grab the files from there and distribute to other locations.

To add a CDP location, use `Add-CACrlDistributionPoint`. Note that not all of the cmdlet parameters are compatible with each other and there are no parameter sets defined, because the compatibility of the parameters depends on the type of URI you use. If you try to use an incompatible combination, you'll get an error like this one:

Example 22: Attempting to add a new CDP with conflicting switches

```
$Params = @{
    Uri = 'http://pki.example.com/CDP/MyRootCA.crl'
    AddToCertificateCdp = $true
    AddToFreshestCrl = $true
    AddToCrlCdp = $true
    AddToCrlIdp = $true
    PublishToServer = $true
    Force = $true
}
Add-CACrlDistributionPoint @Params
```

```
Add-CACrlDistributionPoint : When an HTTP location is specified, the certificate
revocation list distribution point extension supports only the
AddToCertificateCdp, AddToFreshestCrl, and AddToCrlIdp options. At least one of
these options must be specified.
At line:1 char:1
+ Add-CACrlDistributionPoint -Uri 'http://pki.example.com/CDP/MyRootCA. ...
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidOperation: (http://pki.example.com/CDP/
      MyRootCA.crl:String) [Add-CACrlDistributionPoint], InvalidOptionException
    + FullyQualifiedErrorId :InvalidOption,Microsoft.CertificateServices.Adm...
```

There are three switches compatible with HTTP URLs: `AddToCertificateCdp`, `-AddToCrlIdp`, and `-AddToFreshestCrl`.

`-AddToFreshestCrl` specifies Delta CRL location. You won't use Delta CRLs in this chapter. Usually, you need Delta CRLs only when you revoke certificates often and your Base CRL becomes too large for clients to download in a reasonable time. This is an ultra-rare case for most folks.

`-AddToCrlIdp` is an Issuing Distribution Points (IDP) extension (OID 2.5.29.28) which is used for *partitioned CRLs*. Partitioned CRLs aren't supported by Windows PKI clients, therefore you won't use them in this chapter either.

However, the last one, `-AddToCertificateCdp`, is crucial: it specifies that all certificates issued by this CA will have a pointer to this CRL. Without it, your clients won't know where to look for a CRL and you effectively won't be able to revoke your certificates. All certificates you issue must have at least one CDP URI defined. That CDP URI must be available to all the clients.

Assume you have a web server available via the domain name "pki.example.com" and that you have created two folders in the root web directory: one for CDP files and one for AIA files. Let's define this web server as a CRL distribution point. Use the `-AddToCertificateCdp` parameter to add this location into all certificates issued by the root CA—the clients will use this URL to check if a certificate is revoked.

Example 23: Adding a new custom CDP for the root CA

```
$Params = @{
  Uri = 'http://pki.example.com/CDP/MyRootCA.crl'
  AddToCertificateCdp = $true
  Force = $true
}
Add-CACrlDistributionPoint @Params
```

```
RestartCA
---------
     True
```

The second CDP location is the local file system. You need this because you must be able to retrieve a CRL from the CA in order to put it on the HTTP server. Publication to the local file system generates a CRL file, ready for copying to any destination.

As you might notice, for simplification, these examples use filenames without spaces in both commands. This isn't a requirement—you can use different filenames, just don't forget to rename the file when you copy it to another location.

Example 24: Adding the file system CDP for the root CA

```
$Params = @{
  Path = $env:SystemRoot
  ChildPath = 'system32\CertSrv\CertEnroll\MyRootCA.crl'
}
$LocalCRLPath = Join-Path @Params
Add-CACrlDistributionPoint -Uri $LocalCRLPath -PublishToServer -Force
```

```
RestartCA
---------
     True
```

Next, add an Authority Information Access (AIA) extension. This will insert a location of the root certificate file into all certificates issued by your Root CA, thanks to the `-AddToCertificateAia` parameter. Again, the filename is without spaces but, as you might remember, you didn't touch the existing AIA definition pointing to the local filesystem. You'll need to rename the file manually when copying it over to the web server, or just use the filename with spaces in the `-AddToCertificateAia` parameter—the choice is yours.

Example 25: Adding a custom AIA location for the root CA certificate

```
$Params = @{
  Uri = 'http://pki.example.com/AIA/MyRootCA.crt'
  AddToCertificateAia = $true
  Force = $true
}
Add-CAAuthorityInformationAccess @Params
```

```
RestartCA
---------
     True
```

Now, take a look at the current CRL you have already issued at this CA. The most convenient and easy way to do that in PowerShell is to install PSPKI[24], a module by Vadims Podāns. Since this is an offline machine, download the module manually from the PowerShell Gallery[25] and install it on the computer, then import it: `Import-Module -Name 'PSPKI'`.

To get information about a CRL use the `Get-CertificateRevocationList` cmdlet:

Example 26: Retrieving information about a CRL

```
$Params = @{
  Path = $env:SystemRoot
  ChildPath = 'system32\CertSrv\CertEnroll\My Root CA.crl'
}
$LocalCRLPath = Join-Path @Params
Get-CertificateRevocationList -Path $LocalCRLPath
```

```
Version              : 2
Type                 : BaseCrl
IssuerName           :
            System.Security.Cryptography.X509Certificates.X500DistinguishedName
Issuer               : CN=My Root CA
ThisUpdate           : 7/11/2021 11:01:52 PM
NextUpdate           : 7/19/2021 11:21:52 AM
SignatureAlgorithm   : sha512ECDSA (1.2.840.10045.4.3.4)
CRLNumber            : 1
Extensions           : {Authority Key Identifier (2.5.29.35),
                       CA Version (1.3.6.1.4.1.311.21.1),
                       CRL Number (2.5.29.20),
                       Next CRL Publish (1.3.6.1.4.1.311.21.4)...}
RevokedCertificates : {}
RawData              : {48, 130, 2, 18...}
Handle               :
             System.Security.Cryptography.X509Certificates.SafeCRLHandleContext
Thumbprint           :
               075BD031E4331D21D5C4B6E841B36D2565DA035CF08785C4135789023D9E1D5D
```

[24] https://www.pkisolutions.com/tools/pspki/

[25] https://www.powershellgallery.com/packages/PSPKI/

You can see that the `NextUpdate` property in this CRL is set to about 7 days ahead, so you **must** update the CRL by that date and make it available to your clients. Then repeat in a week. And again and again. Given that this node requires manual intervention to bring it online and issue commands on it, this process seems terribly annoying and unproductive (because you most probably won't issue and revoke certificates on the root CA that often).

A solution is to increase the CRL refresh interval to a more reasonable six months:

Example 27: Adjusting the CRL update period from days to months

```
certutil -setreg CA\CRLPeriod Months
```

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\CertSvc\Configuration\
                                                      My Root CA\CRLPeriod:

Old Value:
  CRLPeriod REG_SZ = Weeks

New Value:
  CRLPeriod REG_SZ = Months
CertUtil: -setreg command completed successfully.
The CertSvc service may need to be restarted for changes to take effect.
```

Example 28: Setting the CRL update interval to 6 months

```
certutil -setreg CA\CRLPeriodUnits 6
```

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\CertSvc\Configuration\
                                                  My Root CA\CRLPeriodUnits:

Old Value:
  CRLPeriodUnits REG_DWORD = 1

New Value:
  CRLPeriodUnits REG_DWORD = 6
CertUtil: -setreg command completed successfully.
The CertSvc service may need to be restarted for changes to take effect.
```

Best practices say that the root CA is offline, except when it **must** be online. The most common reason is to renew the certificate for a subordinate CA (or a chaining CA). Practically, the root CA must have OS updates on some regular schedule (not necessarily monthly, but regularly). Also the CRL for the root CA must be updated and copied to CDPs on a regular basis (dependent on the CRL update period as discussed in Example 27 and Example 28). If you do not assign a NIC to the root CA, these procedures can be quite difficult. These choices are dependent on your company's overall security posture.

As another practical matter, if there is no NIC on the root CA, installation of PowerShell modules can be challenging. Without a local NIC, modules must be downloaded remotely (including dependencies), copied to the target, and then installed. This is in comparison to a simple `Install-Module` cmdlet execution.

By default, this CA will issue certificates with a validity period of one year maximum. Usually, you'd prefer something longer for CA certificates—let's increase it up to five years. For this, use the `ValidityPeriodUnits` property:

Example 29: Setting the CA certificate validity interval to 5 years

```
certutil -setreg CA\ValidityPeriodUnits 5
```

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\CertSvc\Configuration\
                                            My Root CA\ValidityPeriodUnits:

Old Value:
  ValidityPeriodUnits REG_DWORD = 1

New Value:
  ValidityPeriodUnits REG_DWORD = 5
CertUtil: -setreg command completed successfully.
The CertSvc service may need to be restarted for changes to take effect.
```

By default, the validity period is measured in years. You can check if this is true:

Example 30: Confirming that the CA certificate validity period is in years

```
certutil -getreg CA\ValidityPeriod
```

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\CertSvc\Configuration\
                                                  My Root CA\ValidityPeriod:

  ValidityPeriod REG_SZ = Years
CertUtil: -getreg command completed successfully
```

If in your installation it returns something other than *Years*, use the following command to correct the period:

```
certutil -setreg CA\ValidityPeriod Years
```

To apply all these changes, you must restart the CA service. No need to restart the whole server, just the service is enough (but if you prefer, you can reboot the machine, of course):

Example 31: Restarting the certificate service after making changes to the root CA

```
Restart-Service -Name CertSvc
```

Now, you can issue an updated CRL on this CA (it will be empty and that's OK):

Example 32: Inspecting the root CA's default certificate and CRL location before issuing a new CRL

```
$Pth = Join-Path -Path $env:SystemRoot -ChildPath 'system32\CertSrv\CertEnroll'
Get-ChildItem -Path $Pth
```

```


    Directory: C:\Windows\system32\CertSrv\CertEnroll


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         7/11/2021  11:11 PM            534 My Root CA.crl
-a----         7/11/2021  11:11 PM            517 ROOTCA01_My Root CA.crt
```

Example 33: Issuing a new CRL with the updated configuration

```
certutil -CRL
```

```
CertUtil: -CRL command completed successfully.
```

Note that the new CRL file has the name we specified in the CDP configuration earlier:

Example 34: Inspecting the new CRL, which has a new file name defined in the CA configuration

```
$Pth = Join-Path -Path $env:SystemRoot -ChildPath 'system32\CertSrv\CertEnroll'
Get-ChildItem -Path $Pth
```

```


    Directory: C:\Windows\system32\CertSrv\CertEnroll


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         7/11/2021  11:11 PM            534 My Root CA.crl
-a----         7/11/2021  11:22 PM            327 MyRootCA.crl
-a----         7/11/2021  11:11 PM            517 ROOTCA01_My Root CA.crt
```

Example 35: Retrieving information about the new CRL

```
$Params = @{
  Path = $env:SystemRoot
  ChildPath = 'system32\CertSrv\CertEnroll\MyRootCA.crl'
}
$RootCRLPath = Join-Path @Params
Get-CertificateRevocationList -Path $RootCRLPath
```

```
Version              : 2
Type                 : BaseCrl
IssuerName           :
System.Security.Cryptography.X509Certificates.X500DistinguishedName

Issuer               : CN=My Root CA
ThisUpdate           : 7/11/2021 11:12:45 PM
NextUpdate           : 1/12/2022 10:32:45 AM
SignatureAlgorithm   : sha512ECDSA (1.2.840.10045.4.3.4)
CRLNumber            : 2
Extensions           : {Authority Key Identifier (2.5.29.35),
                       CA Version (1.3.6.1.4.1.311.21.1),
                       CRL Number (2.5.29.20),
                       Next CRL Publish (1.3.6.1.4.1.311.21.4)}
RevokedCertificates : {}
RawData              : {48, 130, 1, 67...}
Handle               :
System.Security.Cryptography.X509Certificates.SafeCRLHandleContext

Thumbprint           :
A6E99F8127704FAFB5D21EBCD54B01C19A106EA8D54935C73C05702CEDF9C07D
```

Great, the next update date is now six months in the future!

15.5.1.2.2 Issuing CA

The Issuing CA is a server where you issue certificates to endpoints (computers and users), including code signing certificates. Go to the computer that will be an issuing CA in your infrastructure (this chapter assumes it has Windows Server installed on it already and is a member of your AD domain).

You may have several issuing and root CAs in your environment, including several issuing CAs under a single root CA. If you have a large environment, you may have a third tier of CAs, with the intermediate (middle) tier known as "policy CAs." Similar to delegation of OUs in large environments, policy CAs allow you to delegate CA control to administrative tiers.

First, we need to deliver the root CA's certificate to the issuing CA (copy it manually or via RDP copy-and-paste).

The location of the root CA's certificate was discussed in Example 20.

This example places the file in the root of the system volume (the `C:` drive), but you can change it to meet your needs. Import this certificate into the local "Trusted Root Certification Authorities" computer store:

Example 36: Importing the root CA's certificate on an issuing CA

```
$JPParams = @{
  Path = $env:SystemDrive
  ChildPath = 'ROOTCA01_My Root CA.crt'
}
$RootCRTPath = Join-Path @JPParams
$ICParams = @{
  FilePath = $RootCRTPath
  CertStoreLocation = 'Cert:\LocalMachine\Root\'
}
Import-Certificate @ICParams
```

```
   PSParentPath: Microsoft.PowerShell.Security\Certificate::LocalMachine\Root

Thumbprint                                Subject
----------                                -------
EABE909367E3D4F80B399AB8CC2677A5C9686C1F  CN=My Root CA
```

Next, install the AD CS role:

Example 37: Installing the Active Directory Certificate Services CA feature on the issuing CA

```
Install-WindowsFeature -Name ADCS-Cert-Authority -IncludeManagementTools
```

```
Success Restart Needed Exit Code      Feature Result
------- -------------- ---------      --------------
True    No             Success        {Active Directory Certificate Services...
```

Then install the CA itself. In this case, the CA's name is "My Issuing CA," the length of the elliptic curve and hash function are both 256 bits, and the CA type is of course "EnterpriseSubordinateCA."

Example 38: Configuring the issuing Certificate Authority

```
$Params = @{
  CACommonName = 'My Issuing CA'
  CryptoProviderName = 'ECDSA_P256#Microsoft Software Key Storage Provider'
  KeyLength = 256
  HashAlgorithmName = SHA256
  CAType = 'EnterpriseSubordinateCA'
  Force = $true
}
Install-AdcsCertificationAuthority @Params
```

```
WARNING: The Active Directory Certificate Services installation is incomplete.
To complete the installation, use the request file "C:\CA02.ad.example.net_My
Issuing CA.req" to obtain a certificate from the parent CA. Then, use the
Certification Authority snap-in to install the certificate. To complete this
procedure, right-click the node with the name of the CA, and then click Install
CA Certificate. The operation completed successfully. 0x0 (WIN32: 0)

ErrorId ErrorString
------- -----------
    398 The Active Directory Certificate Services installation is incomplete. To
        complete the installation, use the request file "C:\CA02.ad.example.net_
        My Issuing CA.req" to obtain a certificate ...
```

The installation is incomplete, because we need to obtain the certificate of this CA from our root CA, and, since the root CA is offline, we must do this manually. That message tells us that at “C:\CA02.ad.example.net_My Issuing CA.req” a generated certificate request is waiting. Let’s look at it with the help of `Get-CertificateRequest` from the PSPKI module:

Example 39: Inspecting the certificate request file for signing the issuing CA’s certificate

```
Get-CertificateRequest -Path 'C:\CA02.ad.example.net_My Issuing CA.req'
```

```
RequestType        : PKCS10
SubjectDn          :
            System.Security.Cryptography.X509Certificates.X500DistinguishedName
ExternalData       :
Version            : 1
SubjectName        :
            System.Security.Cryptography.X509Certificates.X500DistinguishedName
Subject            : CN=My Issuing CA, DC=ad, DC=example, DC=net
PublicKey          : System.Security.Cryptography.X509Certificates.PublicKey
Extensions         : {CA Version (1.3.6.1.4.1.311.21.1),
                     Subject Key Identifier (2.5.29.14),
                     Certificate Template Name (1.3.6.1.4.1.311.20.2),
                     Key Usage (2.5.29.15)...}
Attributes         : {0}
SignatureAlgorithm : sha256ECDSA (1.2.840.10045.4.3.2)
SignatureIsValid   : True
RawData            : {48, 130, 1, 242...}
```

Using this method you can validate whether all request parameters are correct. To proceed, copy the request file to the root CA computer. The following commands are to be executed on the root CA, not on the issuing CA.

To send the request to the certification authority, use `Submit-CertificateRequest`. Note that we save the CA object in a variable—that’s because we’ll need it later.

Example 40: Submitting the issuing CA's certificate request on the root CA

```
$CA = Connect-CertificationAuthority
$Params = @{
  Path = $env:SystemDrive
  ChildPath = 'CA02.ad.example.net_My Issuing CA.req'
}
$Path = Join-Path @Params
Submit-CertificateRequest -Path $Path -CertificationAuthority $CA
```

```
CertificationAuthority : PKI.CertificateServices.CertificateAuthority
RequestID              : 2
Status                 : UnderSubmission
Certificate            :
ErrorInformation       : Taken Under Submission
```

If you check on the request, you'll see that the certificate will be issued using the "SubCA" template—this is a special template, which Windows PKI uses for subordinate (child) CA certificates.

Example 41: Checking the status of the submitted request on the root CA

```
Get-PendingRequest -CertificationAuthority $CA
```

```
RequestID               : 2
Request.RequesterName   : ROOTCA01\Administrator
Request.SubmittedWhen   : 8/29/2021 9:20:59 PM
Request.CommonName      : My Issuing CA
CertificateTemplate     : SubCA
CertificateTemplateOid  : SubCA
RowId                   : 2
ConfigString            : ROOTCA01\My Root CA
Table                   : Request
Properties              : {[RequestID, 2],
                          [Request.RequesterName, ROOTCA01\Administrator],
                          [Request.SubmittedWhen, 8/29/2021 9:20:59 PM],
                          [Request.CommonName, My Issuing CA]...}
```

Now, you must approve the pending certificate request:

Example 42: Approving the issuing CA's certificate request on the root CA

```
$Request = Get-PendingRequest -CertificationAuthority $CA
Approve-CertificateRequest -Request $Request
```

```
HResult      StatusMessage
-------      -------------
0            The certificate '2' was issued.
```

Example 43: Displaying the approved certificate request on the root CA

```
$IssuedRequest = Get-IssuedRequest -CertificationAuthority $CA -RequestID 2
$IssuedRequest
```

```
RequestID                : 2
Request.RequesterName    : ROOTCA01\Administrator
CommonName               : My Issuing CA
NotBefore                : 8/29/2021 9:11:49 PM
NotAfter                 : 8/29/2026 9:21:49 PM
SerialNumber             : 6b00000002a4208d0a6e2af611000000000002
CertificateTemplate      : SubCA
CertificateTemplateOid   : SubCA
RowId                    : 2
ConfigString             : ROOTCA01\My Root CA
Table                    : Request
Properties               : {[RequestID, 2],
                           [Request.RequesterName, ROOTCA01\Administrator],
                           [CommonName, My Issuing CA],
                           [NotBefore, 8/29/2021 9:11:49 PM]...}
```

The last step on the root CA is to export the approved request into a file. Note the "RequestID_-2.cer" file in the listing below: copy this file to the issuing CA. Note also that the example below writes to the root of the system drive: this will not be allowed for non-administrative users.

Example 44: Exporting the issued certificate so that it can be copied to the issuing CA

```
$Item = Get-Item -Path $Env:SystemDrive
Receive-Certificate -RequestRow $IssuedRequest -Path $Item | Format-List
```

```
Thumbprint: E1BF218D5C4B58B16BABC003558222975B717E3A
Subject:    CN=My Issuing CA, DC=ad, DC=example, DC=net
```

Example 45: Inspecting the system drive (C:) to show the newly issued certificate file

```
Get-ChildItem -Path $Env:SystemDrive
```

```
    Directory: C:\


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----          7/11/2021  11:11 PM                CAConfig
d-----          4/19/2021   3:11 PM                PerfLogs
d-r---          4/18/2021  12:42 PM                Program Files
d-----          4/18/2021  12:42 PM                Program Files (x86)
d-r---          4/18/2021  12:41 PM                Users
d-----          8/29/2021   6:44 PM                Windows
-a----          8/29/2021   9:59 PM            774 CA02.ad.example.net_My Issuing CA.req
-a----          8/29/2021  11:24 PM           1078 RequestID_2.cer
```

We've finished with the root CA—execute the subsequent commands on the issuing CA. Copy the new certificate (`RequestID_2.cer` in this example) to the issuing CA computer. Using RDP copy-and-paste is often the simplest way to accomplish this. If this is not available to you, using a file share is often the next way to attempt it. In the worst case, using a mounted drive that you transfer between VMs, is a way that always works, albeit with the most administrative overhead.

The last step in starting up a subordinate CA is to install the signed certificate to it. Unfortunately, there is no native PowerShell cmdlet for this. Even PSPKI doesn't have a function for that yet[26]. But the "legacy" command-line tools are here to help:

Example 46: Importing the certificate, signed by the root CA, on the issuing CA

```
certutil -installcert "$Env:SystemDrive\RequestID_2.cer"
```

```
CertUtil: -installCert command completed successfully.
The CertSvc service may need to be restarted for changes to take effect.
```

At this stage, you may receive the following error:

```
CertUtil: -installCert command FAILED: 0x80092013
          (-2146885613 CRYPT_E_REVOCATION_OFFLINE)
CertUtil: The revocation function was unable to check
          revocation because the revocation server was offline.
```

This means that the issuing CA machine can't reach the CDP that you defined in the Root CA configuration step. The potential reasons are countless—maybe it's a firewall issue, or you didn't add the DNS record, or perhaps you didn't configure that web server at all. To fix this, make the CDP available to the issuing CA and all your clients now—never ignore certificate revocation problems!

Now you can start the "CertSvc" service and it will work.

[26] https://github.com/PKISolutions/PSPKI/issues/151

Example 47: Starting the certificate service on the issuing CA

```
Start-Service -Name 'CertSvc'
```

However, you aren't finished. Recall how we configured AIA and CDP locations for the root CA? A subordinate CA is no different, because its default configuration is the same. Once again, you need to:

1. Remove all CDP/AIA locations which won't be used in your infrastructure.
2. Make sure the correct locations will be baked into certificates issued by this server.

Check what locations are currently configured:

Example 48: Displaying the default CDPs for the issuing CA

```
Get-CACrlDistributionPoint | Format-List
```

```
PublishToServer      : True
PublishDeltaToServer : True
AddToCertificateCdp  : False
AddToFreshestCrl     : False
AddToCrlCdp          : False
AddToCrlIdp          : False
Uri                  : C:\Windows\system32\CertSrv\CertEnroll\
                       <CAName><CRLNameSuffix><DeltaCRLAllowed>.crl

PublishToServer      : True
PublishDeltaToServer : True
AddToCertificateCdp  : True
AddToFreshestCrl     : True
AddToCrlCdp          : True
AddToCrlIdp          : False
Uri                  : ldap:///CN=<CATruncatedName><CRLNameSuffix>,
                       CN=<ServerShortName>,CN=CDP,CN=Public Key Services,
                       CN=Services,<ConfigurationContainer><CDPObjectClass>

PublishToServer      : False
PublishDeltaToServer : False
AddToCertificateCdp  : False
AddToFreshestCrl     : False
AddToCrlCdp          : False
AddToCrlIdp          : False
Uri                  : http://<ServerDNSName>/CertEnroll/
                       <CAName><CRLNameSuffix><DeltaCRLAllowed>.crl

PublishToServer      : False
PublishDeltaToServer : False
AddToCertificateCdp  : False
AddToFreshestCrl     : False
AddToCrlCdp          : False
AddToCrlIdp          : False
Uri                  : file://<ServerDNSName>/CertEnroll/
                       <CAName><CRLNameSuffix><DeltaCRLAllowed>.crl
```

Example 49: Displaying the default AIA locations for the issuing CA

```
Get-CAAuthorityInformationAccess | Format-List
```

```
AddToCertificateAia : False
AddToCertificateOcsp: False
Uri                 : C:\Windows\system32\CertSrv\CertEnroll\
                      <ServerDNSName>_<CAName><CertificateName>.crt

AddToCertificateAia : True
AddToCertificateOcsp: False
Uri                 : ldap:///CN=<CATruncatedName>,CN=AIA,
                      CN=Public Key Services,CN=Services,
                      <ConfigurationContainer><CAObjectClass>

AddToCertificateAia : False
AddToCertificateOcsp: False
Uri                 : http://<ServerDNSName>/CertEnroll/
                      <ServerDNSName>_<CAName><CertificateName>.crt

AddToCertificateAia : False
AddToCertificateOcsp: False:
Uri                 : file://<ServerDNSName>/CertEnroll/
                      <ServerDNSName>_<CAName><CertificateName>.crt
```

Look at the folder content where the files are put by default:

Example 50: Inspecting the default certificate and CRL location on the issuing CA

```
Get-ChildItem -Path 'C:\Windows\System32\CertSrv\CertEnroll'
```

```
    Directory: C:\Windows\System32\CertSrv\CertEnroll


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        9/4/2021  10:27 PM            741 CA02.ad.example.net_My Issuing CA.crt
-a----        9/4/2021  10:36 PM            592 My Issuing CA+.crl
-a----        9/4/2021  10:36 PM            791 My Issuing CA.crl
```

Remove unneeded locations and add the ones which you’ll use, using the same principles as with the root CA (in this case, meaning to keep only file and HTTP entries):

Example 51: Removing the default CDPs from the issuing CA

```
Get-CACrlDistributionPoint | Remove-CACrlDistributionPoint -Force
```

```
RestartCA
---------
     True
     True
     True
     True
```

By executing Example 51, you remove all of the CDPs (CRL Distribution Points) shown by Example 48. This may, or may not, be appropriate for your production environment. Example 48 shows CDPs hosted: on the local file system, on a remote file system, on a web server, and in Active Directory. In Example 53 below, the web server CDP is added back (with a generic URL). In Example 54 below, the local file system CDP is added back. For your environment, you may want to retain or add values back for some of the other options (especially Active Directory).

Example 52: Removing all AIA locations from the issuing CA except the one for the local file system

```
Get-CAAuthorityInformationAccess |
  Where-Object -FilterScript {$_.Uri -notlike ('{0}*' -f $env:SystemRoot)} |
    Remove-CAAuthorityInformationAccess -Force
```

```
RestartCA
---------
     True
     True
     True
```

By executing Example 52, you remove all of the AIAs (Authority Information Access locations) shown by Example 49, except for the AIA hosted on the local file system. This may, or may not, be appropriate for your production environment. Example 49 shows AIAs hosted: on the local file system, on a remote file system, on a web server, and in Active Directory. In Example 55 below, the web server CDP is added back (with a generic URL). For your environment, you may want to retain or add values back for some of the other options (especially Active Directory).

Example 53: Adding the custom CDP for the issuing CA

```
$Params = @{
  Uri = 'http://pki.example.com/CDP/MyIssuingCA.crl'
  AddToCertificateCdp = $true
  Force = $true
}
Add-CACrlDistributionPoint @Params
```

```
RestartCA
---------
     True
```

Example 54: Adding back the file system CDP for the issuing CA

```
$Params = @{
  Path = $env:SystemRoot
  ChildPath = 'system32\CertSrv\CertEnroll\MyIssuingCA.crl'
}
$Path = Join-Path @Params
Add-CACrlDistributionPoint -Uri $Path -PublishToServer -Force
```

```
RestartCA
---------
     True
```

Example 55: Adding the custom AIA location for the issuing CA

```
$Params = @{
  Uri = 'http://pki.example.com/AIA/MyIssuingCA.crt'
  AddToCertificateAia = $true
  Force = $true
}
Add-CAAuthorityInformationAccess @Params
```

```
RestartCA
---------
     True
```

As you might remember, on the root CA we changed the CRL validity period, because that's an offline server and having a CRL valid only for a week is really inconvenient. For online certification authorities, like "My Issuing CA," this isn't a problem, because you can easily automate CRL release cycles with online servers. Therefore, you won't change this setting here. Let's check that it's set to one week:

Example 56: Confirming that the issuing CA's CRL update period is in weeks

```
certutil -getreg CA\CRLPeriod
```

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\CertSvc\Configuration\
                                                   My Issuing CA\CRLPeriod:

  ValidityPeriod REG_SZ = Weeks
CertUtil: -getreg command completed successfully.
```

Example 57: Confirming that the issuing CA's CRL update interval is 1 week

```
certutil -getreg CA\CRLPeriodUnits
```

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\CertSvc\Configuration\
                                              My Issuing CA\CRLPeriodUnits:

  ValidityPeriodUnits REG_DWORD = 1
CertUtil: -getreg command completed successfully.
```

All seems good! Now you can restart the CA service and reissue the CRL to the newly defined location:

Example 58: Restarting the certificate service on the issuing CA after making changes

```
Restart-Service -Name CertSvc
certutil -CRL
```

Example 59: Inspecting the default certificate and CRL location on the issuing CA after the changes have been applied

```
Get-ChildItem -Path 'C:\Windows\System32\CertSrv\CertEnroll'
```

```
    Directory: C:\Windows\System32\CertSrv\CertEnroll


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        9/4/2021  10:27 PM            741 CA02.ad.example.net_My Issuing CA.crt
-a----        9/4/2021  10:36 PM            592 My Issuing CA+.crl
-a----        9/4/2021  10:36 PM            791 My Issuing CA.crl
-a----        9/4/2021  10:57 PM            358 MyIssuingCA.crl
```

Grab this new "MyIssuingCA.crl" file and the CRT file and place them onto the web server for your clients to download when they'll check for revocation.

We have not covered creating a web-based AIA or CDP in this chapter or configuring those as defaults in the Certificate Authority. These are often hosted on an issuing CA using IIS, but in larger environments may be hosted on a separate web server. When hosted on a single issuing CA, the local file location can be used as the source of an IIS virtual directory (this is `C:\Windows\System32\CertSrv\CertEnroll` by default). But, as you see in Example 53 and Example 55, creating web-based AIAs and CDPs is something which is usually done. If you host a web-based AIA or CDP on a separate web server (or even if you have multiple issuing CAs), then you will need to script and schedule the copying of CRLs and CA certificates to the target location. This is also true for the root certificate and CRL.

15.5.1.2.3 Create a Signing Template

You might think that's it; you're ready to issue certificates. Not quite, unfortunately. In Windows PKI, there's a concept called *certificate templates*. A certificate template is a set of properties which define how a certificate can be issued, for which purposes, who can issue it, to whom, and where that certificate can be stored.

For standalone certificate authorities, templates are stored in the registry. However, we are primarily concerned with enterprise certificate authorities in this chapter. For enterprise certificate authorities, certificate templates are AD DS objects which are located under the Configuration Partition:

```
CN=Certificate Templates,CN=Public Key Services,CN=Services,CN=Configuration,
DC=<your AD forest's distinguished name>
```

You can get a list of them with the `Get-CertificateTemplate` command (part of the PSPKI module):

Example 60: Retrieving a list of certificate templates

```
Get-CertificateTemplate
```

```
DisplayName                          SupportedCA          AutoenrollmentAllowed
-----------                          -----------          ---------------------
Administrator                        Windows 2000 Server                  False
Root Certification Authority         Windows 2000 Server                  False
CA Exchange                          Windows Server 2003 Enterprise Ed... False
CEP Encryption                       Windows 2000 Server                  False
Authenticated Session                Windows 2000 Server                  False
Code Signing                         Windows 2000 Server                  False
Cross Certification Authority        Windows Server 2003 Enterprise Ed... False
Trust List Signing                   Windows 2000 Server                  False
Directory Email Replication          Windows Server 2003 Enterprise Ed... True
Domain Controller                    Windows 2000 Server                  False
Domain Controller Authentication     Windows Server 2003 Enterprise Ed...  True
Basic EFS                            Windows 2000 Server                  False
EFS Recovery Agent                   Windows 2000 Server                  False
Enrollment Agent                     Windows 2000 Server                  False
Exchange Enrollment Agent (Offli...  Windows 2000 Server                  False
Exchange User                        Windows 2000 Server                  False
Exchange Signature Only              Windows 2000 Server                  False
IPSec (Offline request)              Windows 2000 Server                  False
IPSec                                Windows 2000 Server                  False
Kerberos Authentication              Windows Server 2003 Enterprise Ed...  True
Key Recovery Agent                   Windows Server 2003 Enterprise Ed...  True
Computer                             Windows 2000 Server                  False
Enrollment Agent (Computer)          Windows 2000 Server                  False
OCSP Response Signing                Windows Server 2008 Enterprise Ed... False
Router (Offline request)             Windows 2000 Server                  False
RAS and IAS Server                   Windows Server 2003 Enterprise Ed...  True
Smartcard Logon                      Windows 2000 Server                  False
Smartcard User                       Windows 2000 Server                  False
Subordinate Certification Authority  Windows 2000 Server                  False
User                                 Windows 2000 Server                  False
User Signature Only                  Windows 2000 Server                  False
Web Server                           Windows 2000 Server                  False
Workstation Authentication           Windows Server 2003 Enterprise Ed...  True
```

Most Certificate Authority functions can be executed/controlled using the `certutil.exe` native binary. However, `certutil.exe` is not particularly user friendly. For example, `certutil.exe` has several command options to deal with listing certificate templates (as well as creating and deleting them). But those options do not have the filtering or data presentation capabilities in PowerShell, which makes using the built-in cmdlets and various third-party modules much easier than using `certutil.exe`.

You should never use the built-in certificate templates and instead create your own. There are several reasons:

1. The built-in templates don't have all features available on the modern OSes. See that SupportedCA property? While not going deep into the details, the more recent the OS, the more features are available in the template.
2. It's good to always have a default configuration set, so you could roll back to it and try again if you mess settings of your certificate template.

So treat these just as you do default group policies: use them for reference. Don't modify them; instead, if you need to make changes to the templates, then make copies and work with those template copies.

Usually, you create a copy of a template via the GUI, but this book isn't about GUIs. ;) Unfortunately, Microsoft doesn't provide a built-in way to duplicate certificate templates in PowerShell. An external module ADCSTemplate[27] exists, but using it is tricky. To create a new Certificate Template, this module needs a description of that template in JSON form. Usually, you get this JSON using the `Export-ADCSTemplate` cmdlet, but for this you need a template with all the required properties to be in your AD DS already. To spare you from dealing with the GUI, this chapter includes a JSON object for a code signing template:

Example 61: A certificate template for code signing as a JSON object

```
$JSONRaw = @'
{
    "name": "MyCodeSigning",
    "displayName": "My Code Signing",
    "objectClass": "pKICertificateTemplate",
    "flags": 131616,
    "revision": 100,
    "msPKI-Cert-Template-OID": "1.3.6.1.4.1.311.21.8.6625186.9886736.8734392.
15429154.6687822.90.4158550.10744840",
    "msPKI-Certificate-Application-Policy": ["1.3.6.1.5.5.7.3.3"],
    "msPKI-Certificate-Name-Flag": -2113929216,
    "msPKI-Enrollment-Flag": 32,
    "msPKI-Minimal-Key-Size": 256,
    "msPKI-Private-Key-Flag": 101056512,
    "msPKI-RA-Application-Policies": [
        "msPKI-Asymmetric-Algorithm`PZPWSTR`ECDSA_P256`msPKI-Hash-Algorithm`
PZPWSTR`SHA256`msPKI-Key-Usage`DWORD`2`msPKI-Symmetric-Algorithm`
PZPWSTR`3DES`msPKI-Symmetric-Key-Length`DWORD`168`"
    ],
    "msPKI-RA-Signature": 0,
    "msPKI-Template-Minor-Revision": 2,
```

[27] https://www.powershellgallery.com/packages/ADCSTemplate

```
    "msPKI-Template-Schema-Version": 4,
    "pKICriticalExtensions": ["2.5.29.15"],
    "pKIDefaultKeySpec": 2,
    "pKIExpirationPeriod": [0, 64, 57, 135, 46, 225, 254, 255],
    "pKIExtendedKeyUsage": ["1.3.6.1.5.5.7.3.3"],
    "pKIKeyUsage": [128, 0],
    "pKIMaxIssuingDepth": 0,
    "pKIOverlapPeriod": [0, 128, 166, 10, 255, 222, 255, 255]
}
'@
```

Note that the JSON has external line breaks due to book formatting limitations. To correct this limitation, run the following command:

```
$JSONRaw = $JSONRaw -replace '\r?\n'
```

To create the JSON output, the example duplicates the built-in Code Signing template, upgrades its SupportedCA property to Windows Server 2019, and declares that this template is for issuing ECDSA certificates. Feel free to tweak it, according to your needs!

To import this template in your environment, run the following:

Example 62: Importing a certificate template from a JSON string

```
New-ADCSTemplate -DisplayName 'My Code Signing' -JSON $JSONRaw
```

Unfortunately, this cmdlet doesn't import the `msPKI-RA-Application-Policies` property at the time of writing, which describes what crypto algorithms will be used for certificates issued by this template. You'll have to fix it manually:

Example 63: Manually adding the missing property to the imported AD template using the same JSON string

```
$JSON = ConvertFrom-Json -InputObject $JSONRaw
$DN = (Get-ADCSTemplate -DisplayName 'My Code Signing').DistinguishedName
Set-ADObject -Identity $DN -Add @{
  'msPKI-RA-Application-Policies' = $JSON.'msPKI-RA-Application-Policies'
}
```

The `Set-ADObject` cmdlet is part of the `RSAT-ADDS` Windows Feature and is not installed by default on Certificate Authority servers. It also requires elevated permissions (write-all-properties on the particular policy involved, typically assigned only to Domain Admins and the object's Creator/Owner). To install the feature on the CA server, you use the `Install-WindowsFeature -Name RSAT-ADDS -IncludeAllSubFeature` cmdlet from an elevated PowerShell session. Alternatively, after retrieving the distinguished name on the CA server, you can run the cmdlet in an elevated PowerShell session on a domain controller (where `RSAT-ADDS` is already installed by default).

The last step for the template is to publish the template on your issuing CA, therefore allowing it to issue certificates using this template:

Example 64: Publishing the AD CS certificate template to the issuing CA

```
Get-CertificationAuthority | Get-CATemplate |
  Add-CATemplate -DisplayName 'My Code Signing' | Set-CATemplate
```

```
DisplayName                             Templates
-----------                             ---------
My Issuing CA                           {MyCodeSigning}
```

15.5.1.2.4 Root Certificate Deployment

Before you use a certificate from your PKI, you need to make sure that the certificate of its root certification authority is distributed among all potential users of the certificate in your organization and all consumers of your signed scripts. You have several options here:

15.5.1.2.5 Manual Import

You can always import a certificate on any machine by manually executing commands. You would certainly need this for non-domain clients, for example.

Example 65: Manually importing a root CA certificate from file on a client

```
$Path = Join-Path $env:SystemDrive -ChildPath 'ROOTCA01_My Root CA.crt'
$Params = @{
  FilePath = $Path
  CertStoreLocation = 'Cert:\LocalMachine\Root\'
}
Import-Certificate @Params
```

15.5.1.2.6 AD DS Certificate Containers

A preferred way to deploy root CA certificates to domain-joined machines is to use the built-in functionality of Active Directory Domain Services: *certificate containers*. Those are containers located under:

```
CN=Public Key Services, CN=Services, CN=Configuration,
DC=<your AD forest's distinguished name>
```

There are several of them, but we're interested in the one storing root CA certificates: `CN=Certification Authorities`.

The beauty of certificate containers is that by using them, you're making certificates accessible to ALL your domain clients at once. Our trusty PSPKI module has several functions to help with certificate containers. Here's how to use them:

Example 66: Adding a root CA certificate to an AD DS certificate container

```
$Path = Join-Path -Path $env:SystemDrive -ChildPath 'ROOTCA01_My Root CA.crt'
$Cert = [Security.Cryptography.X509Certificates.X509Certificate2]::new($Path)
$RCAADContainer = Get-AdPkiContainer -ContainerType RootCA
$AAParams = @{
  AdContainer = $RCAADContainer
  Certificate = $Cert
  Dispose = $true
}
Add-AdCertificate @AAParams | Format-List
```

```
Certificates : {My Root CA}
DsPath       : CN=Certification Authorities,CN=Public Key Services,
               CN=Services,CN=Configuration,DC=ad,DC=example,DC=net
ContainerType: RootCA
IsModified   : False
```

To learn more about certificate containers, please see this very well-written article[28] by Vadims Podāns.

15.5.1.2.7 DSC

Desired State Configuration is a perfect method to manage system configuration on non-domain computers. It also complements GPO-style management well for domain-joined machines! With DSC, you can control almost any parameter of your environment because deep inside it are regular PowerShell cmdlets and functions. It means that, of course, you can install certificates with DSC. A “CertificateDsc” module[29] from the DCS Community[30] has a CertificateImport resource and ensures the presence (or absence) of a certificate in a store.

If you haven’t worked with Desired State Configuration before, we have just the chapter for you! Please refer to Infrastructure as Code and come back to this once you’re done.

Preparing a configuration for certificate importing is very easy:

1. Place the certificate file on an SMB share available to all computers to which you want to deploy that certificate.
2. Configure a CertificateImport resource, as described in the example below.
3. Deploy the configuration to your nodes!

[28] https://www.pkisolutions.com/understanding-active-directory-certificate-services-containers-in-active-directory/
[29] https://github.com/dsccommunity/CertificateDsc/
[30] https://dsccommunity.org/

Example 67: A DSC resource for importing a root CA certificate from an SMB share

```
Configuration DeployMyRootCertificate
{
    Import-DscResource -ModuleName CertificateDsc

    Node localhost
    {
        CertificateImport MyRootCertificate
        {
            Thumbprint = '0000000000000000000000000000000000000000'
            Location   = 'LocalMachine'
            Store      = 'Root'
            Path       = '\\ad.example.net\NETLOGON\ROOTCA01_My Root CA.crt'
        }
    }
}
```

Now, there are several considerations to be taken into account:

1. The `Thumbprint` parameter of the configuration isn't used when importing a new certificate—you need it only when you want to remove a certificate from the machine. However, the parameter is marked as mandatory, which means you must fill it with a string that looks like a certificate thumbprint (there's a check for that).
2. The `Path` parameter in this example points to a standard NETLOGON share. This is just an example and might not be the best method for your infrastructure. You might use DSC to manage non-domain machines which don't have access to the NETLOGON folder. You should also never distribute large files this way!

15.5.1.2.8 Intune

For workstations connected to Intune, you can use trusted certificate profiles[31] to import root certificates.

15.5.1.2.9 Issue a Signing Certificate

As the result of all this private PKI adventure, you can now issue yourself a new certificate:

Example 68: Inspecting a code signing certificate from the current user's personal store

```
$Params = @{
  Template = 'MyCodeSigning'
  CertStoreLocation = 'Cert:\CurrentUser\My'
}
Get-Certificate @Params
```

[31] https://learn.microsoft.com/en-us/mem/intune/protect/certificates-trusted-root

```
Status Certificate
------ -----------
Issued [Subject]...
```

Hmm… Not much info in the output, so let's take a closer look at it:

Example 69: Retrieving a list of certificates in the current user's personal store

```
$Cert = Get-ChildItem -Path 'Cert:\CurrentUser\My'
$Cert | Format-List
```

```
Thumbprint: BC85FF727E7DA52FC34AD92B75B7FE031EF908DA
Subject   : CN=Administrator, CN=Users, DC=ad, DC=example, DC=net
```

Example 70: Inspecting the code signing certificate in more detail

```
$Path = Join-Path -Path 'Cert:\CurrentUser\My' -ChildPath $Cert.Thumbprint
Get-Item -Path $Path | Select-Object -Property *
```

```
PSPath                   : Microsoft.PowerShell.Security\Certificate::CurrentUs
                           er\My\BC85FF727E7DA52FC34AD92B75B7FE031EF908DA
PSParentPath             : Microsoft.PowerShell.Security\Certificate::CurrentUs
                           er\My
PSChildName              : BC85FF727E7DA52FC34AD92B75B7FE031EF908DA
PSDrive                  : Cert
PSProvider               : Microsoft.PowerShell.Security\Certificate
PSIsContainer            : False
EnhancedKeyUsageList     : {Code Signing (1.3.6.1.5.5.7.3.3)}
DnsNameList              : {Administrator}
SendAsTrustedIssuer      : False
EnrollmentPolicyEndPoint : Microsoft.CertificateServices.Commands.EnrollmentEnd
                           PointProperty
EnrollmentServerEndPoint : Microsoft.CertificateServices.Commands.EnrollmentEnd
                           PointProperty
PolicyId                 : {360BD38F-E9BE-4724-86F3-65CD14FF86C9}
Archived                 : False
Extensions               : {System.Security.Cryptography.Oid...}
FriendlyName             :
IssuerName               : System.Security.Cryptography.X509Certificates.X500Di
                           stinguishedName
NotAfter                 : 9/27/2022 9:35:21 PM
NotBefore                : 9/27/2021 9:35:21 PM
HasPrivateKey            : True
PrivateKey               :
PublicKey                : System.Security.Cryptography.X509Certificates.Public
                           Key
RawData                  : {48, 130, 3, 106...}
SerialNumber             : 3A000000052570DD73698499F7000000000005
SubjectName              : System.Security.Cryptography.X509Certificates.X500Di
                           stinguishedName
SignatureAlgorithm       : System.Security.Cryptography.Oid
Thumbprint               : BC85FF727E7DA52FC34AD92B75B7FE031EF908DA
Version                  : 3
Handle                   : 2611199710288
Issuer                   : CN=My Issuing CA, DC=ad, DC=example, DC=net
Subject                  : CN=Administrator, CN=Users, DC=ad, DC=example, DC=net
```

Yep—that seems about right! Now you can use it to sign code as described in the Signing Process paragraph above.

15.5.2 Deploy Code Signing Certificates in a Corporate Environment

After you've gotten a code signing certificate and used it to sign some scripts, you might want to distribute these scripts across your organization. However, your peers won't be able to trust your signature without having your code signing certificate correctly installed on their machines.

We already know how to install it manually. Let's look at how to distribute code signing certificates at a scale.

15.5.2.1 GPO

Unfortunately, there are no AD DS containers to hold Code Signing certificates. One of the possible alternatives for this task is Group Policies. In GPOs, certificates are stored as registry keys. And that's how they get onto target computers too—as registry keys. Therefore, in order to import a certificate into a group policy object, import it as a registry key, with the help of `Set-GPRegistryValue`. The registry keys that store certificate objects are binary, which means:

1. They aren't human-readable.
2. They aren't easy to operate.

A nice touch is that the structure of these binary keys isn't documented.

⚠ Before you proceed any further, understand that this is an **unofficial** way to import certificates into a GPO—it might stop working at any moment. Unfortunately, in 2021 there is still no official command-line method to do this—Microsoft wants you to import certificates into group policies through the MMC console[32]. If you're fine with that, by all means, go ahead and use the GUI instead of this unsupported command-line method.

ℹ Note that various third parties do have software that provides this support. This includes SDM Software[33] and their `Group Policy Automation Engine`. Also please note that pushing Certificates via Group Policy using Microsoft's `Group Policy Management Console` is not difficult and should be considered if you only have a few certificates that need to be distributed.

Working with undocumented binary structures is always a challenge, but observe that a certificate imported manually into the local machine store and the same certificate imported through Group Policies yield identical registry values. That means that we can just get the value from the local registry and set that value into a group policy object!

ℹ In the code below, `$GPOName` is the name of a Group Policy Object you want to use to deploy the certificate and this object is already linked to a desired OU.

First, you need to get the thumbprint of your certificate for later use:

[32] https://learn.microsoft.com/en-us/windows-server/identity/ad-fs/deployment/distribute-certificates-to-client-computers-by-using-group-policy

[33] https://www.sdmsoftware.com

Example 71: Retrieving the thumbprint for a code signing certificate

```
$Certificate = Get-ChildItem -Path 'Cert:\CurrentUser\My' -CodeSigningCert
$Thumbprint = $Certificate.Thumbprint
```

Note that the above only works properly if you have a single code signing certificate stored for your user Otherwise you may need to view the Subject, NotBefore, and NotAfter fields to identify the proper certificate.

Next, *temporarily* export the certificate into a file (only the public part) and import it back, but into the local machine store this time. You must do this because personal user certificates reside on the file system rather than in the registry. You can't access them through `HKEY_CURRENT_USER` or any other registry hive.

You can find files for your personal certificates in `%APPDATA%\Microsoft\SystemCertificates\My\Certificates`.

Example 72: Exporting the code signing certificate and reimporting it into the machine store

```
$ExportPath = Join-Path -Path $env:TEMP -ChildPath ([guid]::NewGuid().Guid)
Export-Certificate -Cert $Certificate -FilePath $ExportPath
$Params = @{
  FilePath = $ExportPath
  CertStoreLocation = 'Cert:\LocalMachine\Root\'
}
Import-Certificate @Params
Remove-Item -Path $ExportPath
```

Importing certificates into the machine store requires local administrator permission on a computer.

The example doesn't use the `X509Store` .NET class here to copy certificates between the stores because it encodes certificate objects differently than the GUI. It still works, but just to be safe, you'll want to mimic the GUI as closely as possible. `Import-Certificate` helps with that, producing results identical to the GUI.

You can now get the certificate object from the local registry and add it into the GPO registry:

Example 73: Extracting the registry data for the certificate and creating a GPO registry value from it

```
$JPParams = @{
  Path = 'HKLM:\SOFTWARE\Microsoft\SystemCertificates\ROOT\Certificates'
  ChildPath = $Thumbprint
}
$Path = Join-Path @JPParams
$CertBlob = Get-ItemPropertyValue -Path $Path -Name 'Blob'
$GPOCertContainerPath = 'HKLM\SOFTWARE\Policies\Microsoft\SystemCertificates'
$Key = "$GPOCertContainerPath\TrustedPublisher\Certificates\$Thumbprint"
$SGParams = @{
  Name = $GPOName
  Key = $Key
  ValueName = 'Blob'
  Value = $CertBlob
  Type = 'Binary'
}
Set-GPRegistryValue @SGParams
```

And a quick clean-up:

Example 74: Removing the certificate that was temporarily added to the machine store

```
Remove-Item -Path "Cert:\LocalMachine\Root\$Thumbprint"
```

The certificate has been added to the GPO and will be delivered to your computers soon.

15.5.2.2 DSC (Signing Certificate)

The CertificateImport resource from the CertificateDsc module allows you to import certificates not only into the Root store but into virtually any of them![34] Here's an example of a configuration for code-signing certificates:

Example 75: A DSC resource for importing a code signing certificate from an SMB share

```
Configuration DeployMySigningCertificate
{
    Import-DscResource -ModuleName CertificateDsc

    Node localhost
    {
        CertificateImport MySigningCertificate
        {
            Thumbprint = '0000000000000000000000000000000000000000'
            Location   = 'LocalMachine'
            Store      = 'TrustedPublisher'
            Path       = '\\ad.example.net\NETLOGON\Administrator.crt'
        }
    }
}
```

This is much easier than with Group Policies and has no issues with support!

[34] DSC Community. (2021, Feb. 26). *Welcome to the CertificateDsc wiki*. dsccommunity/CertificateDsc on GitHub. [Online]. Available: https://github.com/dsccommunity/CertificateDsc/wiki. [Accessed: Sep. 15, 2022].

15.5.2.3 Intune (Signing Certificate)

Intune doesn't have native functionality to deploy certificates to the Trusted Publishers store. However, the Intune Support team described how you can utilize custom configuration profiles[35] for this.

15.6 Summary

In this chapter, you have learned why you need to sign your code, how to do it, and how to implement your own Public Key Infrastructure to sign an unlimited number of scripts for free. In the next chapter, Script Execution Policies, you'll see how you can use this knowledge to protect your infrastructure even further, by utilizing PowerShell Execution Policies.

15.7 Further Reading

- Official Microsoft script signing docs[36]
- Vadims Podāns's blog on PKI[37]
- More PKI-related goodies by Vadims Podāns[38]
- Wikipedia: Public-key cryptography[39]
- Wikipedia: Code signing[40]

[35] https://techcommunity.microsoft.com/t5/intune-customer-success/adding-a-certificate-to-trusted-publishers-using-intune/ba-p/1974488

[36] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_signing

[37] https://www.sysadmins.lv/blog-en/categoryview/securitypki.aspx

[38] https://www.pkisolutions.com/author/crypt32/

[39] https://en.wikipedia.org/wiki/Public-key_cryptography

[40] https://en.wikipedia.org/wiki/Code_signing

16. Script Execution Policies

Script execution policies dictate the conditions under which PowerShell runs scripts and loads configuration files and PowerShell modules. This chapter discusses in-depth how script execution policies can be used and implemented. It will also give some use cases for each policy.

16.1 Types of Execution Policies

You can view the current active execution policy with the cmdlet `Get-ExecutionPolicy`.

There are seven types of execution policies:[1]

- `AllSigned`
- `RemoteSigned`
- `Restricted`
- `Unrestricted`
- `Bypass`
- `Default`
- `Undefined`

Each policy is covered in detail in this section.

16.1.1 AllSigned

The `AllSigned` execution policy allows scripts to run, but requires all scripts and configuration files to be signed by a trusted publisher, even those scripts, which you wrote yourself.

Signing a script involves using cryptographic signatures to provide information about the integrity and authenticity of the file. The signature is checked at the time the script is run to ensure that the contents of the script haven't changed since signing and that the certificate used to sign the script originates from a trusted source.

The intricacies of script signing are covered in the previous chapter, Script Signing.

The AllSigned policy also causes the shell to prompt you before running scripts from publishers that you haven't yet classified as trusted or untrusted. To designate a publisher as trusted, the certificate that was used to sign the script must be in the Trusted Publishers certificate store, and its root certificate must be in the Trusted Root Certification Authorities store. Documentation on how to import a certificate is available on Microsoft Docs[2].

[1] Microsoft. (2022, Apr. 20). *System.Management.Automation - SecuritySupport.cs*. L32-66. PowerShell/PowerShell on GitHub. [Online]. Available: https://github.com/PowerShell/PowerShell/blob/master/src/System.Management.Automation/security/SecuritySupport.cs. [Accessed: Aug. 03, 2022].

[2] https://learn.microsoft.com/en-us/windows-hardware/drivers/install/trusted-publishers-certificate-store

It's important to note that this policy isn't a failsafe against malicious scripts, as the system still permits a signed script to run, regardless of content. It's not unheard of for disgruntled employees to write innocuous-looking scripts designed to cause catastrophic damage to the environment. These malicious scripts are called "logic bombs" and are a challenging security threat. Since they rarely match any known malware signatures, they can avoid detection by antivirus and antimalware software.

16.1.2 RemoteSigned

`RemoteSigned` is the default execution policy for Windows Server operating systems.[3]

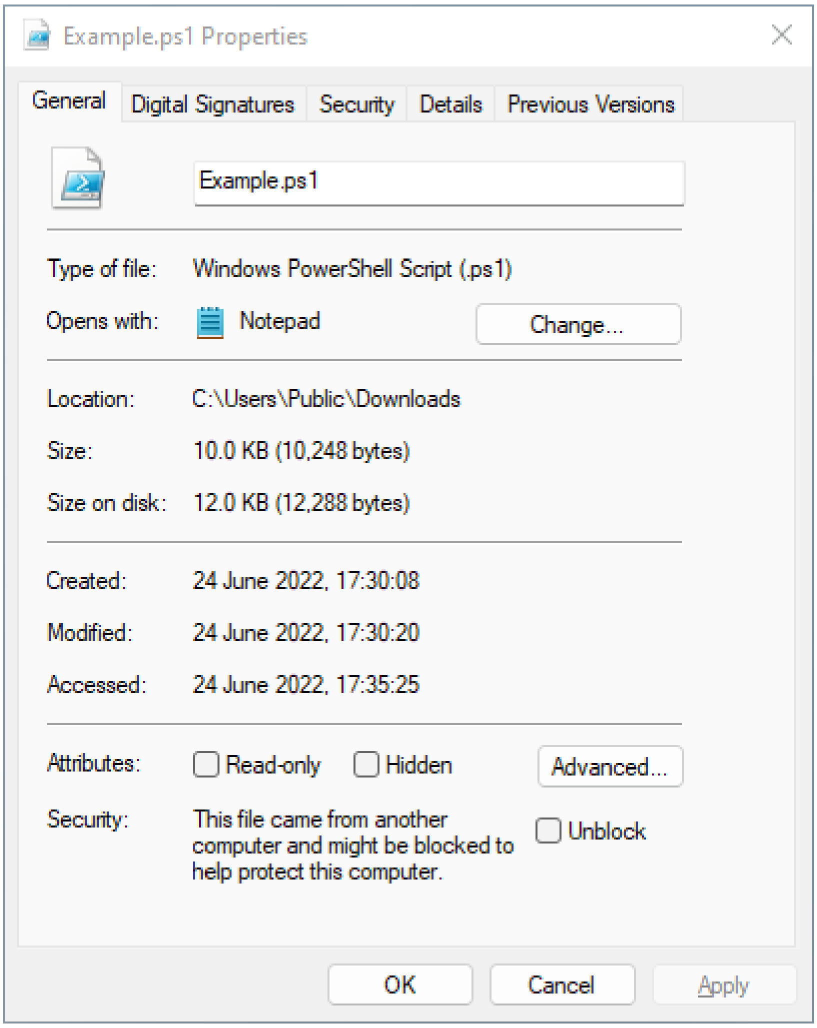
It allows PowerShell scripts to run, but requires that PowerShell scripts and files downloaded from the internet be signed by a trusted publisher. This includes scripts attached to emails or downloaded from instant messaging apps.

`RemoteSigned` differs significantly from `AllSigned` in that it allows unsigned scripts on the local computer to be run, and will permit running scripts downloaded from the internet provided they're not blocked. Files are blocked by Windows when the OS detects that they originate from the internet zone (zone 3). PowerShell's `RemoteSigned` policy blocks scripts and modules from running if they aren't signed by a trusted publisher. The behavior differs from `AllSigned` when an unsigned file becomes unblocked. A file can be unblocked by anyone with write access to the file.

Blocked files are files that originate from outside of the local intranet zone, files downloaded from the internet will be blocked by default. There are two ways to unblock a file:

The first is to use the Windows Graphical User Interface (GUI) by right-clicking the file and selecting 'Properties'. In the 'General' tab, there's a checkbox labeled 'Unblock'.

[3] Microsoft. (2022, Mar. 18). *About Execution Policies (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_execution_policies. [Accessed: Aug. 03, 2022].



Example.ps1

Type of file: Windows PowerShell Script (.ps1)

Opens with: Notepad

Location: C:\Users\Public\Downloads

Size: 10.0 KB (10,248 bytes)

Size on disk: 12.0 KB (12,288 bytes)

Created: 24 June 2022, 17:30:08

Modified: 24 June 2022, 17:30:20

Accessed: 24 June 2022, 17:35:25

Attributes: □ Read-only □ Hidden

Security: This file came from another computer and might be blocked to help protect this computer. □ Unblock

Unblock File option in Windows Explorer

The second method is to use the `Unblock-File` cmdlet, as shown here:

Example 1: Unblocking a file from outside of the intranet zone

```
Unblock-File -Path <Path to file>
```

`RemoteSigned` is useful in cases where a user is an administrator or power user whose job involves creating PowerShell scripts, as it allows running local scripts but still has some measures in place to prevent running potentially harmful scripts downloaded from the internet.

16.1.3 Restricted

The `Restricted` execution policy is the default execution policy for Windows client computers.[4]

[4] Microsoft. (2022, Mar. 18). *About Execution Policies (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_execution_policies. [Accessed: Aug. 03, 2022].

It permits running individual commands but doesn't allow any scripts, configuration files, module files, or PowerShell profiles to run, even when they're signed. `Restricted` is, as its name indicates, the most restrictive execution policy, and is most often used in high-security environments, or on machines where PowerShell isn't used for any processes.

16.1.4 Unrestricted

`Unrestricted` is the default policy for non-Windows systems, and can't be changed.[5]

This policy allows unsigned scripts to run, but still warns the user before running scripts or configuration files that don't originate from the Local intranet zone.

`Unrestricted` execution policies should be avoided wherever possible. If a script file is known to be safe or is part of a known process, confirm that the script isn't being blocked, and that the system it runs on has an execution policy of `RemoteSigned` or higher.

16.1.5 Bypass

Despite its name, the `Bypass` mode is even more permissive than the `Unrestricted` mode. In `Bypass` mode, all scripts can run and there are no warnings nor prompts. This execution policy isn't one you are likely to encounter in most well-managed environments. It's designed for scenarios in which a PowerShell script is part of a larger application, or where PowerShell itself is the foundation for an application that has its own security model and controls. `Bypass` is the least restrictive execution policy.

16.1.6 Default

The `Default` execution policy is a bit different from the others. `Default` can be used as an input for the `Set-ExecutionPolicy` cmdlet, but it's not a valid output from `Get-ExecutionPolicy`. The use case for `Default` lies in the `Set-ExecutionPolicy` cmdlet:

Example 2: Setting the execution policy to its default for that platform

```
Set-ExecutionPolicy -ExecutionPolicy Default
```

This command sets the execution policy based on the type of operating system you are working with. For Windows client computers, the default policy is `Restricted`. For Windows servers, the default policy is `RemoteSigned`.

16.1.7 Undefined

The `Undefined` policy means that there is no execution policy set within the current scope. Without an execution policy set, the effective execution policy is the default for its respective operating system, as outlined in the `Default` execution policy.

[5] Microsoft. (2022, Mar. 18). *About Execution Policies (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_execution_policies. [Accessed: Aug. 03, 2022].

Example 3: Unsetting the execution policy by setting it *Undefined*

```
Set-ExecutionPolicy -ExecutionPolicy Undefined
```

Running this command causes any existing execution policy to be unassigned, unless it's determined by Group Policy. If the execution policy in all scopes is `Undefined`, the effective execution policy is `Restricted`.[6]

16.2 Execution Policy Scope

The scope of an execution policy indicates the broadness of its impact. There are five different scopes listed here in order of their precedence.[7] A scope higher on the list takes precedence over a scope lower on the list. The execution policy on the scope with the highest precedence is known as the effective execution policy.[8]

1. `MachinePolicy`
2. `UserPolicy`
3. `Process`
4. `CurrentUser`
5. `LocalMachine`

16.2.1 Scope Precedence

Unlike NTFS (NT File System) permissions, with which you may be familiar, the effective execution policy is dictated solely by the order of precedence, with no regard to the restrictiveness of the policy. For example, if the execution policy in `MachinePolicy` is `Unrestricted`, but the policy in `CurrentUser` is `Restricted`, the effective execution policy for the user is `Unrestricted`. Alternatively, if the execution policy in `LocalMachine` is `RemoteSigned`, and the execution policy in `CurrentUser` is `Restricted`, the effective execution policy is `Restricted`.

16.2.1.1 MachinePolicy

The `MachinePolicy` scope applies to all users on a machine. Users aren't able to apply execution policies to this scope; it can only be set using Group Policy. Setting execution policies using Group Policy is covered in the AppLocker section of this chapter.

The machine policy for Windows PowerShell is stored in the registry at the following location:

[6] Microsoft. (2022, Mar. 08). *Set-ExecutionPolicy (Microsoft.PowerShell.Security)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.security/set-executionpolicy. [Accessed: Aug. 03, 2022].

[7] Microsoft. (2022, Mar. 18). *About Execution Policies (Microsoft.PowerShell.Core) - Execution policy scope*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_execution_policies#execution-policy-scope. [Accessed: Aug. 03, 2022].

[8] Microsoft. (2022, Mar. 08). *Set-ExecutionPolicy (Microsoft.PowerShell.Security) - Parameters*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.security/set-executionpolicy#parameters. [Accessed: Aug. 03, 2022].

```
HKLM\Software\Policies\Microsoft\Windows\PowerShell
```

The machine polcy for PowerShell 7.0 and later is in the following location:

```
HKLM\SOFTWARE\Policies\Microsoft\PowerShellCore
```

For both locations, `HKLM` is the `HKEY_LOCAL_MACHINE` hive.

16.2.1.2 UserPolicy

`UserPolicy` applies to the current user of the computer. Like `MachinePolicy`, the execution policy for this scope is defined by Group Policy.

The user policy for Windows PowerShell is stored in the registry at the following location:

```
HKCU\Software\Policies\Microsoft\Windows\PowerShell
```

The user policy for PowerShell 7.0 and later is in the following location:

```
HKCU\SOFTWARE\Policies\Microsoft\PowerShellCore
```

For both locations, `HKCU` is the `HKEY_CURRENT_USER` hive.

16.2.1.3 Process

Execution policies with the `Process` scope apply to the current PowerShell process. The policy is stored in the following PowerShell session environmental variable until the session is closed, at which point it's deleted:

```
$ENV:PSExecutionPolicyPreference
```

16.2.1.4 CurrentUser

Policies with the `CurrentUser` scope apply to the current user of a machine, similar to `UserPolicy`. The key difference is that permitted users can set the execution policy in the `CurrentUser` scope using the `Set-ExecutionPolicy` cmdlet. When an execution policy has this scope in Windows PowerShell, it's stored in the registry in the `HKEY_CURRENT_USER` hive at the following location:

```
HKCU\SOFTWARE\Microsoft\PowerShell\1\ShellIds\Microsoft.PowerShell
```

In PowerShell 7.0 and later, it's stored in the `powershell.config.json` configuration file in the user's Documents directory.[9]

[9] Microsoft. (2020, Nov. 27). *System.Management.Automation - PSConfiguration.cs*. L33-L66. PowerShell/PowerShell on GitHub. [Online]. Available: https://github.com/PowerShell/PowerShell/blob/master/src/System.Management.Automation/engine/PSConfiguration.cs. [Accessed: Aug. 03, 2022].

16.2.1.5 LocalMachine

An execution policy in the `LocalMachine` scope applies to all users of the machine, just like `MachinePolicy`. Once again, users are able to change the execution policy of the scope using `Set-ExecutionPolicy`, provided they have the permissions to do so. When an execution policy has this scope in Windows PowerShell, it's stored in the registry in the `HKEY_LOCAL_MACHINE` hive at the following location:

```
HKLM\SOFTWARE\Microsoft\PowerShell\1\ShellIds\Microsoft.PowerShell
```

In PowerShell 7.0 and later, it's stored in the `powershell.config.json` configuration file in the PowerShell installation directory.[10]

16.3 Security Considerations

It's worth noting that an execution policy on its own isn't a complete defense against threats. It may help to prevent users from inadvertently running untrusted scripts, but determined attackers can bypass the defined execution policy, even if it's set restrictively.

For further information, you can read this blog post[11] by Scott Sutherland on ways to bypass an execution policy.

16.4 Setting the Execution Policy

Governing the execution policy can be handled in several ways, or even a combination thereof:

- `Set-ExecutionPolicy` cmdlet[12]
- Group Policy[13]
- Registry[14] [15]
- AppLocker[16]
- Windows Defender Application Control[17]

[10] Microsoft. (2020, Nov. 27). *System.Management.Automation - PSConfiguration.cs.* L33-L66. PowerShell/PowerShell on GitHub. [Online]. Available: https://github.com/PowerShell/PowerShell/blob/master/src/System.Management.Automation/engine/PSConfiguration.cs. [Accessed: Aug. 03, 2022].

[11] https://www.netspi.com/blog/technical/network-penetration-testing/15-ways-to-bypass-the-powershell-execution-policy/

[12] Microsoft. (2022, Mar. 08). *Set-ExecutionPolicy (Microsoft.PowerShell.Security).* Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.security/set-executionpolicy. [Accessed: Aug. 03, 2022].

[13] Microsoft. (2022, Mar. 18). *About Group Policy Settings (Microsoft.PowerShell.Core).* Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_group_policy_settings. [Accessed: Aug. 03, 2022].

[14] Microsoft. (2022, Mar. 08). *Set-ExecutionPolicy (Microsoft.PowerShell.Security).* Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.security/set-executionpolicy. [Accessed: Aug. 03, 2022].

[15] Microsoft. (2020, Nov. 27). *System.Management.Automation - PSConfiguration.cs.* L33-L66. PowerShell/PowerShell on GitHub. [Online]. Available: https://github.com/PowerShell/PowerShell/blob/master/src/System.Management.Automation/engine/PSConfiguration.cs. [Accessed: Aug. 03, 2022].

[16] Microsoft. (2017, Sep. 21). *Script rules in AppLocker.* Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/applocker/script-rules-in-applocker. [Accessed: Oct. 12, 2021].

[17] Microsoft. (2022, Apr. 25). *Understand Windows Defender Application Control (WDAC) policy rules and file rules.* Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/select-types-of-rules-to-create. [Accessed: Aug. 03, 2022].

16.4.1 Set-ExecutionPolicy

To change the execution policy for a scope, use the `Set-ExecutionPolicy` cmdlet:

Example 4: Setting the execution policy for a scope

```
Set-ExecutionPolicy -ExecutionPolicy <ExecutionPolicy> -Scope <Scope>
```

This cmdlet only adjusts the execution policy in the process, current user, or local machine scope. You must use Group Policy to set the `MachinePolicy` or `UserPolicy`. As explained in the Scope Precedence section, execution policies defined by Group Policy Objects override the configuration changes made using the `Set-ExecutionPolicy` cmdlet.

16.4.2 Group Policy

Using Group Policy, you can set the execution policy on machines in your environment *en masse*. You can add the required registry value to a Group Policy Object (GPO) using PowerShell:

The following example shows the registry key for Windows PowerShell's execution policy. For PowerShell 7.0 and later, read ahead.

Example 5: Adding PowerShell execution policies to Group Policy Objects

```
$GPOName = '<Name of your Group Policy Object>'
$RegKey = 'HKLM\Software\Policies\Microsoft\Windows\PowerShell'

$SetGPOParams = @{
    Name      = $GPOName
    Key       = $RegKey
    ValueName = 'EnableScripts'
    Value     = 1
    Type      = 'DWORD'
}
Set-GPRegistryValue @SetGPOParams

$SetGPOParams.ValueName = 'ExecutionPolicy'
$SetGPOParams.Value     = 'AllSigned'
$SetGPOParams.Type      = 'String'
Set-GPRegistryValue @SetGPOParams
```

The registry value in the example correlates with the settings from the Windows PowerShell Group Policy template:[18]

[18] Microsoft. (2022, Mar. 18). *About Group Policy Settings (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_group_policy_settings. [Accessed: Aug. 03, 2022].

```
Computer Configuration\
  Administrative Templates\
    Windows Components\
      Windows PowerShell
```

The same registry path and Group Policy configuration path can be used under `HKEY_CURRENT_USER` and `User Configuration`, respectively.

It's also possible to manage the execution policy on machines using only registry changes, however this isn't advisable. Settings defined in Group Policy override registry keys if they conflict. The registry keys themselves, on the other hand, can be changed by anyone with write access to the registry. If the registry keys are changed, the changes persist until the next Group Policy refresh. By default, Group Policy is refreshed on a computer every 90 minutes, with a random offset of 30 minutes. This method can be useful if an administrator needs to make the execution policy more lax temporarily, but it lacks utility outside of ad hoc cases.

PowerShell 7.0 and later uses different Group Policy settings and registry keys.[19] It comes with additional Group Policy templates that you can install by running the following script from the PowerShell installation directory:

Example 6: Adding PowerShell Group Policy definitions

```
InstallPSCorePolicyDefinitions.ps1
```

This provides new configuration paths in the Group Policy editor:

```
Computer Configuration\
  Administrative Templates\
    PowerShell Core

User Configuration\
  Administrative Templates\
    PowerShell Core
```

These contain the same configuration options as with Windows PowerShell, and set values in the following registry keys:

```
HKLM\SOFTWARE\Policies\Microsoft\PowerShellCore

HKCU\SOFTWARE\Policies\Microsoft\PowerShellCore
```

In the above locations, `HKLM` is the `HKEY_LOCAL_MACHINE` hive and `HKCU` is the `HKEY_CURRENT_USER` hive.

[19] Microsoft. (2022, Mar. 18). *About Group Policy Settings (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_group_policy_settings. [Accessed: Aug. 03, 2022].

16.4.3 AppLocker

For a more robust defense against threat actors, you can use AppLocker to restrict a user's ability to execute scripts. Using AppLocker, you can prevent scripts from running based on a wide array of criteria, including (but not limited to):[20]

- The identity of the user attempting to run a script
- The directory the script exists in
- Whether the script is signed by a trusted authority

It's a good idea to leave the default rules in place, as not all .ps1 files shipped with Windows are signed. The AppLocker service, AppIDSvc, doesn't run by default. You have to start it on each computer you want to protect, and sometimes you need to sign out and back in before AppLocker rules will apply to a session. You can use Group Policy Preferences or Group Policy Policies inside of a GPO to set "Automatic" as the service start type.

A basic use case for AppLocker and Execution policies can be illustrated using the following example: you want to prevent all users from running scripts that aren't signed by Microsoft.

Note: to specify a trusted or untrusted publisher, you must have a .ps1 file signed by the publisher to use as a reference file.

To do this, you would create a GPO with the following settings:

- `Computer Policies\Windows Settings\Security Settings\Application Control Policies\AppLocker\Script Rules`
  - **Permissions**: Deny: `Everyone`
  - **Conditions**: File Path: `*`
  - **Exceptions**: Publisher: `Microsoft`
- `Computer Policies\Administrative Templates\Windows Components\Windows PowerShell`
  - **Turn on Script Execution**: Enabled: `Allow only signed scripts`
- `Computer Policies\Windows Settings\System Services`
  - **Application Identity**: Define this policy setting: `Automatic`

This configuration would set the `MachinePolicy` on computers affected by the GPO to `AllSigned`, which would prevent running any scripts unless they're signed by a trusted publisher.

You can read more about AppLocker in the Constrained Language Mode chapter.

16.4.4 Windows Defender Application Control

Introduced with Windows 10, Windows Defender Application Control (WDAC) offers even more control over machine-level applications.[21]

[20] Microsoft. (2022, Jul. 25). *AppLocker (Windows)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/applocker/applocker-overview. [Accessed: Aug. 04, 2022].

[21] Microsoft. (2022, Jul. 25). *Windows Defender Application Control and AppLocker Overview*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/wdac-and-applocker-overview. [Accessed: Aug. 04, 2022].

WDAC policies apply to the computer as a whole, and unlike AppLocker, there's no option to apply a policy only to certain users. WDAC is also currently being actively developed by Microsoft, while AppLocker only receives security fixes. Microsoft recommends using WDAC policies rather than AppLocker if possible, but this decision depends highly on your environment.

It's possible to deploy both, and use AppLocker as a complement to WDAC. For instance, using WDAC for machine baselines while deploying AppLocker policies for more granular control on a user level.

You can enable code integrity rule options using the `Set-RuleOption` cmdlet. For example, to restrict both kernel-mode and user-mode binaries, you can use `-Option 0`, also known as `Enabled:UMCI`.[22]

Example 7: Restricting user-mode binaries in addition to kernel-mode ones with WDAC

```
Set-RuleOption -FilePath '<Path to policy XML>' -Option 0
```

To enable PowerShell script enforcement, use `-Option 11` with the `-Delete` parameter, also known as `Disabled: Script Enforcement`.

Example 8: Enabling PowerShell script enforcement with WDAC

```
Set-RuleOption -FilePath '<Path to policy XML>' -Option 11 -Delete
```

A full list of WDAC policy rule options is available on Microsoft Docs[23]. You can read more about WDAC in the Constrained Language Mode chapter.

16.5 Further Reading

- About Execution Policies—Microsoft Docs[24]
- Get-ExecutionPolicy cmdlet—Microsoft Docs[25]
- Set-ExecutionPolicy cmdlet—Microsoft Docs[26]
- About Group Policy Settings—Microsoft Docs[27]
- Trusted Publishers Certificate Store—Microsoft Docs[28]
- Learn more about PowerShell script security—Microsoft Docs[29]
- Script rules in AppLocker—Microsoft Docs[30]

[22] Microsoft. (2022, Apr. 25). *Understand Windows Defender Application Control (WDAC) policy rules and file rules.* Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/select-types-of-rules-to-create. [Accessed: Aug. 03, 2022].

[23] https://learn.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/select-types-of-rules-to-create#windows-defender-application-control-policy-rules

[24] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_execution_policies

[25] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.security/get-executionpolicy

[26] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.security/set-executionpolicy

[27] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_group_policy_settings

[28] https://learn.microsoft.com/en-us/windows-hardware/drivers/install/trusted-publishers-certificate-store

[29] https://learn.microsoft.com/en-us/mem/configmgr/apps/deploy-use/learn-script-security

[30] https://learn.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/applocker/script-rules-in-applocker

- AppLocker Overview—Microsoft Docs[31]
- WDAC Overview—Microsoft Docs[32]
- 15 Ways to Bypass the PowerShell Execution Policy—NetSPI Blogs[33]

[31] https://learn.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/applocker/applocker-overview

[32] https://learn.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/wdac-and-applocker-overview

[33] https://www.netspi.com/blog/technical/network-penetration-testing/15-ways-to-bypass-the-powershell-execution-policy/

17. Constrained Language Mode

Constrained Language Mode (CLM) is a PowerShell Language Mode used by the PowerShell Remoting Session Configuration and the PowerShell Default (console) Runspace. Constrained Mode provides an out-of-the-box solution to console-based security by restricting the scope of non-approved scripts, reducing the risk of malicious code execution within a console session. This chapter explores several topics:

- What Constrained Language Mode is
- How it works
- Its limitations
- How it fits within Microsoft's Secure Boot platform

Finally, the chapter describes how to implement CLM with AppLocker and Windows Defender Application Control (WDAC).

17.1 In Depth

What is Constrained Language Mode?

First introduced in PowerShell 3.0, Constrained Language Mode was initially a mechanism for Windows Defender Application Control (WDAC), formerly User-mode Code Integrity (UMCI), to manage PowerShell's default Runspace (console) on Windows 8.1 RT. Following Windows 8.1, subsequent PowerShell versions included CLM as a means to provide console-based security.

WDAC/AppLocker is a top-level security model in the Windows Shell within Microsoft's Trusted Boot Process,[1] in which Windows uses these to secure the top-level 'Shell' of Windows. Once a system enforces any WDAC/AppLocker script rules, the PowerShell (including PowerShell Core) console session will start in Constrained Language Mode.

17.1.1 Language Modes

PowerShell has four language modes that define the security state on the PowerShell session. These are:

- **FullLanguage**: The default mode. This mode permits all language elements.
- **ConstrainedLanguage**: Allows all language elements but limits permitted types.
- **RestrictiveLanguage**: Restricted use of language elements, limited variable usage.
- **NoLanguage**: No language elements are available.

To view the language mode of the current session, use: '$ExecutionContext.SessionState.LanguageMode'.

[1] Microsoft. (2021, Jan. 29). *Windows 10 Device Guard and Credential Guard Demystified.* Microsoft TechCommunity. [Online]. Available: https://techcommunity.microsoft.com/t5/iis-support-blog/windows-10-device-guard-and-credential-guard-demystified/ba-p/376419. [Accessed: Oct. 12, 2021].

Example 1: Displaying the current language mode

```
$ExecutionContext.SessionState.LanguageMode
```

```
FullLanguage
```

The *LanguageMode* property is settable only in the `FullLanguage` Language Mode. You can use this to test scripts for potential issues that may arise. The example below demonstrates changing the LanguageMode from `FullLanguage` to `ConstrainedLanguage`, and back again. Note that the console throws an error when attempting to reset the property.

Example 2: Changing the language mode in a session isn't reversible

```
$ExecutionContext.SessionState.LanguageMode

$ExecutionContext.SessionState.LanguageMode = "ConstrainedLanguage"
$ExecutionContext.SessionState.LanguageMode

$ExecutionContext.SessionState.LanguageMode = "FullLanguage"
```

```
FullLanguage

ConstrainedLanguage

InvalidOperation: Cannot set property. Property setting is supported only on
core types in this language mode.
```

You should only use the Session State for debugging or testing since it's not enforceable.

17.1.2 Constrained Language Mode Features

CLM has several security features that limit the scope and impact of the PowerShell console session. These features are:

17.1.2.1 Allowed Types

Constrained Language Mode limits the use of types within the session. It permits the following types:

Types A-N	Types O-Z
• `AliasAttribute`	• `OutputTypeAttribute`
• `AllowEmptyCollectionAttribute`	• `ParameterAttribute`
• `AllowEmptyStringAttribute`	• `PSCredential`
• `AllowNullAttribute`	• `PSDefaultValueAttribute`
• `Array`	• `PSListModifier`
• `Bool`	• `PSObject`
• `byte`	• `PSPrimitiveDictionary`
• `char`	• `PSReference`
• `CmdletBindingAttribute`	• `PSTypeNameAttribute`
• `DateTime`	• `Regex`
• `decimal`	• `SByte`
• `DirectoryEntry`	• `string`
• `DirectorySearcher`	• `SupportsWildcardsAttribute`
• `double`	• `SwitchParameter`
• `float`	• `System.Globalization.CultureInfo`
• `Guid`	• `System.Net.IPAddress`
• `Hashtable`	• `System.Net.Mail.MailAddress`
• `int`	• `System.Numerics.BigInteger`
• `Int16`	• `System.Security.SecureString`
• `long`	• `TimeSpan`
• `ManagementClass`	• `UInt16`
• `ManagementObject`	• `UInt32`
• `ManagementObjectSearcher`	• `UInt64`
• `NullString`	

Users can get or set allowed properties, invoke methods, and convert objects to the type.

Constrained Language Mode permits the following Component Object Model (COM) objects:

- `Scripting.Dictionary`
- `Scripting.FileSystemObject`
- `VBScript.RegExp`

As with types, users can get properties but can only set properties on core types.

It's also important to remember that the `class` statement isn't available in Constrained Language Mode.

Native Win32 executables (such as *ping.exe*, *net.exe*, *gpupdate.exe*) will continue to function as expected. However, they're subject to AppLocker/WDAC Application Policies (if implemented).

Allowed types are accessible in a static class named `CoreTypes` in the

`System.Management.Automation` namespace.[2] However, this class has an internal access modifier, preventing modification.[3]

`[PSCustomObject]` isn't an approved type in Constrained Language Mode.

Example 3: `PSCustomObject` isn't allowed in CLM

```
$ExecutionContext.SessionState.LanguageMode

$psobject = [PSCustomObject]@{ key = 'item' }
```

```
ConstrainedLanguage

InvalidArgument: Cannot convert value to type "System.Management.Automation.
LanguagePrimitives+InternalPSCustomObject". Only core types are supported
in this language mode.
```

17.1.2.2 Modules

When importing PowerShell scripts or modules, cmdlets will inherit the parent Language Mode. For example:

Example 4: CLM restrictions apply to module cmdlets, too

```
#  Module.psm1 saved in C:\Windows\System32\WindowsPowerShell\v1.0\Modules
function Do-Something {
    # Perform a Restricted Action
    [System.Net.WebClient]::new().DownloadFile(
        'https://google.com/favicon.ico', 'D:\TEMP\SomeFile.html'
    )
}
```

Load the Module and Invoke the PowerShell Function:

```
# Import the Module from C:\Windows\System32\WindowsPowerShell\v1.0\Modules
Import-Module Module

Do-Something
```

[2] Microsoft. (2021, Jan. 08). *System.Management.Automation - TypeResolver.cs*. L706. [Online]. Available: https://github.com/PowerShell/PowerShell/blob/master/src/System.Management.Automation/engine/parser/TypeResolver.cs. [Accessed: Oct. 12, 2021].

[3] Microsoft. (2015, Jul. 20). *internal - C# Reference*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/internal. [Accessed: Oct. 12, 2021].

```
InvalidOperation: C:\Windows\System32\WindowsPowerShell\v1.0\Modules\
Test.psm1:4
Line |
   4 |      [System.Net.WebClient]::new().DownloadFile(
     |      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     | Cannot create type. Only core types are supported in this language mode.
```

CLM permits compiled modules provided they use approved types:

Example 5: A compiled module using approved types works in CLM

```
// Compiled Cmdlet Code
using System.Management.Automation;

namespace SamplePowerShellModule
{
    [Cmdlet(VerbsCommon.Get, "Something")]
    public class GetSomething : Cmdlet
    {
        // Declare the parameters for the cmdlet.
        [Parameter(Mandatory = true)]
        public string Name
        {
            get { return name; }
            set { name = value; }
        }
        private string name;

        protected override void ProcessRecord()
        {
            // We will create a PSObject which is an approved object
            PSObject pSObject = new PSObject();
            pSObject.Members.Add(new PSNoteProperty("Test Object", name));
            WriteObject(pSObject);
        }
    }
}
```

```
# Execution
Import-Module "SamplePowerShellModule.psd1"

Get-Something -Name 'Test'
```

```
Test Object
-----------
Test
```

17.1.2.3 Windows PowerShell Workflow

PowerShell 7 doesn’t support PowerShell Workflows.

Constrained Language Mode permits Windows PowerShell Script Workflows, but XAML workflows aren’t allowed (using `Invoke-Expression -Language XAML`). It permits nested workflows but not calling other workflows. It’s considered bad practice to implement due to its language limitations.

17.1.2.4 Scripts

Constrained Language Mode permits script execution (using dot-sourcing or `Invoke-Command`). It blocks scripts excluded from WDAC or App Locker policies.[4]

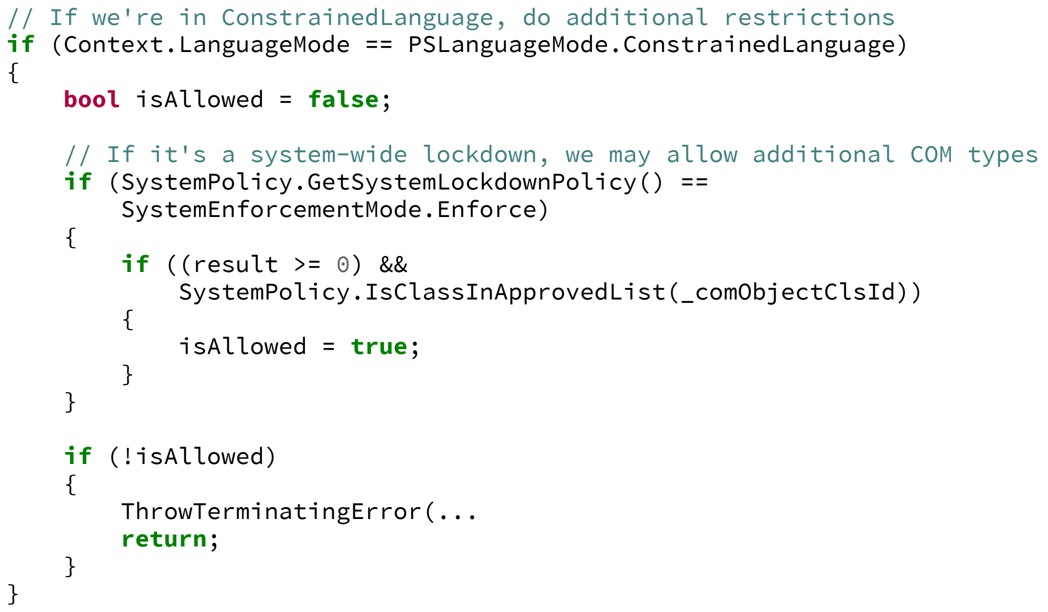
17.1.2.5 New-Object

Constrained Language Mode permits object instantiation but limits it to allowed types.

You can find Language Mode functionality embedded directly into the source code for `New-Object`.[5]

Example 6: Source code for the `New-Object` cmdlet showing Language Mode recognition

```
//protected override void BeginProcessing()

if (Context.LanguageMode == PSLanguageMode.ConstrainedLanguage)
{
    if (!CoreTypes.Contains(type))
    {
        ThrowTerminatingError(...
    }
}
```



```
// If we're in ConstrainedLanguage, do additional restrictions
if (Context.LanguageMode == PSLanguageMode.ConstrainedLanguage)
{
    bool isAllowed = false;

    // If it's a system-wide lockdown, we may allow additional COM types
    if (SystemPolicy.GetSystemLockdownPolicy() ==
        SystemEnforcementMode.Enforce)
    {
        if ((result >= 0) &&
            SystemPolicy.IsClassInApprovedList(_comObjectClsId))
        {
            isAllowed = true;
        }
    }

    if (!isAllowed)
    {
        ThrowTerminatingError(...
        return;
    }
}
```

The example below demonstrates the attempted creation of a non-approved type, resulting in an exception:

[4] Microsoft. (2017, Sep. 21). *Script rules in AppLocker.* Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/applocker/script-rules-in-applocker. [Accessed: Oct. 12, 2021].

[5] Microsoft. (2021, Jan. 08). *Microsoft.PowerShell.Commands.Utility - New-Object.cs.* PowerShell/PowerShell on GitHub. [Online]. Available: https://github.com/PowerShell/PowerShell/blob/master/src/Microsoft.PowerShell.Commands.Utility/commands/utility/New-Object.cs. [Accessed: Oct. 12, 2021].

Example 7: You can't create non-approved types in CLM

```
class Fake {}

$ExecutionContext.SessionState.LanguageMode = 'ConstrainedLanguage'

New-Object -TypeName Fake

[Fake]::new()
```

```
New-Object: Cannot create type. Only core types are supported
in this language mode.

InvalidOperation: Cannot create type. Only core types are supported
in this language mode.
```

The `[Object]::New()` method is also allowed for approved type names only.

17.1.2.6 Add-Type

`Add-Type` can load signed C# assemblies in Constrained Language Mode. It doesn't permit unsigned assemblies or Win32 APIs, however.

Example 8: Attempted to load an unsigned assembly in CLM

```
$ExecutionContext.SessionState.LanguageMode = 'ConstrainedLanguage'

$PSDir = 'C:\Windows\System32\WindowsPowerShell\v1.0'
Add-Type -LiteralPath "$PSDir\en\powershell_ise.resources.dll"
```

```
Add-Type: Cannot add type. Definition of new types is not supported
in this language mode.
```

It's important to remember that type limitations still exist, and CLM will prevent the loading of signed DLLs if they define new types.

Example 9: Attempting to load a signed assembly that defines new types

```
Add-Type -LiteralPath "C:\Program Files\PowerShell\7\pwsh.dll"
```

```
Add-Type: Cannot add type.
Definition of new types is not supported in this language mode.
```

With CLM, `Add-Type` can't load native C# code.

Example 10: Attempting to parse C# code with `Add-Type`

```
# Set Constrained Language Mode
$ExecutionContext.SessionState.LanguageMode = 'ConstrainedLanguage'

# Define C# Class
$Source = @"
public class Demo
{
    public static int Add(int a, int b)
    {
        return (a + b);
    }
}
"@

# Load the Class
Add-Type -TypeDefinition $Source
```

```
Add-Type: Cannot add type. Definition of new types is not supported
in this language mode.
```

17.1.2.7 Type Conversion

Constrained Language Mode permits type and string conversion only when the converted type is an Allowed Type. In the example below, you can see type conversion from a `string` to `DateTime` working in Constrained Language Mode:

Example 11: Type conversion between allowed types in CLM

```
"01/01/2021" -is 'String'

"01/01/2021" -is 'DateTime'

[DateTime]"01/01/2021" -is 'String'

[DateTime]"01/01/2021" -is 'DateTime'

# Wanting to take your PowerShell Further?
# USE CODE PWSH4THEWIN for 50% off the PowerShell Conference Books on Leanpub!
# https://leanpub.com/powershell-conference-book/c/PWSH4THEWIN
# https://leanpub.com/psconfbook2/c/PWSH4THEWIN
# https://leanpub.com/psconfbook3/c/PWSH4THEWIN
```

```
True
False
False
True
```

In the following example, you can see an instance of a type conversion from `string` to `fake`, resulting in an error.

Example 12: Attempting a conversion to a non-approved type

```
[Fake]"String"
```

```
InvalidArgument: Cannot convert value to type "Fake". Only core types
are supported in this language mode.
```

17.1.2.8 PowerShell Methods()

Constrained Language Mode allows calling any methods of **Approved Types**.

Calling `GetType()` with the `[String]` type:

Example 13: Calling a method of an approved type

```
([string]'Test String').GetType().Name
```

```
String
```

Calling `GetType()` with the `[Microsoft.PowerShell.Commands.WebResponseObject]` type:

Example 14: Attempting to call a method of a non-approved type

```
(Invoke-WebRequest 'https://google.com/favicon.ico').GetType().Name
```

```
InvalidOperation: Cannot invoke method.
Method invocation is supported only on core types in this language mode.
```

17.2 Limitations of Constrained Language Mode

PowerShell Constrained Language Mode is a powerful addition to PowerShell, yet there are limitations:

- Malware can exist within the constraints of Constrained Language Mode by limiting its usage to cmdlets. In this instance, it's possible to use `Invoke-WebRequest` to download malware over HTTP and use `New-ScheduledTask` and `Register-ScheduledTask` to initialize the code. Therefore, it's vital that you also deploy Application Control.
- Constrained Language Mode is only active in the PowerShell Process. Added PowerShell Assemblies won't load in Constrained Language Mode.
- You can hypothetically bypass Constrained Language Mode on Administrative accounts. Restrict non-essential administrative access to only privileged accounts.

17.2.1 PowerShell Protect

You can implement the *PowerShell Protect* module side-by-side with CLM to add additional security. The module features two parts:

1. PowerShell cmdlets that help you to create and deploy policies.
2. An AMSI (Antimalware Scan Interface) interface that inspects pre-parsed PowerShell for malicious code execution.[6]

For more information, please refer to https://ironmansoftware.com/powershell-protect[7].

17.3 Deep Diving into Windows Lockdown Policy

PowerShell relies on the Windows Lockdown Policy (WLDP) to enforce Windows Lockdown Policies, Device Guard, and Constrained Language Mode, using the `SystemPolicy` class.[8]

While WLDP provides several methods, there are three key ones. These are:

- `GetSystemLockdownPolicy()`: This is a top-level method that PowerShell uses to check the lockdown state of the system. This method calls `GetLockdownPolicy()` and `GetDebugLockdownPolicy()`. `New-Object` uses this method to validate approved object types.[9]
- `GetLockdownPolicy(string path, SafeHandle handle)`: Retrieves the lockdown policy that applies to a file. The `GetSystemLockdownPolicy()` and `ReadScriptContents()` method use this.[10]
- `GetDebugLockdownPolicy(string path)`: Retrieves any enabled debugging overrides in the `__PSLockdownPolicy` environment variable (described below). If you enable a debugging override and a WLDP policy is already in force, the system ignores the override.

17.3.1 `GetLockdownPolicy()`

`GetLockDownPolicy()` makes up the primary segment of the code, testing the WLDP and AppLocker policies. It tests them in the following order:

1. WLDP File Policy
2. AppLocker File Policy
3. System WLDP Policy
4. Debug Override Policy

[6] Microsoft. (2019, Apr. 19). *Antimalware Scan Interface (AMSI)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/amsi/antimalware-scan-interface-portal. [Accessed: Oct. 12, 2021].

[7] https://ironmansoftware.com/powershell-protect

[8] Microsoft. (2021, Jan. 10). *System.Management.Automation - wldpNativeMethods.cs*. PowerShell/PowerShell on GitHub. [Online]. Available: https://github.com/PowerShell/PowerShell/blob/master/src/System.Management.Automation/security/wldpNativeMethods.cs. [Accessed: Oct. 12, 2021].

[9] Microsoft. (2021, Jan. 08). *Microsoft.PowerShell.Commands.Utility - New-Object.cs*. PowerShell/PowerShell on GitHub. [Online]. Available: https://github.com/PowerShell/PowerShell/blob/master/src/Microsoft.PowerShell.Commands.Utility/commands/utility/New-Object.cs. [Accessed: Oct. 12, 2021].

[10] Microsoft. (2021, Apr. 19). *System.Management.Automation - ExternalScriptInfo.cs*. PowerShell/PowerShell on GitHub. [Online]. Available: https://github.com/PowerShell/PowerShell/blob/master/src/System.Management.Automation/engine/ExternalScriptInfo.cs. [Accessed: Oct. 12, 2021].

17.3.2 `GetWldpPolicy()`

This method calls and caches the result from the `WldpGetLockdownPolicy()` function (used by *wldp.dll*).[11] WDAC uses WLDP to define the configuration state.

17.3.3 `GetAppLockerPolicy()`

`GetAppLockerPolicy()` evaluates the currently implemented system and user AppLocker Policies. It achieves this by:

1. Creating an AppLocker test file name (*.psd1* & *.psm1*) inside the user's *%Temp%* directory. If the *%Temp%* directory isn't accessible, it will try `\AppData\LocalLow\Temp`.
2. Returning an enforcement policy to the caller.

17.3.4 `GetDebugLockdownPolicy()`

`GetDebugLockdownPolicy()` is an override setting for debugging purposes.

It consists of:

- Loose file matching. The system trusts all scripts that reside inside of a directory named 'System32'.
- The `__PSLockdownPolicy` environment variable. The value of this variable affects the Language Mode.

This applies to **all** scripts that reside in **any** directory named 'System32'. The `Contains()` method allows for matching in any parent or subdirectory.

17.3.4.1 `__PSLockdownPolicy` Environment Variable

`GetDebugLockdownPolicy(string path)` uses this variable for debugging, and you shouldn't use it in production. Setting `_PSLockdownPolicy` to `0` won't override existing enforced WDAC or App Locker policies.

The policy settings are:[12] [13]

- `0`: Undefined—No changes are applied.
- `4`: UMCI Enforced—Enforce the UMCI policy, enabling Constrained Language Mode.
- `8`: UMCI Audit—Set the UMCI policy to 'audit'. PowerShell won't enable Constrained Language Mode.

[11] Microsoft. (2018, May. 31). *WldpGetLockdownPolicy function*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/devnotes/wlpdgetlockdownpolicy. [Accessed: Oct. 12, 2021].

[12] Microsoft. (2021, Jan. 10). *System.Management.Automation - wldpNativeMethods.cs*. L494-504. [Online]. Available: https://github.com/PowerShell/PowerShell/blob/master/src/System.Management.Automation/security/wldpNativeMethods.cs. [Accessed: Oct. 12, 2021].

[13] Microsoft. (2021, Jan. 10). *System.Management.Automation - wldpNativeMethods.cs*. L438-454. [Online]. Available: https://github.com/PowerShell/PowerShell/blob/master/src/System.Management.Automation/security/wldpNativeMethods.cs. [Accessed: Oct. 12, 2021].

Example 15: Source code for WLDP's `__PSLockdownPolicy` implementation

```
internal static class WldpNativeConstants
{
    internal const uint WLDP_HOST_INFORMATION_REVISION = 0x00000001;

    internal const uint WLDP_LOCKDOWN_UNDEFINED = 0;
    internal const uint WLDP_LOCKDOWN_DEFINED_FLAG = 0x80000000;
    internal const uint WLDP_LOCKDOWN_SECUREBOOT_FLAG = 1;
    internal const uint WLDP_LOCKDOWN_DEBUGPOLICY_FLAG = 2;
    internal const uint WLDP_LOCKDOWN_UMCIENFORCE_FLAG = 4;
    internal const uint WLDP_LOCKDOWN_UMCIAUDIT_FLAG = 8;
}
```

```
private static SystemEnforcementMode GetLockdownPolicyForResult(
    uint pdwLockdownState)
{
    if ((pdwLockdownState & WldpNativeConstants.WLDP_LOCKDOWN_UMCIAUDIT_FLAG)
        == SystemPolicy.WldpNativeConstants.WLDP_LOCKDOWN_UMCIAUDIT_FLAG)
    {
        return SystemEnforcementMode.Audit;
    }
    else if ((pdwLockdownState &
                WldpNativeConstants.WLDP_LOCKDOWN_UMCIENFORCE_FLAG) ==
            WldpNativeConstants.WLDP_LOCKDOWN_UMCIENFORCE_FLAG)
    {
        return SystemEnforcementMode.Enforce;
    }
    else
    {
        return SystemEnforcementMode.None;
    }
}
```

17.4 Implementing Policies Using AppLocker Script Rules

17.4.1 Introduction

You can manage AppLocker with Group Policy at the following path:

```
Computer Configuration\Windows Settings\Security Settings\
Application Control Policies\App Locker
```

There are five rules definition types:

1. Executable rules: These control which executables (`*.exe, *.com`) can and can't run.[14]

[14] Microsoft. (2017, Sep. 21). *Executable rules in AppLocker*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/applocker/executable-rules-in-applocker. [Accessed: Oct. 12, 2021].

2. Windows Installer rules: These control permitted installation packages (`*.msi, *.msp, *.mst`).[15]
3. Script rules: These define permitted script formats (`*.ps1, *.bat, *.cmd, *.vbs, *.js`) and files. PowerShell Module files (`*.psm1`) and PowerShell Module Manifest files (`*.psd1`) remain unaffected.[16]
4. DLL rules: These control locations from which modules and libraries (`*.dll, *.ocx`) can run and by which users. These rules may affect system stability and performance.[17]
5. Packaged app rules: These control which Universal Windows Platform (UWP) apps and installers can run.[18]

You must also configure and enforce the rules by enabling each definition type within the AppLocker Properties. You can find the DLL rules under the *Advanced* tab. AppLocker uses an allowlist approach, enforcing all definition types. AppLocker supports policy 'Auditing' to test policy definitions before enforcement, reducing the impact on users.

If you are considering implementing AppLocker within your organization, please review the AppLocker Design Guide.[19]

AppLocker rules have three primary conditions used for evaluation. These rules are in order of best approach:

1. *(Recommended)* Publisher: Code-signed scripts are the recommended option, as this approach offers the most flexibility and security.
2. File hash: File hashes prevent modification to code but require changes to the policy as the file changes.
3. *(Not-Recommended)* File path: Rule application depending on the file or directory path or name. This approach doesn't ensure the integrity of the file since there are no mechanisms to test the identity of files.

After making any changes to AppLocker, you should refresh Group Policy by running `gpupdate /force`.

[15] Microsoft. (2017, Sep. 21). *Windows Installer rules in AppLocker.* Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/applocker/windows-installer-rules-in-applocker. [Accessed: Oct. 12, 2021].

[16] Microsoft. (2017, Sep. 21). *Script rules in AppLocker.* Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/applocker/script-rules-in-applocker. [Accessed: Oct. 12, 2021].

[17] Microsoft. (2017, Sep. 21). *DLL rules in AppLocker.* Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/applocker/dll-rules-in-applocker. [Accessed: Oct. 12, 2021].

[18] Microsoft. (2017, Oct. 10). *Packaged apps and packaged app installer rules in AppLocker.* Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/applocker/packaged-apps-and-packaged-app-installer-rules-in-applocker. [Accessed: Oct. 12, 2021].

[19] Microsoft. (2017, Sep. 21). *AppLocker design guide.* Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/applocker/applocker-policies-design-guide. [Accessed: Oct. 12, 2021].

17.4.2 Getting Started

17.4.2.1 Enable the AppLocker Service

Before creating and configuring rules, you must enable the ‘Application Identity’ Windows Service through Group Policy, under the following path:

```
Computer Configuration\Windows Settings\Security Settings\
System Services\Application Identity
```

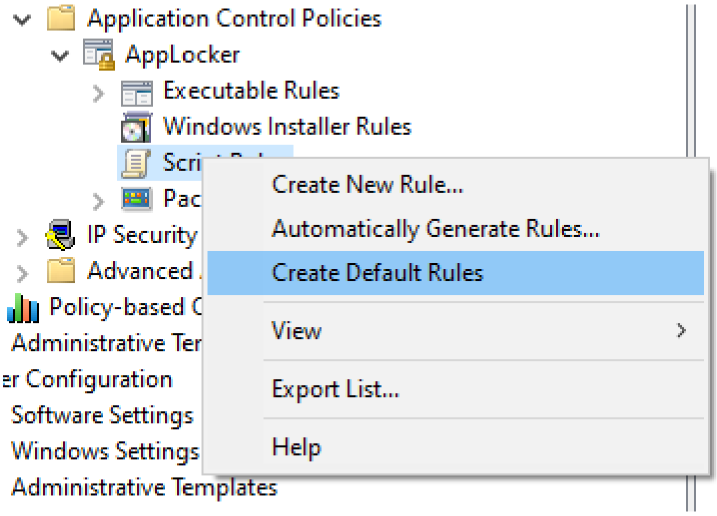
You can also manually enable the `Application Identity` service.

To enable the service with PowerShell, start a session as an Administrator and enter `Start-Service -Name 'Application Identity'`.

17.4.2.2 Creating Default Rules

AppLocker provides several tools to aid in usability. When enabling AppLocker for the first time, allow it to create the default rules.

- *Right-click* **Script Rules** and select **Create Default Rules**.



Group Policy AppLocker Menu

On the right-hand side of the window, you can see that AppLocker has created and enabled the default script rules. The default rules are:

1. Allow *All Users* to run Scripts in the “Program Files” directory, defined by the `%PROGRAMFILES%\*` PATH variable.
2. Allow *All Users* to run Scripts in the “Windows” directory, defined by the `%WINDIR%\*` PATH variable.

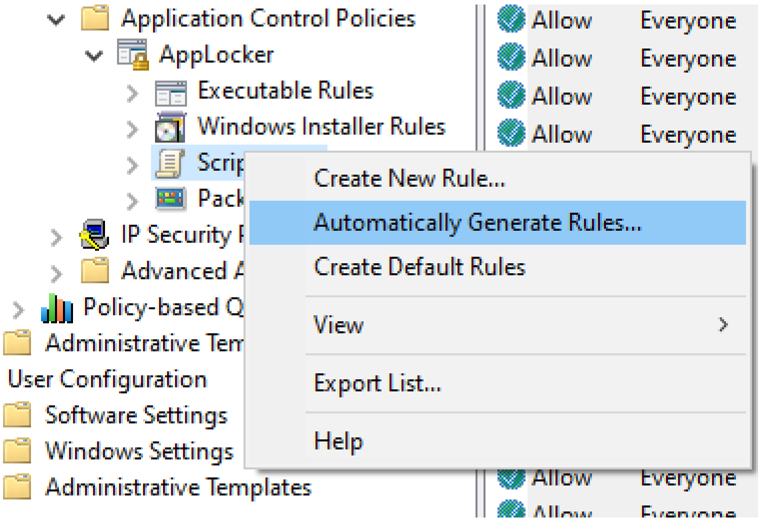
3. Allow *Local Administrators* to run *All Scripts*, enabling Administrators to continue to maintain/manage the environment.

You should remove these rules once you have your custom rules defined to prevent them from becoming a "catch-all".

17.4.2.3 Automatically Generating Rules

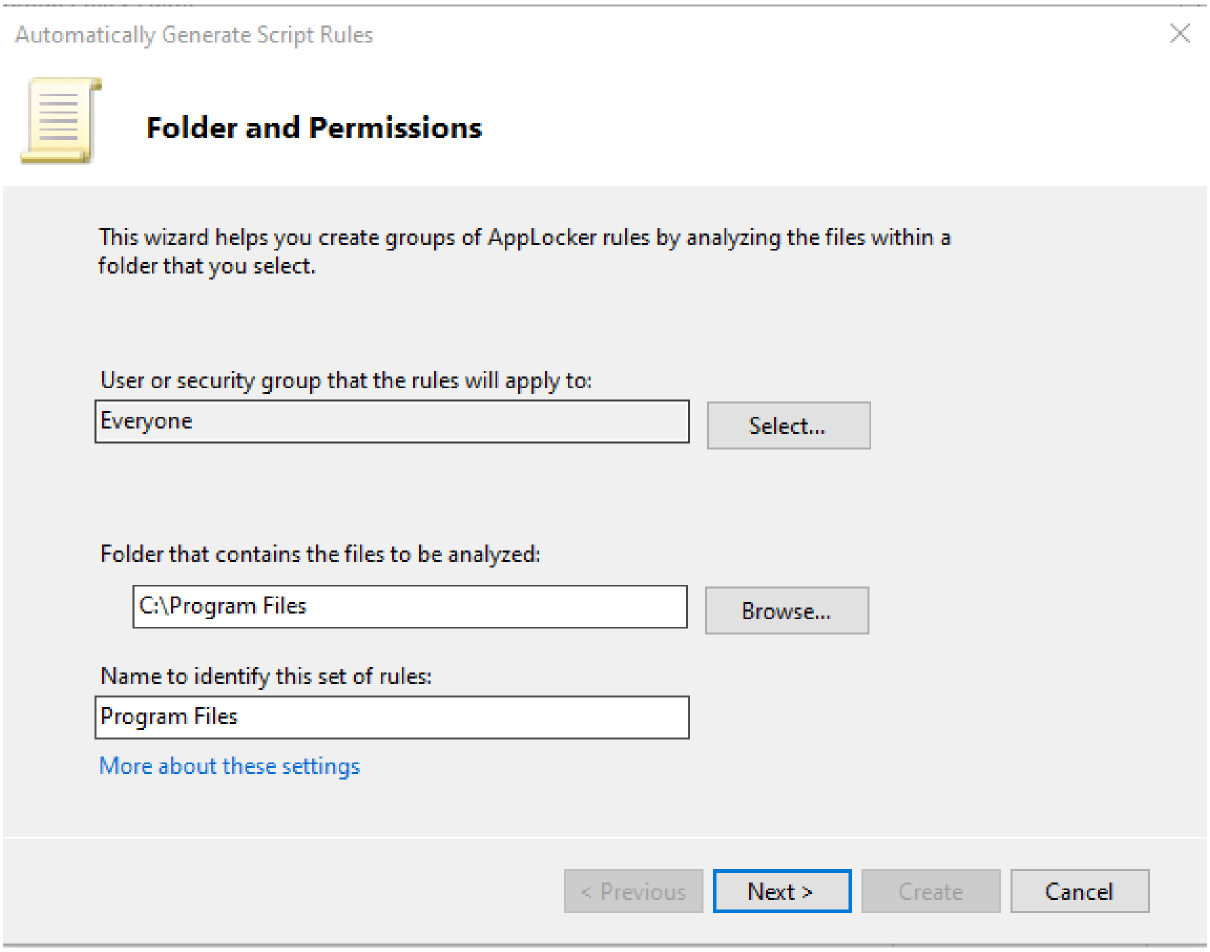
Another alternative is to generate the rules automatically. This process scans directories and creates rules from the analyzed files within them. The wizard preferentially applies the rules in order: Publisher rules, File Hash rules, followed by File Path rules.

1. *Right-click* **Script Rules** and *select* **Automatically Generate Rules…**.



AppLocker Script Rules: Autogeneration Option

2. *Select* the **Security Group** to target the users to which this will apply.
3. *Enter* the scan **Directory Path**.
4. *Enter* a *name* to identify the ruleset.



This wizard helps you create groups of AppLocker rules by analyzing the files within a folder that you select.

User or security group that the rules will apply to:
Everyone

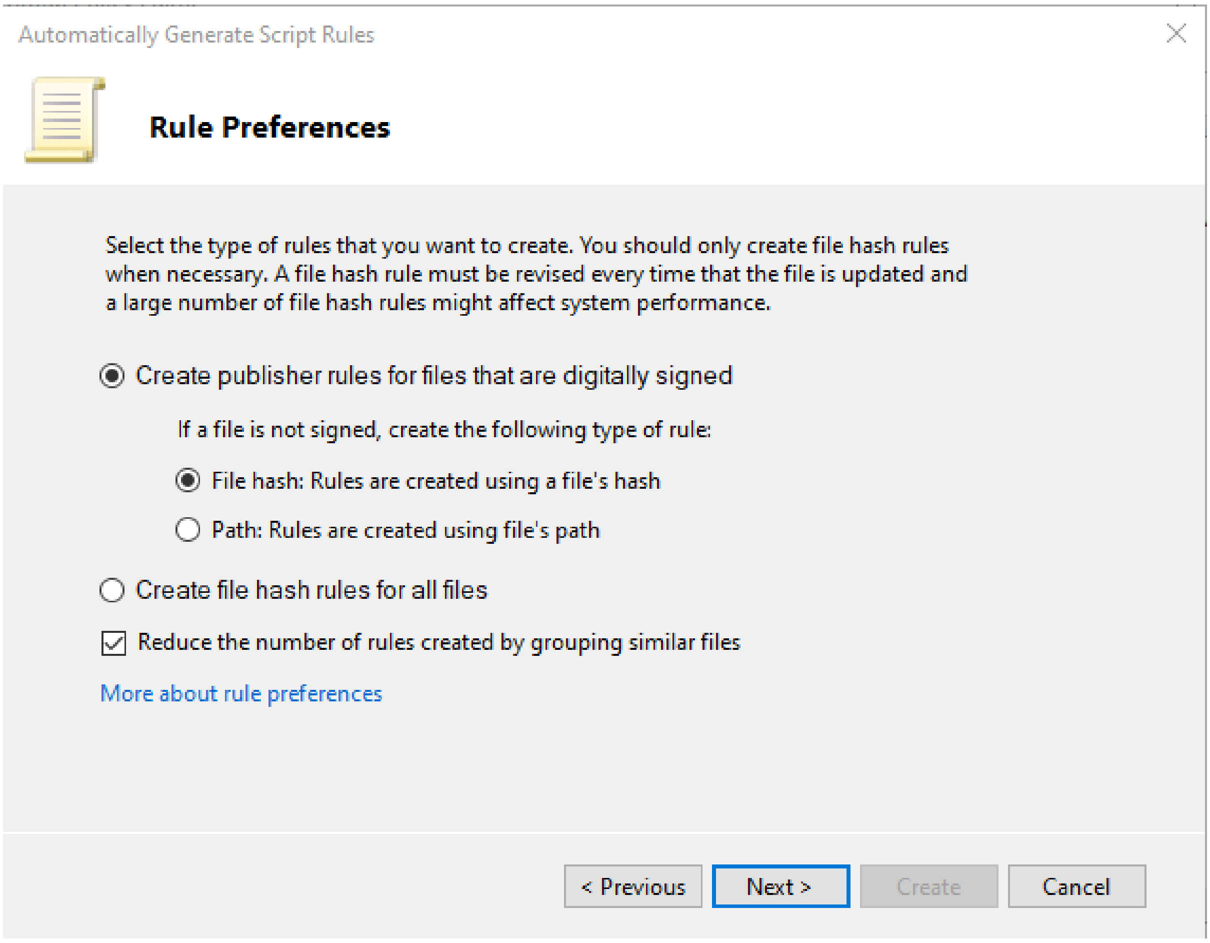
Folder that contains the files to be analyzed:
C:\Program Files

Name to identify this set of rules:
Program Files

AppLocker Script Rules Autogeneration Wizard: Group and Target

5. *Select* **Next** >.
6. Ensure the following settings are selected:

   - *Create publisher rules for files that are digitally signed.*
   - *File Hash: Rules are created using a file's hash.*
   - *Reduce the number of rules created by grouping similar files.*



Select the type of rules that you want to create. You should only create file hash rules when necessary. A file hash rule must be revised every time that the file is updated and a large number of file hash rules might affect system performance.

● Create publisher rules for files that are digitally signed

If a file is not signed, create the following type of rule:

● File hash: Rules are created using a file's hash

○ Path: Rules are created using file's path

○ Create file hash rules for all files

☑ Reduce the number of rules created by grouping similar files

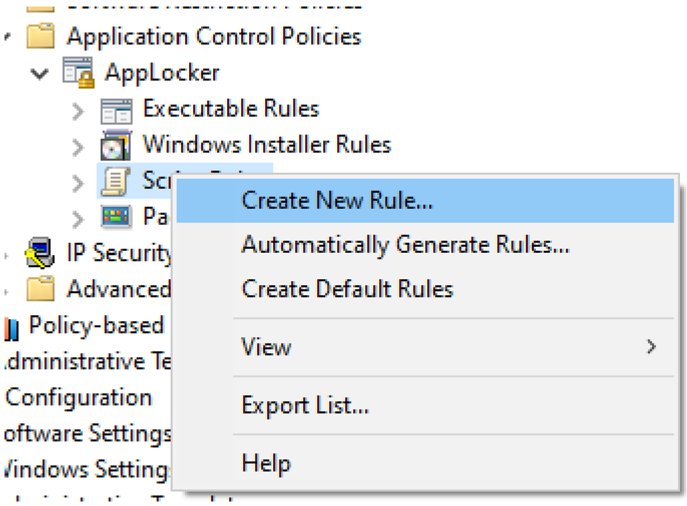
AppLocker Script Rules Autogeneration Wizard: Primary Condition Options

7. *Select* **Next >**.
8. *Select* **Create**.

17.4.2.4 Creating Custom Rules

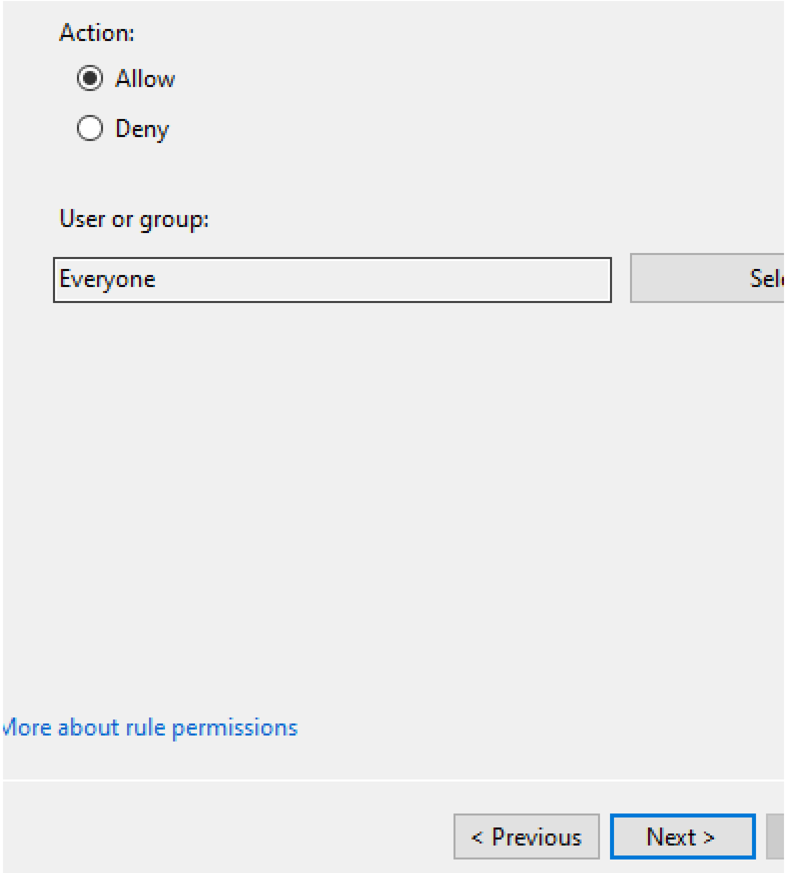
Creating custom rules is the preferred option providing granular scope over the target environment.

1. *Right-click* **Script Rules** and *select* **Create New Rule**.



AppLocker Script Rules: New Rule Option

2. *Select* **Next >**.
3. Under the *Action*, *select* **Allow**.
4. *Select* the **Security Group** to target the users to which this will apply.

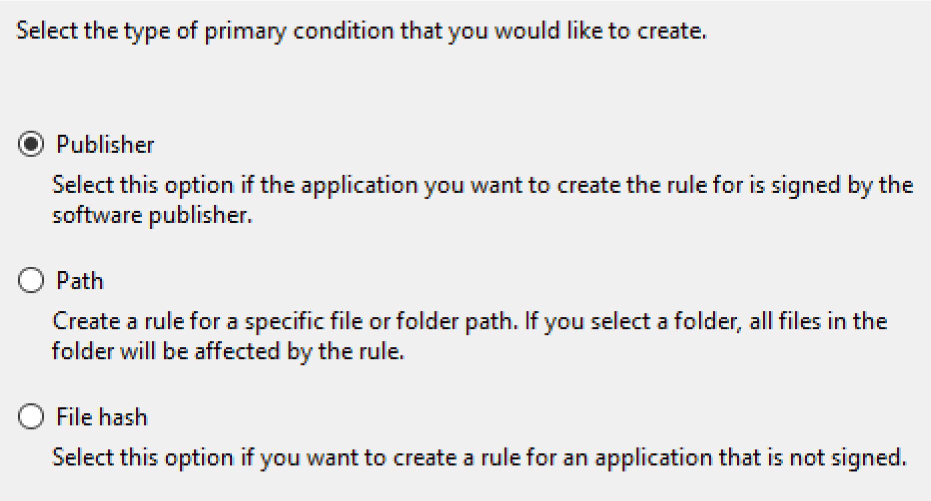


Action:
● Allow
○ Deny

User or group:
Everyone

AppLocker Script Rules New Rule Wizard: Group and Action

5. *Select* **Next >**
6. *Select* the primary condition (Publisher, Path, File Hash).



Select the type of primary condition that you would like to create.

● Publisher
Select this option if the application you want to create the rule for is signed by the software publisher.

○ Path
Create a rule for a specific file or folder path. If you select a folder, all files in the folder will be affected by the rule.

○ File hash
Select this option if you want to create a rule for an application that is not signed.

AppLocker Script Rules New Rule Wizard: Primary Condition Options

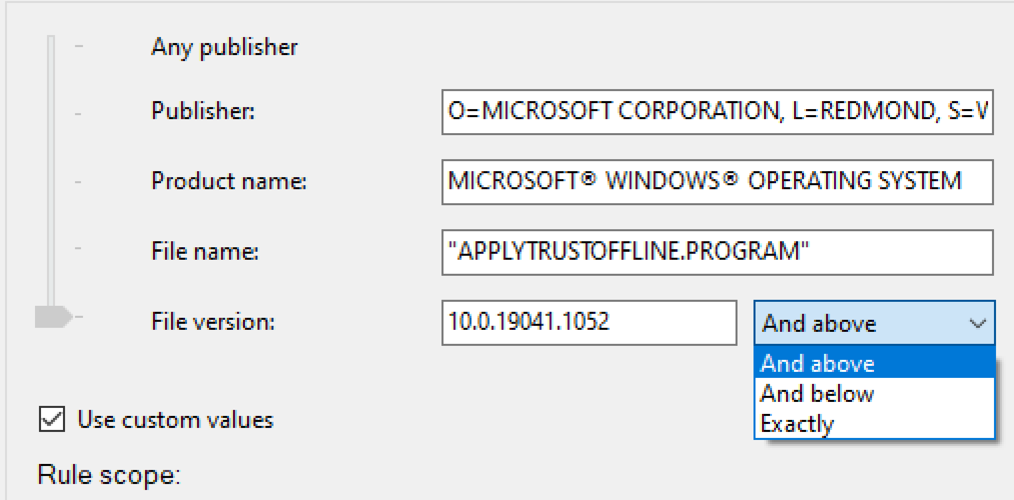
7. *Select* **Next >**

17.4.2.4.1 Publisher Condition

1. Under *Reference file*, *Select* **Browse...** and locate the `*.ps1` file.
2. You can assign different security levels, ranging from least restrictive (top) to most restrictive (bottom). Each level includes the requirements from the previous one. These are:

   - *(Lowest)* Publisher: Requires the issuer of the file's certificate to adhere to the current value.
   - Product name: Requires the signed file's product name to adhere to the current value.
   - File name: Requires the signed file's file description to adhere to the current value.
   - *(Highest)* File version: Requires the signed file's version to adhere to the current value, and optionally higher or lower.

   You can customize the fields further by selecting *Use custom values*.



Any publisher

Publisher: O=MICROSOFT CORPORATION, L=REDMOND, S=V

Product name: MICROSOFT® WINDOWS® OPERATING SYSTEM

File name: "APPLYTRUSTOFFLINE.PROGRAM"

File version: 10.0.19041.1052 And above

☑ Use custom values

Rule scope:

AppLocker Script Rules New Rule Wizard: Publisher Condition Options

Drag the slider to the desired level. It's recommended to use the *Product Name* for an internal self-signed certificate.

3. *Select* **Next** >
4. Add any exceptions to the rule Exceptions follow the AppLocker conditions (File, Publisher, Hash).
5. *Select* **Next** >
6. *Enter* a **Name**: and (optionally) **Description** and *Select* **Create**.

> For PowerShell scripts, AppLocker populates only the Publisher field from the reference file.

17.4.2.4.2 Path Condition

1. *Select* **Browse Files**… or **Browse Folders**… and select the file/directory you wish to add.
2. *Select* **Next** >.
3. Add any exceptions to the rule. Exceptions follow the AppLocker conditions (File, Publisher, Hash).
4. *Enter* a **Name**: and (optionally) **Description** and *Select* **Create**.

17.4.2.4.3 File Hash Condition

1. *Select* **Browse Files**… or **Browse Folders**… and file/directory you wish to add. By selecting a directory, the wizard will enumerate all `*.ps1` files in the directory. *Note: This isn't a recursive search.*
2. *Enter* a **Name**: and (optionally) **Description** and *Select* **Create**.

17.4.2.5 Enabling Auditing

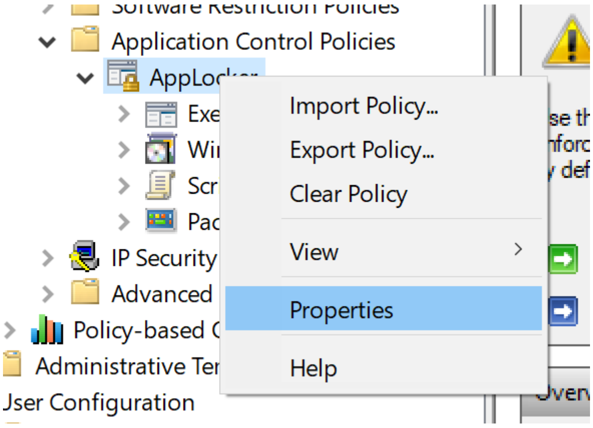
Before policy enforcement, you can test changes within your organization without affecting end-users.[20] AppLocker raises events in the Event Log under `Application and Services Logs\Microsoft\Windows\AppLocker`. For more information on Event IDs, please refer to Using Event Viewer with AppLocker[21].

To enable policy Auditing:

1. *Right-click* on **AppLocker** and *Select* **Properties**.

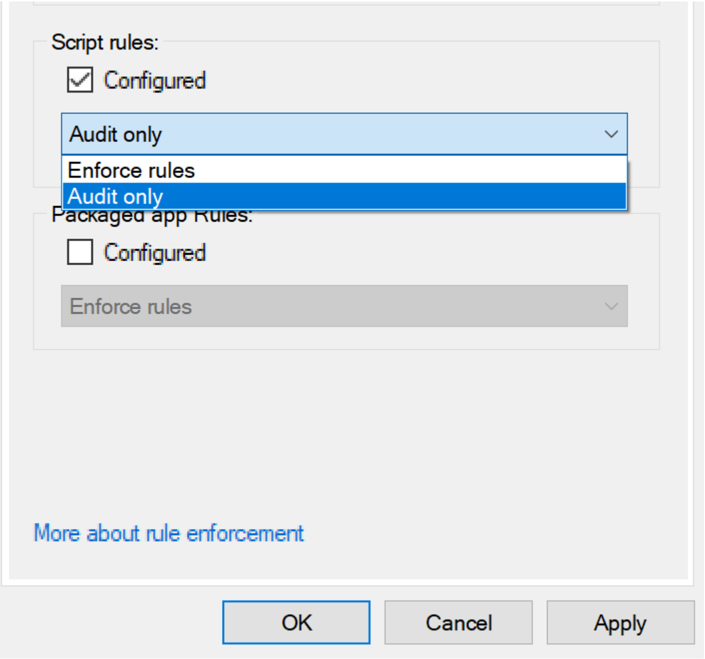
[20] Microsoft. (2018, Jun. 08). *Configure an AppLocker policy for audit only*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/applocker/configure-an-applocker-policy-for-audit-only. [Accessed: Oct. 12, 2021].

[21] https://learn.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/applocker/using-event-viewer-with-applocker



AppLocker Properties Option

2. *Enable* **Script rules** and *Select* **Audit only**.


Script rules:
☑ Configured
Audit only
Enforce rules
Audit only

Packaged app Rules:
□ Configured
Enforce rules

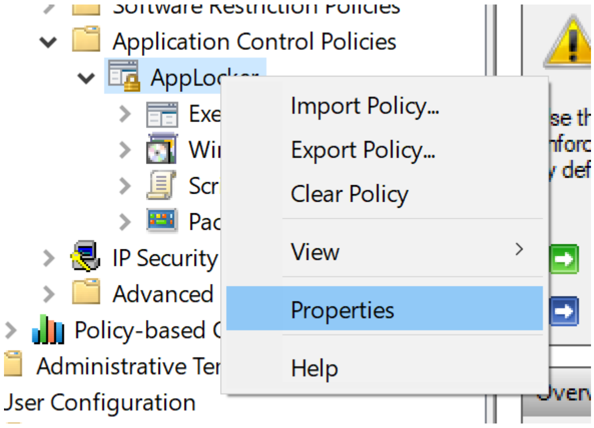
AppLocker Properties: Script Rules Enforcement

3. *Select* **Apply** and **OK**.

17.4.2.6 Enabling Policy Enforcement

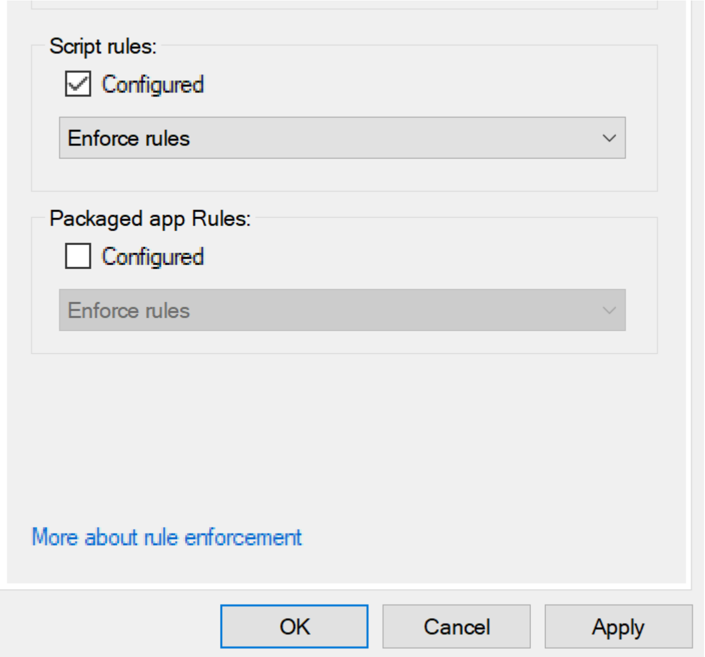
Once ready, you can enforce AppLocker policies:

1. *Right-click* on **AppLocker** and *Select* **Properties**.



AppLocker Properties Option

2. *Enable* **Script rules** and *Select* **Enforce rules**.


Script rules:

☑ Configured

Enforce rules

Packaged app Rules:

□ Configured

Enforce rules

AppLocker Properties: Script Rules Enforcement

3. *Select* **Apply** and **OK**.

17.5 Implementing Policies Using WDAC

17.5.1 What Is WDAC?

WDAC or Windows Defender Application Control Policy is a policy framework implemented alongside Windows Defender on Windows 10 desktop (1903) and Server 2016 operating systems. WDAC was formally known as *Configurable Code Integrity* (CGI) and *Device Guard* (DG). When WDAC policies are in force, PowerShell runs Constrained Language Mode.

WDAC is supported on Windows 10 1903 and Windows Server 2016 or higher.

WDAC under the hood functions the same as AppLocker, yet policy enforcement isn't editable using AppLocker Group Policy and uses WLDP to query the state. AppLocker is limited to on-premise endpoints on the local domain using Group Policy, whereas WDAC is deployable to cloud and on-premise endpoints. WDAC supports the following endpoints:

- Mobile Device Management (MDM) Solutions (such as Microsoft Intune). *This solution only supports Windows 10 Devices.*
- Microsoft Endpoint Configuration Manager (MECM)
- Scripting Solution
- Group Policy

WDAC is the recommended deployment for cloud devices.

Similar to AppLocker, it's mandatory to enable Policy Auditing before enforcement. Otherwise, end-users may not be able to use apps/programs.

The following section focuses on deploying WDAC using Microsoft Intune. For more deployment methodologies, please see the WDAC Deployment Guide[22].

17.6 Deploying WDAC Using Microsoft Intune

These settings will change over time as Microsoft adds or changes features.

17.6.1 Prerequisites

17.6.1.1 Intune Setup Requirements

The setup requirements for Intune are as follows:[23]

- Enterprise Mobility + Security (EMS)/Intune subscription
- Office 365 subscription (for Office apps and app-protection-policy managed apps)
- Apple APNs Certificate (to enable iOS/iPadOS device platform management)
- Azure AD Connect (for directory synchronization)
- Intune On-Premises Connector for Exchange (for Conditional Access for Exchange On-Premises, if needed)
- Intune Certificate Connector (for SCEP certificate deployment, if needed)

[22] https://learn.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/windows-defender-application-control-deployment-guide

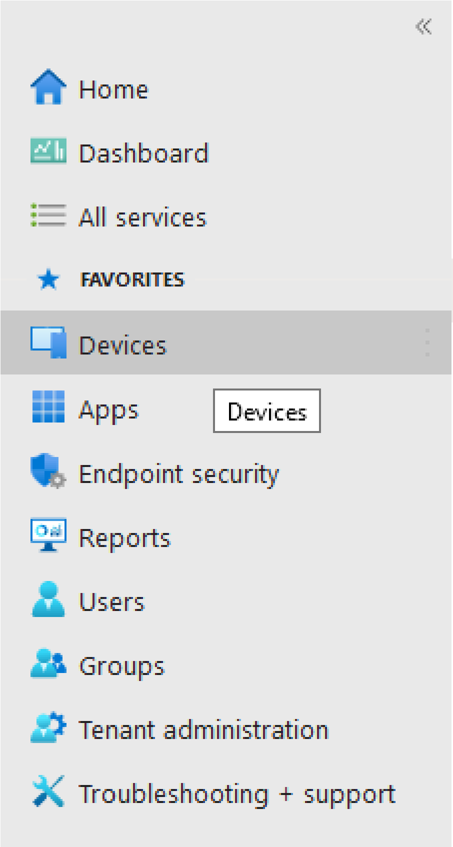
[23] Microsoft. (2020, Feb. 19). *Implement your Microsoft Intune plan.* MicrosoftDocs/IntuneDocs on GitHub. [Online]. Available: https://github.com/MicrosoftDocs/IntuneDocs/blob/master/intune/fundamentals/planning-guide-onboarding.md. [Accessed: Oct. 12, 2021].

17.6.1.2 Intune Device Requirements

- Windows 10 Desktop (1903 or Higher).

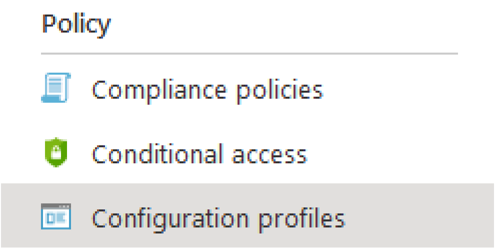
17.6.2 Creating a Device Policy

1. *Open* the URL: **`https://endpoint.microsoft.com/`**.
2. On the left-hand pane, *select* **Devices**.



Microsoft Endpoint Manager Admin Center (MEMAC) Devices Option

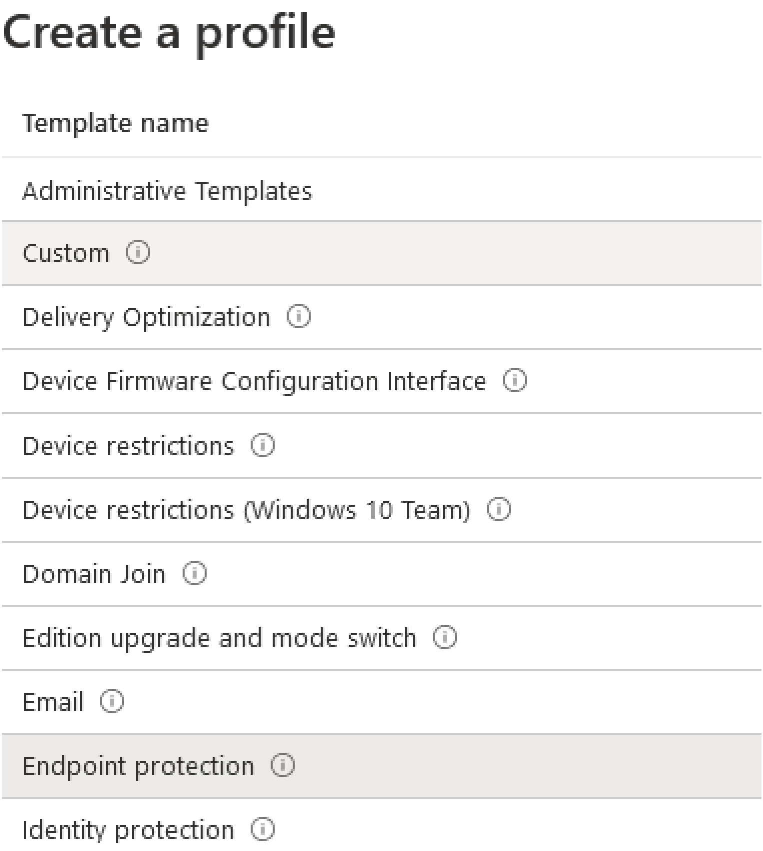
3. Under **Policy**, *Select* **Configuration profiles**.



MEMAC Device Configuration Profiles

4. *Select* **Create profile**.
5. Under **Platform**, *Select* **Windows 10 and Later**.
6. Under **Profile type**, *Select* **Templates**.
7. *Select* **Endpoint protection**.

8. *Select* **Create**



Create a profile

Template name

Administrative Templates

Custom ⓘ

Delivery Optimization ⓘ

Device Firmware Configuration Interface ⓘ

Device restrictions ⓘ

Device restrictions (Windows 10 Team) ⓘ

Domain Join ⓘ

Edition upgrade and mode switch ⓘ

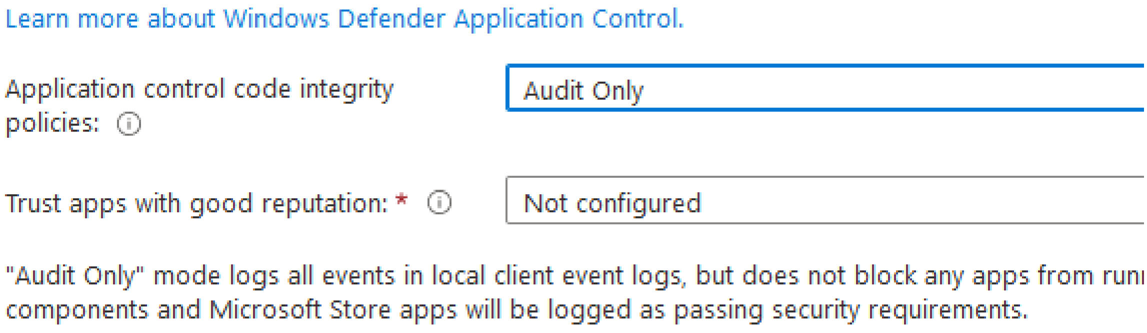
Email ⓘ

Endpoint protection ⓘ

Identity protection ⓘ

MEMAC Device Profile Templates

9. *Enter* a **Name** and **Description** and press *Next.*
10. *Select* **Microsoft Defender Application Control**.
11. *Select* **Application control code integrity policies**
12. Depending on the type, *select* **Audit Only** to audit the policy or **Enforce** to enforce the policy.
13. *Select* **Next**



Learn more about Windows Defender Application Control.

Application control code integrity policies: ⓘ Audit Only

Trust apps with good reputation: * ⓘ Not configured

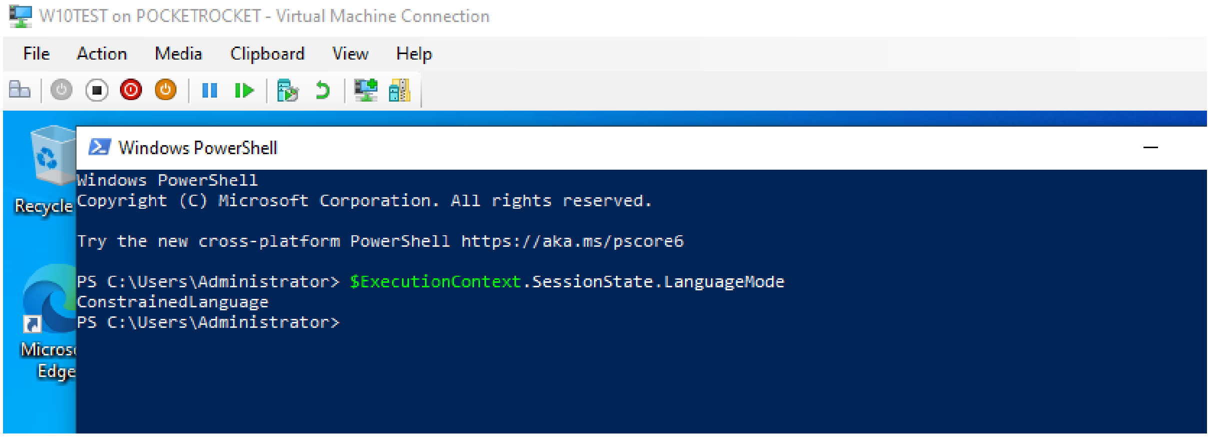
"Audit Only" mode logs all events in local client event logs, but does not block any apps from run
components and Microsoft Store apps will be logged as passing security requirements.

MEMAC Application Control Enforcement

14. *Assign* the targeted users that will apply to this policy. The policy uses Azure Active Directory (AAD) Groups as a filter mechanism.
15. *Select* **Next**

16. Intune offers additional attribute-based filtering, which you can apply. *Add* any additional rules as required.
17. *Select* **Next**
18. *Select* **Create**

If the policy type is set to **enforce**, the onboarded machine will apply the WDAC policy, and PowerShell will start in Constrained Language Mode.


W10TEST on POCKETROCKET - Virtual Machine Connection
File Action Media Clipboard View Help
Windows PowerShell
Recycle
Micros
Edge

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\Administrator> $ExecutionContext.SessionState.LanguageMode
ConstrainedLanguage
PS C:\Users\Administrator>
```

PowerShell with WDAC-Enforced Constrained Language Mode

17.7 Best Practices

1. Remove PowerShell 2.0 from all machines. See Windows PowerShell 2.0 Deprecation[24].
2. Implement Script Signing (refer to the Script Signing chapter) for production environments.
3. Implement WDAC (for Desktops) and AppLocker (for Server)
4. Enforce Script Signing Policies based on script signing certificates.

17.8 Further Reading

- About Languages Modes—Microsoft Docs[25]
- Windows Lockdown Policy—Microsoft Docs[26]
- WLDP Native Methods Source—PowerShell on GitHub[27]
- WDAC and AppLocker Overview—Microsoft Docs[28]
- Constrained Language Mode—Microsoft DevBlogs[29]
- TypeResolver Source—PowerShell on GitHub[30]

[24] https://devblogs.microsoft.com/powershell/windows-powershell-2-0-deprecation/

[25] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_language_modes?view=powershell-7.1#constrained-language-constrained-language

[26] https://learn.microsoft.com/en-us/windows/win32/devnotes/windows-lockdown-policy

[27] https://github.com/PowerShell/PowerShell/blob/master/src/System.Management.Automation/security/wldpNativeMethods.cs

[28] https://learn.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/wdac-and-applocker-overview

[29] https://devblogs.microsoft.com/powershell/powershell-constrained-language-mode/

[30] https://github.com/PowerShell/PowerShell/blob/master/src/System.Management.Automation/engine/parser/TypeResolver.cs

- Script Rules in AppLocker—Microsoft Docs[31]
- Windows PowerShell 2.0 Deprecation—Microsoft DevBlogs[32]
- Windows 10 Device Guard and Credential Guard Demystified—Microsoft TechCommunity Blogs[33]
- Introduction To Device Guard VBS and WDAC—Microsoft Docs[34]
- AppLocker Policies Deployment Guide[35]
- Configure an AppLocker Policy for Audit Only—Microsoft Docs[36]
- Using Event Viewer with AppLocker—Microsoft Docs[37]
- WDAC Deployment Guide—Microsoft Docs[38]

[31] https://learn.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/applocker/script-rules-in-applocker

[32] https://devblogs.microsoft.com/powershell/windows-powershell-2-0-deprecation/

[33] https://techcommunity.microsoft.com/t5/iis-support-blog/windows-10-device-guard-and-credential-guard-demystified/ba-p/376419

[34] https://learn.microsoft.com/en-us/windows/security/threat-protection/device-guard/introduction-to-device-guard-virtualization-based-security-and-windows-defender-application-control

[35] https://learn.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/applocker/applocker-policies-deployment-guide

[36] https://learn.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/applocker/configure-an-applocker-policy-for-audit-only

[37] https://learn.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/applocker/using-event-viewer-with-applocker

[38] https://learn.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/windows-defender-application-control-deployment-guide

18. Just Enough Administration

18.1 Introduction

Just Enough Administration (JEA) is the last security pillar of the PowerShell security ecosystem. JEA is a configuration framework that enables IT to deploy and use PowerShell remoting in secure environments. JEA utilizes PowerShell session configuration to define PowerShell remoting runspaces. This chapter will explore the use of JEA to secure a PowerShell remoting endpoint.

18.1.1 Requirements

This chapter assumes that you have a rudimentary understanding of Desired State Configuration (DSC). Please refer to the Infrastructure as Code chapter, which describes DSC.

Both Windows PowerShell and PowerShell (Core) running on Windows support JEA.

> This chapter refers to Windows PowerShell versions through 5.1 as ***Windows PowerShell*** and versions of the cross-platform edition (formerly PowerShell Core) beginning with 6.0 as ***PowerShell (Core)***.

18.2 Background of JEA

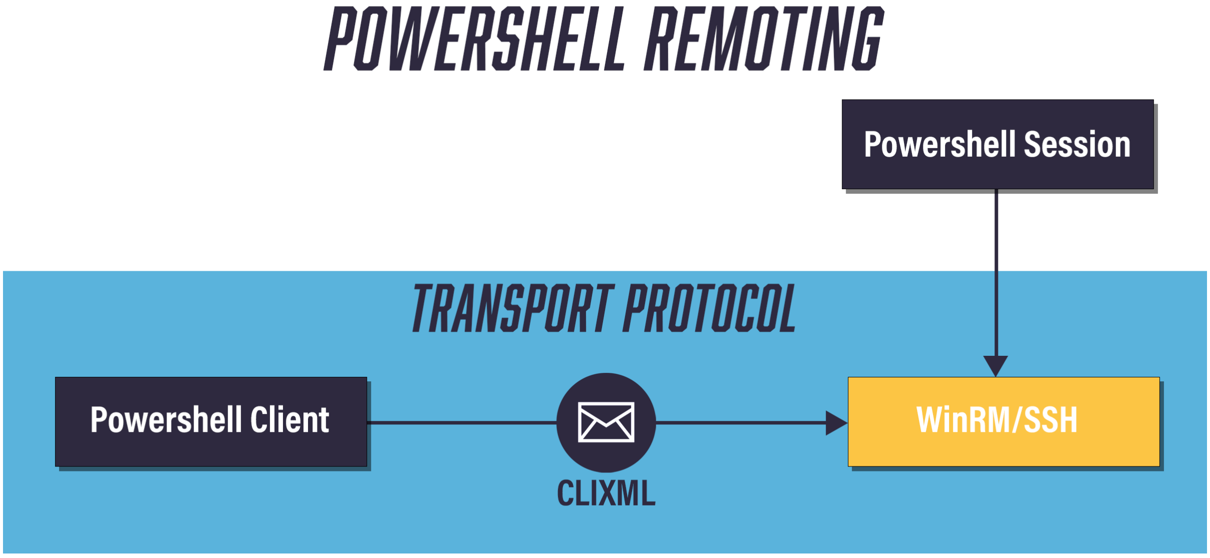
Just Enough Administration is based on the 'Just-in-Time' security model, which provides time-sensitive and granular access to privileged tasks. This model enables you to use the *principle of least privilege* (PoLP) to reduce the scope and nature of lateral attacks by limiting permissions. PowerShell provides granular control over its features by registration of a PowerShell remoting session configuration. This configuration includes control over:

- Language Mode.
- Execution Policy.
- Preloaded or defined aliases, assemblies, functions, or modules.
- Cmdlet allowlisting.
- Cmdlet denylisting.
- Runtime accounts.
- Logging.
- User drives.
- Versioning.
- Granular Control over Parameter and values.

18.2.1 PowerShell Remoting 101

PowerShell remoting is a mechanism which allows remote execution of PowerShell over a network. That is, you can execute PowerShell commands on one or more remote computers. PowerShell remoting uses the WinRM (Windows Remote Management) or SSH protocol as a transport layer, encapsulating the session state within the body/payload. WinRM is the Microsoft implementation of the WS-Management (Web Services-Management) SOAP protocol, which uses HTTP(S) for transport on ports 5985 (HTTP) and 5986 (HTTPS).[1] Windows PowerShell versions 2.0 through 5.1 use WinRM, while PowerShell (Core) versions beginning with 6.0 can use either WinRM or SSH. The PowerShell remoting session uses the CLIXML data type to serialize/deserialize object data between the endpoints. CLIXML is an XML-based representation of .NET object structures.[2]

JEA is only available with WinRM/WSMan and not SSH.



PowerShell Remoting Block Diagram

18.2.2 An Overview of PowerShell Session Configuration

When PowerShell connects to a remote session, the remote PowerShell instance preloads a configuration stored by the Web Services-Management (WS-Management or WSMan) service. This configuration describes which JEA features the session supports.

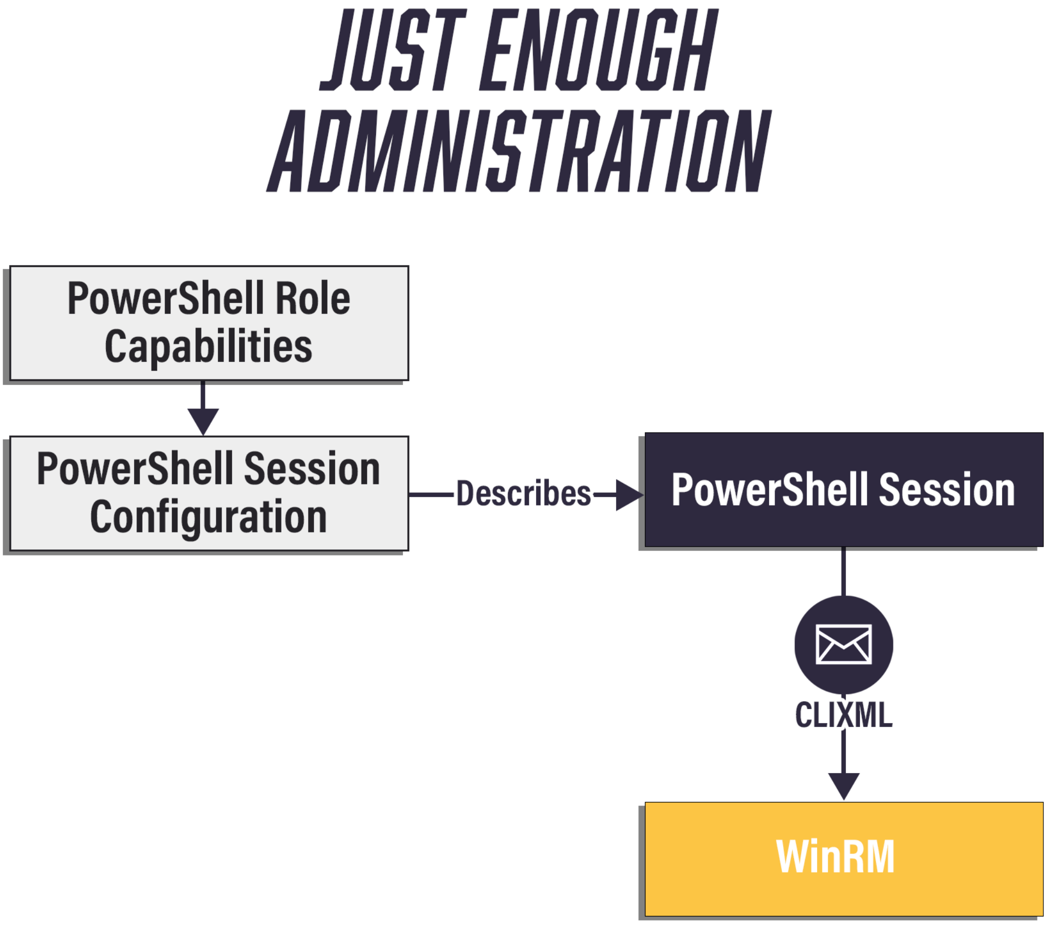
PowerShell registers its remoting WSMan plugin configuration using PowerShell session configuration files. Each of these stores configuration data as a `[Hashtable]` and has a `.pssc` file extension.

[1] Microsoft. (2020, Aug. 19). *About Windows Remote Management.* Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/winrm/about-windows-remote-management. [Accessed: Oct. 14, 2021].

[2] ECMA. (2012, Jun.). *ECMA-335: Common Language Infrastructure (CLI).* Ecma International. [Online]. Available: https://www.ecma-international.org/publications-and-standards/standards/ecma-335/. [Accessed: Oct. 14, 2021].

You can find the PowerShell session configuration cmdlets in the `Microsoft.PowerShell.Core` module.[3] They include:

1. `New-PSSessionConfigurationFile`: Creates a PowerShell session configuration.
2. `Register-PSSessionConfiguration`: Registers a PowerShell session configuration file.
3. `Get-PSSessionConfiguration`: Returns registered PowerShell session configurations.
4. `Unregister-PSSessionConfiguration`: Unregisters a previously registered PowerShell session configuration.
5. `Get-PSSessionCapability`: Audits a user's JEA capabilities within a PowerShell remoting session configuration.
6. `Test-PSSessionConfigurationFile`: Tests a PowerShell session configuration file for errors.
7. `Set-PSSessionConfiguration`: Updates or changes a registered PowerShell session configuration.
8. `Enable-PSSessionConfiguration`: Enables a registered session configuration.
9. `Disable-PSSessionConfiguration`: Disables a registered session configuration.


JUST ENOUGH ADMINISTRATION
PowerShell Role Capabilities
PowerShell Session Configuration
Describes
PowerShell Session
CLIXML
WinRM

PowerShell JEA Block Diagram

This chapter explores creating, registering, updating, and auditing WSMan plugins.

[3] Microsoft. (2020, Nov. 17). *Microsoft.PowerShell.Core*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/. [Accessed: Oct. 14, 2021].

18.2.3 PowerShell Remoting Authentication and Transport Encryption

PowerShell Remoting can use any of several authentication methodologies. These are:

- **Default**: Default authentication uses the *Negotiate* authentication type.
- **Kerberos**: Kerberos Authentication is a ticket-based mutual-authentication protocol requiring devices to prove their identity.[4] Mutual Authentication is a feature where both the client and service must prove their identity. The Kerberos Key Distribution Center (KDC) runs on the Domain Controller role. The KDC is responsible for issuing tickets. The KDC uses Active Directory as its account database and the Global Catalog for directing referrals to other domains.
- **Basic (`RFC7235`)**: Basic Authentication uses the 'basic' authentication type within the HTTP Authorization Header for each WinRM SOAP request as the authentication mechanism. Basic Authentication doesn't provide any mutual encryption and relies on WinRM's transport encryption for secure transmission (HTTPS). [5]

  The Authorization header encapsulates the Username and Password as `basic <username>:<password>` in Base64.

  For Example:

  ```
  # UserName: Michael.Zanatta
  # Password: 123Password
  #
  # Header
  Authorization = Basic TWljaGFlbC5aYW5hdHRhOjEyM1Bhc3N3b3Jk
  ```

- **Negotiate (Windows Integrated Authentication)**: Windows Integrated Authentication (IWA) or Windows Authentication uses two authentication protocols (Kerberos or NTLM). Both authentication mechanisms function as a Single-Sign-On mechanism within the local Active Directory Domain.

  - (NTLM) NT LAN Manager is a challenge-response authentication protocol used to authenticate clients within an Active Directory domain. The services will send a challenge to the client, requiring the client to respond with its authentication token. Services that can't validate the authentication token will forward the request to a Domain Controller. NTLM doesn't provide mutual authentication, so the client can't verify the server's identity.
  - (Kerberos) is a ticket-based mutual-authentication protocol requiring devices to prove their identity. The Kerberos Key Distribution Center (KDC) runs on the Domain Controller role. The KDC is responsible for issuing tickets. The KDC uses Active Directory as its account database and the Global Catalog for directing referrals to other domains.

[4] Microsoft. (2021, Jul. 07). *Kerberos Authentication Overview.* Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/windows-server/security/kerberos/kerberos-authentication-overview. [Accessed: Oct. 28, 2021].

[5] Internet Engineering Task Force. (2014, Jun.). *RFC7235: Hypertext Transfer Protocol (HTTP/1.1): Authentication.* RFC Editor. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7235. [Accessed: Oct. 27, 2021].

During authentication, IWA primarily uses Kerberos, with NTLM serving as a backup. The means of selecting which authentication type is:

- Kerberos: Used if the client is in a Windows Domain and the service isn't:
    * An IPv4 or IPv6 Address (*192.168.1.1* or *[2001:0db8:0000:0000:0000:ff00:0042:8329]*).
    * A Loopback Address (*127.0.0.1* or *[::1]*)
    * Using `localhost` as the endpoint.
- NTLM: Authenticates within a Windows Domain with IP Addresses/Loopback and serves as a backup.[6]

- **CredSSP (Credential Security Support Provider)**: Credential Security Support Provider is a Security Support Provider (SSP)[7], used to extend the Windows authentication mechanism outside the required scope of the authentication type. The provider encrypts User Credentials (using TLS) and transmits them to the SSP, where the SSP uses SPNEGO [8] [9] to negotiate which authentication mechanism to use (Kerberos/NTLM).[10] Once negotiated, the server will validate the credentials on behalf of the user.

⚠ User credentials can be retrieved if a server is compromised.

- **Cert Authentication (Certificate-Based Authentication)**: Certificate-Based Authentication uses X.509 certificates as a means of authentication. The TLS process is as follows:

1. Client connects to the server.
2. The Server presents its TLS Public Certificate to the client. The Client validates that the Certificate Authority is trusted, based on the Certificate Chain contained within the certificate.
3. The Client presents its TLS Public Certificate to the server. The Certificate Thumbprint, defined by the PowerShell Remoting Client, will locate the certificate from the Local Machine or Local User Certificate store (`Certificate::localmachine\my` or `Certificate::currentuser\my`). When running PowerShell under a non-elevated prompt, only the local user certificate store is accessible.[11]
4. The Server verifies the Client Certificate by:
    1. Validating whether the Certificate Authority is trusted, based on the Certificate Chain contained within the certificate.

[6] Microsoft. (2021, Aug. 07). *Security Considerations for PowerShell Remoting using WinRM*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/scripting/learn/remoting/winrmsecurity. [Accessed: Oct. 28, 2021].

[7] Microsoft. (2020, Aug. 20). *Security Support Providers (SSPs)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/rpc/security-support-providers-ssps-. [Accessed: Oct. 29, 2021].

[8] Microsoft. (2021, Aug. 01). *Microsoft Negotiate*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/secauthn/microsoft-negotiate. [Accessed: Oct. 29, 2021].

[9] Network Working Group. (2006, Jun.). *RFC4559: SPNEGO-based Kerberos and NTLM HTTP Authentication in Microsoft Windows*. RFC Editor. [Online]. Available: https://www.rfc-editor.org/rfc/rfc4559. [Accessed: Oct. 27, 2021].

[10] Microsoft. (2021, Aug. 01). *Credential Security Support Provider*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/secauthn/credential-security-support-provider. [Accessed: Oct. 29, 2021].

[11] Jordan Borean. (2018, Jan. 24). *Demystifying WinRM*. Blogging for Logging. [Online]. Available: https://www.bloggingforlogging.com/2018/01/24/demystifying-winrm/. [Accessed: Oct. 28, 2021].

   2. Checking that the certificate is in the `TrustedPeople` Store.

5. The session is established (the server grants access).[12]

Once the PowerShell Session is established, either PowerShell Session State encryption or WinRM encryption is applied depending on whether the WinRM protocol is HTTP or HTTPS. If the protocol is HTTP, message-level encryption based on the authentication protocol is applied. For HTTPS, it uses TLS within the HTTP transport. The following table describes the Authentication Type with the type of encryption using WinRM HTTP:

Authentication Type	**WinRM HTTP Message-Level Encryption Type**
Default *(using Kerberos)*	AES-256
Default *(using NTLM)*	RC4
Kerberos	AES-256
Basic	None
Negotiate - NTLM	RC4
Negotiate - Kerberos	AES-256
CredSSP	TLS

18.3 PowerShell Role Capabilities

PowerShell role capability (PSRC) files are configuration files that define the module capabilities *exposed through the PowerShell session configuration.* PowerShell session capability files contain configuration data stored as a PowerShell `[Hashtable]`. During the initialization of the session, role capabilities permissions combine into a single role capability, similar in functionality to Access Control Entries (ACE) and Access Control Lists (ACL).

Windows PowerShell preloads `*.psrc` files from the `RoleCapabilities` subdirectory within a module directory. In PowerShell (Core), you can define PSRC files directly within the PowerShell session configuration file using the `RoleDefinitions` property.

An example of the directory structure for a Windows PowerShell module is below:

Example 1: PowerShell Module directory structure

```
ModuleName
│   Module.psm1 # Module File
│   Module.psd1 # Module Manifest File
│
├───Public
├───Private
├───Help
└───RoleCapabilities
        ServiceMaintenance.psrc
        ProcessMaintenance.psrc
```

[12] Microsoft. (2020, Aug. 20). *What is mutual TLS (mTLS)?*. CloudFlare. [Online]. Available: https://www.cloudflare.com/en-us/learning/access-management/what-is-mutual-tls/. [Accessed: Nov. 01, 2021].

Within the example, role capability files exist in the `RoleCapabilities` subdirectory of the PowerShell module *ModuleName*.

18.3.1 Implementing Windows PowerShell Role Capabilities in the Console

The following example describes the process of implementation using Windows PowerShell:

1. Create the PSRC file using `New-PSRoleCapabilityFile`[13].

Example 2: Creating two role capability files

```
$roleParameters = @{
    Path = ".\ServiceMaintenance.psrc"
    Author = "User01"
    CompanyName = "Fabrikam Corporation"
    Description = "This role enables users to get/restart any service"
    VisibleCmdlets = "Restart-Service", "Get-Service"
}
New-PSRoleCapabilityFile @roleParameters

$roleParameters = @{
    Path = ".\ProcessMaintenance.psrc"
    Author = "User01"
    CompanyName = "Fabrikam Corporation"
    Description = "This role enables users to stop/start processes"
    VisibleCmdlets = "Get-Process", "Stop-Process"
}
New-PSRoleCapabilityFile @roleParameters
```

2. *(Windows PowerShell)* Copy the file into a PowerShell module.

Example 3: Creating a blank PowerShell module

```
# Create the PowerShell Module

# Create a folder for the module

$joinPathParams = @{
    Path = $env:ProgramFiles
    ChildPath = "WindowsPowerShell\Modules\TestJEAModule"
}

$modulePath = Join-Path @joinPathParams
New-Item -ItemType Directory -Path $modulePath
# Create an empty script module
New-Item -ItemType File -Path (Join-Path $modulePath "TestJEAModule.psm1")

# Create a PowerShell Module Manifest File

$moduleManifestParams = @{
    Path = Join-Path $modulePath "TestJEAModule.psd1"
    RootModule = "TestJEAModule.psm1"
}

New-ModuleManifest @moduleManifestParams
```

[13] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/new-psrolecapabilityfile

```
Directory: C:\Program Files\WindowsPowerShell\Modules

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d----          14/07/1993 00:00 AM                TestJEAModule

    Directory: C:\Program Files\WindowsPowerShell\Modules\TestJEAModule

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a---          14/07/1993 00:00 AM              0 TestJEAModule.psm1
```

18.3.2 Implementing PowerShell (Core) Role Capabilities in the Console

The following steps describe the process of implementation using PowerShell (Core):

1. Create the PSRC file using `New-PSRoleCapabilityFile`. Please refer to Example 2.
2. Define the PSRC file within the PowerShell session configuration using the `RoleDefinitions` parameter. The `RoleDefinitions` parameter is a `[Dictionary]` or `[Hashtable]` that uses the `DOMAIN/(User/Group): @{Configuration Type}` syntax. PowerShell (Core) defines three types of configuration. These are:

- **RoleCapabilities**: These are the enumerated PSRC file names from preloaded PowerShell modules.
- **RoleCapabilityFiles**: Any other PSRC files. PowerShell (Core) doesn't require you to couple PSRC files to a module, making it applicable to configuration outside of a module.
- **Custom**: You can define custom PSRC configurations associated with a group using parameters instead of a PSRC file or loaded path. Using the cmdlet parameters reduces management complexity at the cost of scalability. Values can be:[14]

```
  [-ModulesToImport <Object[]>]
  [-VisibleAliases <String[]>]
  [-VisibleCmdlets <Object[]>]
  [-VisibleFunctions <Object[]>]
  [-VisibleExternalCommands <String[]>]
  [-VisibleProviders <String[]>]
  [-ScriptsToProcess <String[]>]
  [-AliasDefinitions <IDictionary[]>]
  [-FunctionDefinitions <IDictionary[]>]
  [-VariableDefinitions <Object>]
  [-EnvironmentVariables <IDictionary>]
  [-TypesToProcess <String[]>]
  [-FormatsToProcess <String[]>]
  [-AssembliesToLoad <String[]>]
```

In the following example, the configuration references two PSRC files using the `RoleCapabilityFiles` key:

[14] Microsoft. (2021, Sep. 27). *New-PSSessionConfigurationFile (Microsoft.PowerShell.Core).* Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/new-pssessionconfigurationfile. [Accessed: Oct. 14, 2021].

Example 4: Creating a PowerShell session configuration with two role capabilities

```
$configSettings = @{

    # Sample Properties. These are incomplete.
    Path = '.\SampleFile.pssc'
    SchemaVersion = '1.0.0.0'
    Author = 'User01'
    Copyright = '(c) Fabrikam Corporation. All rights reserved.'
    CompanyName = 'Fabrikam Corporation'

    # This is the role definitions.
    RoleDefinitions = @{
        # Define the group and then role capability files.
        # *Note that this is an array, so multiple files can be parsed.*
        'CONTOSO\PSSC_HELPDESK_RAS01' = @{
            RoleCapabilityFiles = '.\ServiceMaintenance.psrc',
                                  '.\ProcessMaintenance.psrc'
        }
    }
}

New-PSSessionConfigurationFile @configSettings
```

18.3.3 Implementing PowerShell Role Capabilities Within DSC

The JeaDsc module[15] uses the `JeaRoleCapabilities` resource for PSRC definitions and supports the same formatting as `New-PSRoleCapabilityFile`. The duplicate files will be rewritten in DSC and referenced within the `JeaSessionConfiguration` resource:

Example 5: A JeaRoleCapabilities DSC resource defining two role capabilities

```
[DscLocalConfigurationManager()]
Configuration JeaRoleCapabilities
{
    # This is a partial DSC resource that is used
    # to store the JeaRoleCapabilities.

    Import-DscResource -ModuleName JeaDSC

    Node localhost {

        # Define the first resource
        JeaRoleCapabilities ServiceMaintenanceCapability {

        Path =  "C:\Temp\ServiceMaintenance.psrc"
        VisibleCmdlets = "Restart-Service", "Get-Service"
        Description = "This role enables users to get/restart any service."
        # Author/ComanyName/GUID/Copyright are not supported.

        }

        # Define the second resource
        JeaRoleCapabilities ProcessMaintenanceCapability {

        Path =  "C:\Temp\ProcessMaintenance.psrc"
        VisibleCmdlets = "Get-Process", "Stop-Process"
```

[15] https://www.powershellgallery.com/packages/JeaDsc

```
        Description = "This role enables users to stop/start processes."
        # Author/ComanyName/GUID/Copyright are not supported.

        }

        <#
        # This is an incomplete resource.
        JeaSessionConfiguration AppServerMaintanceEndpoint
        {
            # Before executing this resource, the DSC Resources:
            # ServiceMaintenanceCapability & ProcessMaintenanceCapability need
            # to execute before the creation of the session configuration.
            DependsOn =
            '[JeaRoleCapabilities]ServiceMaintenanceCapability',
            '[JeaRoleCapabilities]ProcessMaintenanceCapability'
        }
        #>

    }
}
```

It's important to remember that the `JeaRoleCapabilities` *must* execute before the creation/registration of the session configuration. Otherwise, the DSC configuration will fail.

Please read the JEA Role Capabilities documentation[16]. This covers in-depth architecture/design and implementation.

You can also use `Find-RoleCapability` to search registered repositories for PowerShell role capabilities.

18.4 Getting Started With PowerShell Session Configuration

Creating a PowerShell session configuration involves:

1. Enabling PowerShell remoting.
2. Creating one or more PowerShell role capability files.
3. Registering the PowerShell session configuration.

Please see Enable Azure Automation State Configuration[17] for more details on DSC onboarding for Azure Automation.

[16] https://learn.microsoft.com/en-us/powershell/scripting/learn/remoting/jea/role-capabilities
[17] https://learn.microsoft.com/en-us/azure/automation/automation-dsc-onboarding

Example 6: The DSC server configuration used for the examples in this chapter

```
# ServerConfigurationData.psd1
@{

    AllNodes = @(
        @{
            NodeName = "DC01"
            Role = "DomainController"
            PSRemotingEnabled = $true
            PSRemotingConfigurationType = @(
                "HelpdeskResetPassword",
                "DNSManagement"
            )
        },
        @{
            NodeName = "FS01"
            Role = "FileSrver"
            PSRemotingEnabled = $false
        },
        @{
            NodeName = "HRBW"
            Role = "HybridRunbookWorker"
            PSRemotingEnabled = $true
            PSRemotingConfigurationType = "NoRestrictions"
        }
    )
}
```

18.4.1 Step 1: Enabling PowerShell Remoting

PowerShell remoting is best deployed and managed using a management framework, as opposed to using ad hoc tools.

1. **Ansible/Chef/Desired State Configuration (DSC)**: This chapter uses DSC for examples.
2. **Group Policy**: You can use Group Policy to enable or disable PowerShell remoting.
3. **SCCM/Intune**: This is another good option for deploying PowerShell remoting to machines; incorporated into the machine deployment script.
4. **Azure Arm/Terraform/Cloud Formation/Puppet**: Many tools allow the deployment of PowerShell using Infrastructure as Code (IaC).

You can use `Enable-PSRemoting -Force` to enable PowerShell via the console. However, these examples use DSC.

Use the following DSC configuration to enable PowerShell remoting:

Example 7: A DSC configuration to enable PSRemoting

```
# Installing the Module

Find-Module -Name WSManDsc -Repository PSGallery | Install-Module

# WSManConfig.ps1
#Requires -module WSManDsc

<#
    .DESCRIPTION
        Enable compatibility HTTP and HTTPS listeners, set
        maximum connections to 100.
#>

Configuration PowerShellRemoting
{

    Import-DscResource -ModuleName 'PSDesiredStateConfiguration'
    Import-DscResource -ModuleName 'WSManDsc'

    Node $AllNodes.Where{$_.PSRemotingEnabled}.NodeName {


        WSManServiceConfig ServiceConfig {

            IsSingleInstance                 = 'Yes'
            MaxConnections                   = 100
            AllowUnencrypted                 = $false
            AuthCredSSP                      = $false

        }

        Service EnableWinRM {
            Name = 'WinRM'
            StartupType = 'Automatic'
            State = 'Running'
        }

        Script EnablePowerShellRemoting {
            DependsOn = '[Service]EnableWinRM'

            SetScript = {
                Enable-PSRemoting
            }
            GetScript = {
                @{
                    Result = (
                        [Bool](New-PSSession . -ErrorAction SilentlyContinue)
                        -and [Bool](Test-WSMan -ComputerName .))
                }
            }
            TestScript = {
                $state = [scriptblock]::Create($GetScript).Invoke()
                $state.Result
            }
        }

    }
}
```

18.4.2 Step 2: Creating/Registering the PowerShell Session Configuration

Traditionally, you would have to create PowerShell session configuration files using `New-PSSessionConfigurationFile` and `Register-PSSessionConfiguration` in the PowerShell console. The process is:

1. Create the PowerShell session configuration file using `New-PSSessionConfigurationFile`.
2. Test the file using `Test-PSSessionConfigurationFile`.
3. Register the configuration using `Register-PSSessionConfiguration`.

As an alternative, this chapter demonstrates the `JeaDSC` resource, which only requires `JeaSessionConfiguration` to create and register PowerShell session configurations.

18.4.2.1 What is JeaDsc?

JeaDsc is a Microsoft DSC resource that allows you to deploy session configurations on multiple machines. The following are prerequisites for JeaDsc.

1. **PowerShell 5.1**: PowerShell 5.1 or later is required.
2. **PowerShell role capabilities**: One or more PowerShell role capability (PSRC) files as part of a PowerShell module.
3. **JeaDsc**: Installation of the JEA DSC module from the PowerShell Gallery.
4. Access to a DSC Pull Server such as Azure Automation[18].

Once the JeaDsc configuration is ready, you can apply it to endpoints using the JeaRoleCapabilities and JeaSessionConfiguration resources. For more information on `JeaRoleCapabilities`, please review the Implementing Windows PowerShell Role Capabilities in the Console section. The `JeaSessionConfiguration` resource is similar to `New-PSSessionConfigurationFile`. However, some properties (for example, `RoleDefinitions`) require a `[Hashtable]` wrapped as a `[String]`.

That is:

```
RoleDefinitions     = "@{RoleDefinitions}"
```

In the following scenario, management has decided that all helpdesk staff must be able to restart services and stop processes. Below is an example of the JeaDsc PowerShell session configuration for this:

[18] https://learn.microsoft.com/en-us/azure/automation/tutorial-configure-servers-desired-state

Example 8: DSC session configuration enabling two capabilities for helpdesk staff

```
Configuration JEAMaintenance
{
    Import-DscResource -Module JeaDsc

    # Apply the session configuration to only the machines
    # that have PSRemoting Enabled
    Node $AllNodes.Where{$_.PSRemotingEnabled}.NodeName {

        # Define the first resource
        JeaRoleCapabilities ServiceMaintenanceCapability {

            Path =  "C:\Program Files\WindowsPowerShell\Modules\" +
                    "Demo\RoleCapabilities\ServiceMaintenance.psrc"
            VisibleCmdlets = "Restart-Service", "Get-Service"
            Description = "This role enables users to get/restart any service"
            # Author/ComanyName/GUID/Copyright are not supported.

        }

        # Define the second resource
        JeaRoleCapabilities ProcessMaintenanceCapability {

            Path =  "C:\Program Files\WindowsPowerShell\Modules\" +
                    "Demo\RoleCapabilities\ProcessMaintenance.psrc"
            VisibleCmdlets = "Get-Process", "Stop-Process"
            Description = "This role enables users to stop/start processes"
            # Author/ComanyName/GUID/Copyright are not supported.

        }

        JeaSessionConfiguration HelpDeskManagmenetEndpoint
        {
            Name                = 'JEAMaintenance'
            RunAsVirtualAccount = $true
            Ensure              = 'Present'
            DependsOn           =
            '[JeaRoleCapabilities]ServiceMaintenanceCapability',
            '[JeaRoleCapabilities]ProcessMaintenanceCapability'
            RoleDefinitions     = "@{
            'Contoso\ServiceMaintenanceCapability' = @{ RoleCapabilities =
                                                    'ServiceMaintenanceCapability'}
            'Contoso\ProcessMaintenanceCapability' = @{ RoleCapabilities =
                                                    'ProcessMaintenanceCapability'}
            }
            "
            TranscriptDirectory = 'C:\Temp\Transcripts'
        }

    }

}
```

Changes to the PowerShell session configuration will cause the *WinRM* service to restart, disconnecting any open PowerShell sessions.

The DSC module functions as a wrapper for the PowerShell session configuration, so most parameters within the cmdlet apply to the resource. JeaDsc has the following properties:

- `[String]` **RoleDefinitions**: Defines the role definition map for the endpoint. *Requires a* `[Hashtable]` *wrapped as a* `[String]`

  Syntax:

```
RoleDefinitions = @'
@{
    DOMAIN\User|Group =
        @{ RoleCapabilities = "Setting" }
}
'@
```

- `[Bool]` **RunAsVirtualAccount**: Runs the session configuration as the machine's (virtual) administrator account.

  Syntax:

```
RunAsVirtualAccount = $true
```

- `[String[]]` **RunAsVirtualAccountGroups**: Optional groups associated with the virtual administrator account.

  Syntax:

```
RunAsVirtualAccountGroups = 'Group1', 'Group2'
```

- `[String]` **GroupManagedServiceAccount**: Configures the session to run within a Group Managed Service Account[19].

  Syntax:

```
GroupManagedServiceAccount = 'DOMAIN\gMSAGroup1'
```

- `[String]` **TranscriptDirectory**: A directory where JeaDsc should save transcripts.

  Syntax:

```
TranscriptDirectory = 'C:\FolderPath\'
```

- `[String[]]` **ScriptsToProcess**: Scripts to run on startup.

  Syntax:

```
TranscriptDirectory = 'C:\FolderPath\Script1.ps1',
                      'C:\FolderPath\Script2.ps1'
```

- `[String]` **SessionType**: Specifies the type of session that PowerShell should create.

  Values can be:

  - **Empty**: No modules added to the session. Use for creating custom sessions.
  - **Default**: Adds `Microsoft.PowerShell.Core`. Includes `Import-Module`.
  - **RestrictedRemoteServer**: Includes `Exit-PSSession`, `Get-Command`, `Get-FormatData`, `Get-Help`, `Measure-Object`, `Out-Default`, and `Select-Object`.

  Syntax:

[19] https://learn.microsoft.com/en-us/windows-server/security/group-managed-service-accounts/getting-started-with-group-managed-service-accounts

```
SessionType = 'RestrictedRemoteServer'
```

- `[Bool]` **MountUserDrive**: Configures sessions that use this configuration to expose the `User:` PSDrive.

  Syntax:

```
MountUserDrive = $true
```

- `[Long]` **UserDriveMaximumSize**: Optional maximum size of user drive in bytes. The default is 50MB. The example below sets the maximum size to 500MB.

  Syntax:

```
UserDriveMaximumSize = 524288000
```

- `[String[]]` **RequiredGroups**: Conditional Access rules. *Requires a* `[Hashtable]` *wrapped as a* `[String]`

  Syntax:

```
'
@{
    And = "RequiredGroup1",
    @{ Or = "OptionalGroup1", "OptionalGroup2" }
}
'
```

- `[Object[]]` **ModulesToImport**: List of modules to import. *Requires an array of* `[String]` *and* `[Hashtable]` *objects.*

  Syntax:

```
"
'CustomModule',
@{
    ModuleName = 'CustomModuleName';
    ModuleVersion = '1.0.0.0';
    GUID = 'GUID'
}
"
```

- `[String[]]` **VisibleAliases**: Command aliases to expose in the PowerShell session.

  Syntax:

```
VisibleAliases = 'gci', 'gm'
```

- `[String[]]` **VisibleCmdlets**: Cmdlets to expose in the PowerShell session.

  Syntax:

```
VisibleCmdlets = 'Get-ChildItem', 'Get-Member'
```

- `[String[]]` **VisibleFunctions**: Functions to expose in the PowerShell session.

  Syntax:

```
  VisibleFunctions = 'Do-Something', 'Do-SomethingElse'
```

- `[String[]]` **VisibleExternalCommands**: Limits the session's external binaries, scripts, and commands. By default, no commands are visible.

  Syntax:

```
  VisibleExternalCommands = 'C:\dosomething.ps1', 'C:\thirdparty.dll'
```

- `[String[]]` **VisibleProviders**: Limits which PowerShell providers are available in the session.

  Syntax:

```
  VisibleProviders = 'FileSystem', 'Function', 'Variable'
```

- `[String[]]` **AliasDefinitions**: Command aliases to add to the session. *Requires a* `[Hashtable]` *wrapped as a* `[String]`

  Syntax:

```
  "@{ AliasName = 'Cmdlet'; AnotherAlias = 'Cmdlet'}"
```

- `[String[]]` **FunctionDefinitions**: Custom functions to define in the session. These run in the default Language Mode and therefore have access to the filesystem, registry, and commands not available in the session. You must also add the names of these to **VisibleFunctions** to make them visible to JEA users. *Requires a* `[Hashtable]` *wrapped as a* `[String]`

  Syntax:

```
  FunctionDefinitions = "@{
      Name = 'Do-Something';
      ScriptBlock = {
          param($MyInput)
          $MyInput
      }
  }"
```

- `[String]` **VariableDefinitions**: Variables to declare in the session. *Requires a* `[Hashtable]` *wrapped as a* `[String]`

  Syntax:

```
VariableDefinitions = "@{
    Name = 'Variable1';
    Value = { # Dynamic Value }
},
@{
    Name = 'Variable1';
    Value = 'Static'
}"
```

- `[String]` **EnvironmentVariables**: Environmental variables to declare in the session. *Requires a* `[Hashtable]` *wrapped as a* `[String]`

  Syntax:

```
EnvironmentVariables = "@{
    Variable1 = 'Variable Value';
    Variable2 = 'Variable Value2';
    Variable3 = 'Variable Value3';
},
```

- `[String[]]` **TypesToProcess**: Type files (.ps1xml) to load into session.

  Syntax:

```
TypesToProcess = 'Types1.ps1xml', 'Types2.ps1xml'
```

- `[String[]]` **FormatsToProcess**: Format files (.ps1xml) to load into session.

  Syntax:

```
FormatsToProcess = 'Types1.ps1xml', 'Types2.ps1xml'
```

- `[String[]]` **AssembliesToLoad**: Assemblies to load into the session.

  Syntax:

```
AssembliesToLoad = 'Newtonsoft.Json', 'System.IO.File'
```

- `[int]` **HungRegistrationTimeout**: Timeout period to wait for the PowerShell session configuration registration in seconds. The default timeout threshold is `10`. Setting a value of `0` disables the timeout.

  Syntax:

```
HungRegistrationTimeout = 50
```

For more information on the JeaDsc Module, refer to the JeaDsc documentation[20].

18.4.3 Connecting to a PowerShell Session Configuration

Traditionally, when connecting to a PowerShell remoting session, administrators and service accounts use `Invoke-Command`, `Import-PSSession`, or `New-PSSession; Enter-PSSession` to connect to a remote machine. These connections default to the `Microsoft.PowerShell` session configuration, which is Windows PowerShell. However, the use of the `-ConfigurationName` parameter allows you to specify a different session configuration:

[20] https://github.com/dsccommunity/JeaDsc/blob/master/source/Classes/JeaSessionConfiguration.ps1

Example 9: Sending a PSRemoting command with a custom session configuration

```
$params = @{
    ComputerName = 'DC1'
    ConfigurationName = 'JEAMaintenance'
    ScriptBlock = { Get-Service }
}

Invoke-Command @params
```

18.4.4 Role Definition Design Considerations

When deploying PowerShell session configuration files, it's essential to consider Active Directory and its place in PowerShell remoting governance.

- Use Active Directory groups to define Administrator permission scopes. Direct user assignment to configuration adds complexity to the solution.
- When using Active Directory groups, describe the *RoleDefinition*, capability, and machine. For instance, `PSHRemoting-<ComputerName>-<Capability>` becomes: `PSHRemoting-DC1-ServiceMaintenance`. There are also limitations within the *RoleDefinitions* parameter/property in that you can't assign duplicate Active Directory groups to separate capabilities. Always ensure that the capability name matches the Active Directory group.
- Implement Role-Based Access Control (RBAC) to prevent direct assignment to Active Directory groups. Leveraging RBAC roles and groups ensures that user roles themselves receive PowerShell remoting session configuration. The result is that the role governs user access instead of a group. Owners can assign/remove access according to approvals.
- Simplify each role capability. Rather than having a single role capability for a specific function, it's better to practice simplifying each role capability down to a single instance. For example, a helpdesk might require the ability to query/stop processes and query/stop/start services on an application server. Breaking this down reveals two role capabilities:

  1. Query/stop processes
  2. Query/stop/start services

  You can add these role capabilities to the PowerShell session configuration file:

Example 10: Two role capabilities defined as part of a session configuration

```
    JeaSessionConfiguration HelpDeskManagmenetEndpoint
    {
        Name                  = 'JEAMaintenance'
        RunAsVirtualAccount = $true
        Ensure                = 'Present'
        DependsOn             =
        '[JeaRoleCapabilities]ServiceMaintenance',
        '[JeaRoleCapabilities]ProcessMaintenance'
        # We define the role capabilities by specifying the
        # role capability Active Directory group:
        RoleDefinitions       = "
            @{
            'Contoso\PSHRemoting-APP1-ServiceMaintenance' = @{
                RoleCapabilities = 'ServiceMaintenance'}
            'Contoso\PSHRemoting-APP1-ProcessMaintenance' = @{
                RoleCapabilities = 'ProcessMaintenance'}
            }
        "
    }
```

- Use a DSC Compiler to interpolate the configuration for each server. Interpolating DSC configuration into each node reduces node management complexity and enables different session configurations and role capabilities to be modular and reusable within different configurations. *Datum*[21] is a DSC Compiler that uses YAML configuration to link DSC configurations.
- When defining the WSMan configuration, Kerberos is considered the best authentication mechanism, providing mutual end-to-end authentication, as well as encryption. NTLM is considered the second-best authentication method. However, it doesn't support mutual authentication.
- When defining a PowerShell session configuration, consider the security *use cases* for that configuration and simplify against each *use case*.
- Enforce the use of the PowerShell Language Mode. Always select the most restrictive language mode possible without compromising on usability. The language modes are (from most to least restrictive):

1. **No Language**: PowerShell permits no script text of any form.
2. **Restricted Language**: Users can run commands but can't use script blocks.
3. **Constrained Language**: See the Constrained Language Mode chapter.
4. **Full Language**: The default language mode. All features are available.

18.4.5 Managing PowerShell Session Configurations

When DSC manages the PowerShell session configuration, ongoing changes to the PowerShell session configuration aren't required. You can nevertheless query the session configuration using `Get-PSSessionConfiguration` and configure it using `Set-PSSessionConfiguration`.

You can use `Get-PSSessionConfiguration` to retrieve information about session configurations:

Example 11: Retrieving a list of registered session configurations

```
Get-PSSessionConfiguration
```

[21] https://github.com/gaelcolas/datum

```
Name          : microsoft.powershell
PSVersion     : 5.1
StartupScript :
RunAsUser     :
Permission    : NT AUTHORITY\NETWORK AccessDenied,
                NT AUTHORITY\INTERACTIVE AccessAllowed, BUILTIN\Administrators
                AccessAllowed, BUILTIN\Remote Management Users AccessAllowed

Name          : microsoft.powershell.workflow
PSVersion     : 5.1
StartupScript :
RunAsUser     :
Permission    : NT AUTHORITY\NETWORK AccessDenied,
                BUILTIN\Administrators AccessAllowed, BUILTIN\Remote Management
                Users AccessAllowed

Name          : microsoft.powershell32
PSVersion     : 5.1
StartupScript :
RunAsUser     :
Permission    : NT AUTHORITY\NETWORK AccessDenied,
                NT AUTHORITY\INTERACTIVE AccessAllowed, BUILTIN\Administrators
                AccessAllowed, BUILTIN\Remote Management Users AccessAllowed
```

You can view the entire object structure by piping the results into `Format-List`. The example below shows the first registered session configuration and its properties.

> For the sake of readability, this example excludes some object properties.

Example 12: Displaying all properties of a single session configuration

```
Get-PSSessionConfiguration | Select-Object -First 1 | Format-List *
```

```
RunAsPassword                 :
Capability                    : {Shell}
PSVersion                     : 5.1
AutoRestart                   : false
ExactMatch                    : False
RunAsVirtualAccount           : false
...
MaxShells                     : 2147483647
SupportsOptions               : true
lang                          : en-US
MaxIdleTimeoutms              : 2147483647
Enabled                       : true
...
Name                          : microsoft.powershell
XmlRenderingType              : text
...
```

You can change PowerShell session configuration properties using `Set-PSSessionConfiguration` with the `-Name` parameter.

Example 13: Adding a startup script to the `microsoft.powershell` session configuration

```
# Startup script
Set-Content -Value 'Write-Host "Startup!"' -Path 'C:\Temp\Startup.ps1'

# Implementing the change with PowerShell console
Get-PSSessionConfiguration -Name microsoft.powershell |
    Set-PSSessionConfiguration -StartupScript C:\TEMP\Startup.ps1
```

```
WARNING: Set-PSSessionConfiguration may need to restart the WinRM service if
a configuration using this name has recently been unregistered, certain system
data structures may still be cached. In that case, a restart of WinRM may be
required. All WinRM sessions connected to Windows PowerShell session
configurations, such as Microsoft.PowerShell and session configurations that
are created with the Register-PSSessionConfiguration cmdlet, are disconnected.

   WSManConfig: Microsoft.WSMan.Management\WSMan::localhost\Plugin\
   microsoft.powershell\InitializationParameters

ParamName                      ParamValue
---------                      ----------
startupscript                  C:\TEMP\Startup.ps1
```

Example 14: Entering a PSRemoting session with the updated configuration

```
Enter-PSSession -ComputerName . -ConfigurationName 'microsoft.powershell'
```

```
Startup!
[localhost]: PS C:\>
```

There are limitations to updating the existing session configurations, as some parameters aren't present within the cmdlet. See the parameter listing below:[22]

```
Set-PSSessionConfiguration
   [-Name] <String>
   [-RunAsCredential <PSCredential>]
   [-ThreadOptions <PSThreadOptions>]
   [-AccessMode <PSSessionConfigurationAccessMode>]
   [-UseSharedProcess]
   [-StartupScript <String>]
   [-MaximumReceivedDataSizePerCommandMB <Double>]
   [-MaximumReceivedObjectSizeMB <Double>]
   [-SecurityDescriptorSddl <String>]
   [-ShowSecurityDescriptorUI]
   [-Force]
   [-NoServiceRestart]
   [-TransportOption <PSTransportOption>]
   -Path <String>
   [-WhatIf]
   [-Confirm]
   [<CommonParameters>]
```

To change other properties, you should update the `.pssc` file generated by `New-PSSessionConfiguration` and then use the `-Path` parameter with `Set-PSSessionConfiguration`.

[22] Microsoft. (2021, Sep. 27). *Set-PSSessionConfiguration (Microsoft.PowerShell.Core)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/set-pssessionconfiguration. [Accessed: Oct. 14, 2021].

```
# File path: C:\Temp\TempPowerShellSession.pssc
@{
    # Version number of the schema used for this document
    SchemaVersion = '2.0.0.0'

    # ID used to uniquely identify this document
    GUID = '109fee56-540e-4688-801f-a390842355b4'

    # Author of this document
    # Updated the Author
    Author = 'Michael Zanatta'
}
```

Update the PowerShell session configuration:

Example 15: Updating a session configuration from file

```
Set-PSSessionConfiguration -Path 'C:\Temp\TempPowerShellSession.pssc'
```

When you change the PowerShell session configuration, you'll see a warning that the WinRM service will restart. This will disconnect any existing PowerShell remoting sessions. To prevent the service restart, use the `-NoServiceRestart` parameter. To suppress any confirmation of the service restart, use the `-Force` parameter.

At the time of writing, the `-NoServiceRestart` parameter *doesn't guarantee* that the WinRM service won't restart. As a rule of thumb, implicitly assume that the service will restart.

18.5 An Overview of the Security Descriptor Definition Language (SDDL)

18.5.1 Terms

- **Security Descriptor Definition Language** (SDDL): A string representation of a Security Descriptor.
- **Security Descriptor**: A data structure of security information about a securable object, including its owner, group, DACL, and SACL.
- **Securable Object**: Any object or resource, such as a file, process, or event, that can have a Security Descriptor.
- **Discretionary Access Control List** (DACL): Identifies a list of trustees (access control entries) that will be allowed or denied access to a securable object.
- **System Access Control List** (SACL): Identifies a list of trustees (access control entries) that the system audits during a successful or failed access to a securable object.
- **Trustee**: A user account, group, or logon session to which an ACE applies.

- **Access Control List** (ACL): A list of ACEs that together define the access rights of a securable object in a DACL or SACL.
- **Access Control Entry** (ACE): A single entry that represents specific access rights of a securable object for a trustee.
- **Security Identifier** (SID): An immutable identifier that represents a trustee.

18.5.2 SDDL Overview

The Security Descriptor Definition Language (SDDL) is a format that defines security descriptors as *Strings*. SDDLs within a PowerShell session configuration are used to define which users can connect to the PowerShell remoting session using the `AccessAllowed` and `AccessDenied` ACE Flags. When you add role capabilities within a PowerShell session configuration, the system adds the associated AD groups to the `AccessAllowed` DACL within the SDDL string.

When viewing registered session configurations, PowerShell automatically formats the permissions into readable text.

Example 16: Displaying formatted permissions for the *JeaEndpoint* session configuration

```
Get-PSSessionConfiguration
```

```
Name          : JeaEndpoint
PSVersion     : 5.1
StartupScript :
RunAsUser     :
Permission    : CONTOSO\ServiceMaintenanceCapability AccessAllowed
```

However, reviewing the permission configuration in the WSMan PSDrive reveals the SDDL format:

Example 17: Equivalent raw SDDL permissions for the *JeaEndpoint* session configuration

```
   WSManConfig: Microsoft.WSMan.Management\WSMan::localhost\
                Plugin\JEAMaintenance\Resources\Resource_417209259\
                Security\Security_5898658

Key:   Uri
Value: http://schemas.microsoft.com/powershell/JEAMaintenance

Key:   Sddl
Value: O:NSG:BAD:P(D;;GA;;;NU)
(A;;GA;;;S-1-5-21-1769934282-1541694284-2448955753-1106)
(A;;GA;;S-1-5-21-1769934282-1541694284-2448955753-1107)S:P(AU;FA;GA;;;WD)
(AU;SA;GXGW;;;WD)

Key:   ExactMatch
Value: False

Key:   xmlns
Value: http://schemas.microsoft.com/wbem/wsman/1/config/PluginConfiguration

Key:   ParentResourceUri
Value: http://schemas.microsoft.com/powershell/JEAMaintenance
```

18.5.3 SDDL Syntax

SDDLs follow a specific syntax: `O:<Owner>G:<Primary Group>D:<DACL>:S<SACL>` for each entry. A colon (`:`) separates each part of the security descriptor.

Raw SDDL:

```
O:NSG:BAD:P(D;;GA;;;NU)(A;;GA;;;S-1-5-21-1769934282-1541694284-2448955753-1106)
(A;;GA;;S-1-5-21-1769934282-1541694284-2448955753-1107)S:P(AU;FA;GA;;;WD)
(AU;SA;GXGW;;;WD)
```

Constituent parts of the SDDL:

```
# Owner (O:owner_sid)
O:NS

# Primary group (G:group_sid)
G:BA

# DACL (D:dacl_flags(string_ace1)(string_ace2)...)
D:P(D;;GA;;;NU)(A;;GA;;;S-1-5-21-1769934282-1541694284-2448955753-1106)
(A;;GA;;S-1-5-21-1769934282-1541694284-2448955753-1107)

# SACL (S:sacl_flags(string_ace1)(string_ace2)...)
S:P(AU;FA;GA;;;WD)(AU;SA;GXGW;;;WD)
```

The four parts of the SDDL syntax are:

1. **`O:` The object's owner**: The owner of the securable object, as a trustee SID or well-known SID string constant. SID string constants include:

    - "**NS**" : Network Service.
    - "**BA**" : Built-in Administrators.
    - "**BG**" : Built-in Guests.
    - "**WD**" : Everyone.
    - "**SY**" : Local System.
    - "**AU**" : Authenticated Users.

2. **`G:` The object's primary group**. Primary groups are a legacy property retained for backward compatibility, and PowerShell session configuration doesn't use them.
3. **`D:` The DACL**. Defined permissions that determine the access to the resource, with the following syntax: `dacl_flag(string_ace1)(string_ace2)....`

    - `dacl_flag`: Inheritance Control Flags that apply to the DACL. Flags can be:

        - "**P**" : Block inheritance from containers that are higher in the hierarchy.
        - "**AR**" : Allow inheritance.
        - "**AI**" : Child objects inherit permissions.

- `(string_ace1)(string_ace2)...`: ACEs. Each ACE wrapped in parentheses ‘`(string_ace1)`’.

4. **`S:` The SACL**. Audit permissions that determine auditing on the resource, using the following syntax: `sacl_flag(string_ace1)(string_ace2)...`. SACLs have the same syntax and control flags as the `dacl_flags`.

For a complete list of trustee SID strings, see SID Strings[23]. For a complete list of ACE strings/flags, see ACE Strings[24].

ACE strings contain several components:

- **ACE Type**: A value that defines the ACE type. Possible values are:
  - “**A**” : Access Allowed.
  - “**D**” : Access Denied.
  - “**AU**” : Audit.
- **ACE Flags**: Controls the inheritance and auditing behavior.[25] Possible values are:
  - “**CI**” : Container Inherit.
  - “**OI**” : Object Inherit.
  - “**SA**” : Audit Success.
  - “**FA**” : Audit Failure.
- **Rights**: Controls the standard, specific, and generic rights.[26] Possible values are:
  - “**GA**” : All Generic Access.
  - “**GR**” : Read Access.
  - “**RC**” : Read/Control Standard Access.
  - “**SD**” : Delete.

  You can combine multiple flags by appending them. For example, combining all the flags from this table would produce `GAGRRCSD`.
- **Account SID**: The target SID.
- **Object GUID**: Not used within PowerShell session configuration.
- **Inherit Object GUID**: Not used within PowerShell session configuration.
- **Resource Attribute**: Not used within PowerShell session configuration.

[23] https://learn.microsoft.com/en-us/windows/win32/secauthz/sid-strings

[24] https://learn.microsoft.com/en-us/windows/win32/secauthz/ace-strings

[25] Microsoft. (2021, Sep. 24). *AceFlags Enum (System.Security.AccessControl)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/api/system.security.accesscontrol.aceflags. [Accessed: Oct. 14, 2021].

[26] Microsoft. (2021, Sep. 01). *ACCESS_MASK (Winnt.h)*. Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/secauthz/access-mask. [Accessed: Oct. 14, 2021].

18.5.4 Reading SDDLs

You can test and review the SDDLs parsed into PowerShell session configuration files by using `ConvertFrom-SddlString` in PowerShell to convert it into a readable `[PSObject]`:

`ConvertFrom-SddlString` is only available on Windows.

Example 18: Converting a raw SDDL string to formatted permissions

```
# Show the SDDL Value
$SDDL.Value

# Converting it to a PowerShell Object
$SDDL.Value | ConvertFrom-SddlString
```

```
O:NSG:BAD:P(D;;GA;;;NU)(A;;GA;;;S-1-5-21-1769934282-1541694284-2448955753-1106)
(A;;GA;;;S-1-5-21-1769934282-1541694284-2448955753-1107)S:P(AU;FA;GA;;;WD)
(AU;SA;GXGW;;;WD)

Owner            : NT AUTHORITY\NETWORK SERVICE
Group            : BUILTIN\Administrators
DiscretionaryAcl : {
                        NT AUTHORITY\NETWORK: AccessDenied (GenericAll),
                        CONTOSO\ServiceMaintenanceCapability: AccessAllowed
                                    (GenericAll),
                        CONTOSO\ProcessMaintenanceCapability: AccessAllowed
                                    (GenericAll)
                   }
SystemAcl        :  {
                        Everyone: SystemAudit FailedAccess (GenericAll),
                        Everyone: SystemAudit SuccessfulAccess (GenericExecute,
                                    GenericWrite)
                    }
RawDescriptor    : System.Security.AccessControl.CommonSecurityDescriptor
```

To make adjustments to existing SDDLs, parse the SDDL string into `[RawSecurityDescriptor]`:

Example 19: Modifying an SDDL string by conversion to a `[RawSecurityDescriptor]`

```
$SecurityDescriptor =
    [System.Security.AccessControl.RawSecurityDescriptor]::new(
        'SDDL STRING'
    )
# Make adjustments

# Export back into SDDL string format
$SecurityDescriptor.GetSddlForm(
    [System.Security.AccessControl.AccessControlSections]::All
)
```

18.5.5 Creating SDDLs from a Security Descriptor

Creating SDDLs from a security descriptor is possible by using the `[RawSecurityDescriptor]` class. The process of creating a security descriptor is as follows:

1. **Define the *owner***: Instantiate the `[SecurityIdentifier]` class with a Predefined Trustee or a SID:

```
$Owner = [System.Security.Principal.SecurityIdentifier]::new("NS")
```

2. **Define the *primary group***: Using the same class in the last step, create an object with the predefined trustee or SID:

```
$PrimaryGroup = [System.Security.Principal.SecurityIdentifier]::new("BA")
```

3. **Create the Discretionary ACL**: Create an empty Discretionary ACL object (*using* `[RawAcl]`):

```
$DiscretionaryACL = [System.Security.AccessControl.RawAcl]::new(0,0)
```

4. **Create the ACEs for the DACL**: Create an empty `[ObjectAce]` ACE object, with:

   1. A qualifier: `[AceQualifier]`
   2. A security identifier (a trustee): `[SecurityIdentifier]`

   For example:

```
$EveroneAllowedACE = [System.Security.AccessControl.ObjectAce]::new(
    [System.Security.AccessControl.AceFlags]::None,
    # Define a Qualifier. In this case, we will allow Access
    [System.Security.AccessControl.AceQualifier]::AccessAllowed,
    1,
    # Define a SecurityIdentifier:
    # 'WD' is the shorthand version of Everyone
    [System.Security.Principal.SecurityIdentifier]::new('WD'),
    [System.Security.AccessControl.ObjectAceFlags]::None,
    [System.Guid]::NewGuid(),
    [System.Guid]::NewGuid(),
    $false,
    $null
)
```

5. **Add the ACE to the Discretionary ACL**:

```
$index = 0
# Add to the ACL
$DiscretionaryACL.InsertAce($index, $EveroneAllowedACE)
```

   To add more ACEs, increment the `$index` variable.

6. **Create the System ACL**: Create an empty System ACL object (*using* `[RawAcl]`):

```
$SystemACL = [System.Security.AccessControl.RawAcl]::new(0,0)
```

7. **Create the ACEs for the SACL**: Create an empty `[ObjectAce]` ACE object, with:

    1. A flag (`AceFlags`) defining the audit type.
    2. A qualifier: `[AceQualifier]`. The example will use the `SystemAudit` enum.
    3. A security identifier (a trustee): `[SecurityIdentifier]`.

    For example:

```
# Insert ACE into SystemACL
$EveroneAuditACE = [System.Security.AccessControl.ObjectAce]::new(
    # Define a Flag
    [System.Security.AccessControl.AceFlags]::SuccessfulAccess,
    # Define a Qualifier
    [System.Security.AccessControl.AceQualifier]::SystemAudit,
    1,
    # Define a Security Identifier
    # 'WD' is the shorthand version of Everyone
    [System.Security.Principal.SecurityIdentifier]::new('WD'),
    [System.Security.AccessControl.ObjectAceFlags]::None,
    [System.Guid]::NewGuid(),
    [System.Guid]::NewGuid(),
    $false,
    $null
)
```

8. **Add the ACE to System ACL**:

```
$index = 0
# Add to the ACL
$SystemACL.InsertAce($index, $EveroneAuditACE)
```

    To add more ACEs, increment the `$index` variable.

9. **Define control flags** for the security descriptor to determine the descriptor behavior. A PowerShell session configuration security descriptor contains:

```
$ControlFlags = [System.Security.AccessControl.ControlFlags]

$Flags = @(
    # Specifies that the DACL is not null.
    # Set by resource managers or users.
    $ControlFlags::DiscretionaryAclPresent

    # Specifies that the SACL is not null.
    # Set by resource managers or users.
    $ControlFlags::SystemAclPresent

    # Specifies that the resource manager prevents auto-inheritance.
    # Set by resource managers or users.
    $ControlFlags::DiscretionaryAclProtected

    #Specifies that the resource manager prevents auto-inheritance.
    #Set by resource managers or users.
    $ControlFlags::SystemAclProtected

    # Specifies that the security descriptor binary representation
    # is in the self-relative format. This flag is always set.
    $ControlFlags::SelfRelative

)
```

For a complete list of control flags, refer to the `ControlFlags`[27] .NET class documentation.

10. **Create the security descriptor** and export using the `GetSddlForm()` method.

```
# Create the Secruity Descriptor, parsing in all the previous variables.
$SecurityDescriptor =
    [System.Security.AccessControl.RawSecurityDescriptor]::new(
        $Flags, $Owner, $PrimaryGroup, $SystemACL, $DiscretionaryACL
    )
# Now that the security descriptor exists,
# convert it to SDDL by calling GetSddlForm()
$SecurityDescriptor.GetSddlForm(
    [System.Security.AccessControl.AccessControlSections]::All)
```

Full example:

Example 20: Building an SDDL string from scratch

```
# Step 1: Create the owner

# Alternatively use an SID
# $Owner = [System.Security.Principal.SecurityIdentifier]::new
#                       ('S-1-5-21-2274662803-2033504868-1650952085-500')

# In this case we will use a prefedined trustee
# NS being the SID string constant for 'Network Security'
$Owner = [System.Security.Principal.SecurityIdentifier]::new("NS")

# Step 2: Primary group
# BA being the SID string constant for 'Built-in Administrators'
$PrimaryGroup = [System.Security.Principal.SecurityIdentifier]::new("BA")

# Step 3: Create the Discretionary ACL
$DiscretionaryACL = [System.Security.AccessControl.RawAcl]::new(0,0)

# Step 4: Create ACE
$EveroneAllowedACE = [System.Security.AccessControl.ObjectAce]::new(
    [System.Security.AccessControl.AceFlags]::None,
    [System.Security.AccessControl.AceQualifier]::AccessAllowed,
    1,
    # 'WD' is an SID string constant for 'Everyone'
    [System.Security.Principal.SecurityIdentifier]::new('WD'),
    [System.Security.AccessControl.ObjectAceFlags]::None,
    [System.Guid]::NewGuid(),
    [System.Guid]::NewGuid(),
    $false,
    $null
)

# Step 5: Add the ACE into the ACL
$index = 0

# Add to the ACL
$DiscretionaryACL.InsertAce($index, $EveroneAllowedACE)

# Step 6: Create the System ACL
$SystemACL = [System.Security.AccessControl.RawAcl]::new(0,0)
```

[27] https://learn.microsoft.com/en-us/dotnet/api/system.security.accesscontrol.controlflags

```

# Step 7: Create an ACE for SystemACL
$EveroneAuditACE = [System.Security.AccessControl.ObjectAce]::new(
    [System.Security.AccessControl.AceFlags]::SuccessfulAccess,
    [System.Security.AccessControl.AceQualifier]::SystemAudit,
    1,
    # 'WD' is an SID string constant for 'Everyone'
    [System.Security.Principal.SecurityIdentifier]::new('WD'),
    [System.Security.AccessControl.ObjectAceFlags]::None,
    [System.Guid]::NewGuid(),
    [System.Guid]::NewGuid(),
    $false,
    $null
)

# Step 8: Add ACE into SystemACL
$index = 0

# Add to the ACL
$SystemACL.InsertAce($index, $EveroneAuditACE)

# Step 9: Define the control flags for the security descriptor

# Within PowerShell remoting we will set the Protected flag ('P') for
# the SystemACL and DiscretionaryACL
$ControlFlags = [System.Security.AccessControl.ControlFlags]

$Flags = @(
    $ControlFlags::DiscretionaryAclPresent
    $ControlFlags::SystemAclPresent
    $ControlFlags::DiscretionaryAclProtected
    $ControlFlags::SystemAclProtected
    $ControlFlags::SelfRelative
)

# Step 10: Create the security descriptor and
# convert the object to the SDDL format.
$SecurityDescriptor =
    [System.Security.AccessControl.RawSecurityDescriptor]::new(
        $Flags, $Owner, $PrimaryGroup, $SystemACL,$DiscretionaryACL
    )
$SecurityDescriptor.GetSddlForm(
    [System.Security.AccessControl.AccessControlSections]::All
)
```

```
O:NSG:BAD:P(OA;;CC;;;WD)S:P(OU;SA;CC;;;WD)
```

18.6 Auditing PowerShell Remoting Sessions

While JEA session configuration is a capable security solution for PowerShell remoting, JEA also has several monitoring features which allow you to manage ongoing connections within those environments. They are:

1. Reviewing effective rights
2. PowerShell Event Logs
3. Session transcription logs

18.6.1 Review Effective Rights

Troubleshooting user access can be cumbersome, especially with the complexities of reviewing PowerShell session configurations. The `Get-PSSessionCapability` cmdlet enumerates all the commands available from the PowerShell remoting endpoint for a specific user.

Example 21: Retrieving all available commands in the current PSRemoting session

```
$PSSessionCapabilityParams = @{
    ConfigurationName = 'JEAMaintenance'
    Username = 'CONTOSO\HelpdeskUser01'
}

Get-PSSessionCapability @PSSessionCapabilityParams
```

```
CommandType  Name                        Version  Source
-----------  ----                        -------  ------
Alias        clear -> Clear-Host
Alias        cls -> Clear-Host
Alias        exsn -> Exit-PSSession
Alias        gcm -> Get-Command
Alias        measure -> Measure-Object
Alias        select -> Select-Object
Function     Clear-Host
Function     Exit-PSSession
Function     Get-Command
Function     Get-FormatData
Function     Get-Help
Function     Measure-Object
Function     Out-Default
Function     Select-Object
Cmdlet       Get-Process                 3.0.0.0  ...
Cmdlet       Get-Service                 3.0.0.0  ...
Cmdlet       Restart-Service             3.0.0.0  ...
Cmdlet       Stop-Process                3.0.0.0  ...
```

Active Directory user direct-assigned permissions may not reflect the capabilities available via `Get-PSSessionCapability`. Always use Active Directory group permissions (preferably using the RBAC Active Directory group model, using Active Directory role groups) when delegating access in the PowerShell session or the role capability.

This solution assists helpdesks and system administrators in troubleshooting issues with user access. Another use case is to provide reporting and ongoing security testing. In the example below, a Pester test validates the effective access of several users:

This example requires *Pester* 5.0 or higher.

Example 22: Validating effective access for a session configuration with Pester

```
$Params = @(
    @{
        # User1 should have no access to the server
        ADUserName = 'CONTOSO\User1'
        ExpectedCmdlets = @()
    }
    @{
        # User2 should only have Get-ChildItem
        ADUserName = 'CONTOSO\User2'
        ExpectedCmdlets = @('Get-ChildItem')
    }
    @{
        # User3 should only have Get-Service and Stop-Service
        ADUserName = 'CONTOSO\User3'
        ExpectedCmdlets = @('Get-Service', 'Stop-Service')
    }
)

Describe "Test PowerShell remoting effective access on $ENV:Computer" {

    It "Returns <ExpectedCmdlets> (<ADUserName> on <Computer>)"
                                        -ForEach $Params -Test {

        $params = @{
            ConfigurationName = 'JEAMaintenance'
            UserName = $ADUserName
        }

        $session = Get-PSSessionCapability @params |
                        Where-Object CommandType -eq 'Cmdlet'

        # Test that the commands returns remain correct.
        # No more, no less.

        ($session | Sort-Object -Property Name).Name |
                        Should -Be ($ExpectedCmdlets | Sort-Object)

    }

}
```

18.6.2 PowerShell Event Logs

PowerShell Script Block and module logging can also log each command executed within a PowerShell JEA session.

Using Group Policy to enable script execution logging:

1. *Open* **Group Policy Editor**.
2. *Browse to*: **Computer Configuration \ Administrative Templates \ Windows Components \ Windows PowerShell**
3. *Select* **Turn on PowerShell Script Block Logging**
4. *Select* **Enabled**
5. *Select* **OK**
6. *Restart* the computer to apply the Group Policy configuration

⚠ Enabling the ‘Log script block invocation start/stop events’ will generate a large number of events.

Use Computer Configuration to enable logging on all machines within the domain.

You can consolidate PowerShell event logs and transcription logs by ingesting the contents into Azure Sentinel. Azure Sentinel uses the Azure Log Analytics (OMS Agent). You can configure event log and custom log ingestion.[28] [29]

Be aware that Azure Sentinel logging isn’t free. A large number of events can lead to a large charge for storage space.

18.6.3 Session Transcription Logs

PowerShell session configuration provides transcription logging of PowerShell sessions, enabling you to analyze ongoing PowerShell remoting session configuration. When implementing transcription, always consolidate all transcription locations into a single restricted staging directory, pending upload to Azure Log Analytics/Sentinel.

18.6.4 Removing Existing PowerShell Sessions

While JEA is a valuable tool that enables you to restrict users/applications to predefined role capabilities, PowerShell leaves the default session configurations enabled by default. If there is no requirement for the default session configurations to be available, use DSC to disable them.

The script resource shown below uses `Get-PSSessionConfiguration -Name microsoft.*` to enumerate the default Microsoft configurations and pipes the result into `Disable-PSSessionConfiguration`.

[28] Microsoft. (2021, Jun. 09). *Collect Windows event log data sources with Log Analytics agent.* Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-monitor/agents/data-sources-windows-events. [Accessed: Oct. 14, 2021].

[29] Microsoft. (2021, Aug. 10). *Overview of Azure Monitor agents.* Microsoft Docs. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-monitor/agents/agents-overview. [Accessed: Oct. 14, 2021].

Example 23: DSC configuration removing the default session configurations

```
Configuration DisableDefaultPowerShellRemoting
{
    Import-DscResource -ModuleName 'PSDesiredStateConfiguration'
    Import-DscResource -ModuleName 'WSManDsc'

    Node $AllNodes.Where{$_.PSRemotingEnabled}.NodeName {
        Script EnablePowerShellRemoting {
            SetScript = {
                Get-PSSessionConfiguration -Name microsoft.* |
                    Disable-PSSessionConfiguration
            }
            GetScript = {
                # Fetch all OOBE default PS sessions.
                @{
                    Result = Get-PSSessionConfiguration -Name microsoft.*
                }
            }
            TestScript = {
                $state = [scriptblock]::Create($GetScript).Invoke() |
                            Where-Object Enabled -eq $true
                $state.Result -contains $true
            }
        }
    }
}
```

18.7 Further Reading

- About Session Configuration Files—Microsoft Docs[30]
- About Session Configurations—Microsoft Docs[31]
- New-PSSessionConfigurationFile—Microsoft Docs[32]
- New-PSRoleCapabilityFile—Microsoft Docs[33]
- JEA Role Capabilities—Microsoft Docs[34]
- Introducing the Updated JEA Helper Tool—Microsoft Docs[35]
- Find-RoleCapability—Microsoft Docs[36]
- Invoke-Command—Microsoft Docs[37]
- About Language Modes—Microsoft Docs[38]
- Security Descriptor String Format—Microsoft Docs[39]
- ACE Trustees—Microsoft Docs[40]
- ACE Strings—Microsoft Docs[41]

[30] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_session_configuration_files
[31] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_session_configurations
[32] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/new-pssessionconfigurationfile
[33] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/new-psrolecapabilityfile
[34] https://learn.microsoft.com/en-us/powershell/scripting/learn/remoting/jea/role-capabilities
[35] https://learn.microsoft.com/en-us/archive/blogs/privatecloud/introducing-the-updated-jea-helper-tool
[36] https://learn.microsoft.com/en-us/powershell/module/powershellget/find-rolecapability
[37] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/invoke-command
[38] https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_language_modes
[39] https://learn.microsoft.com/en-us/windows/win32/secauthz/security-descriptor-string-format
[40] https://learn.microsoft.com/en-us/windows/win32/secauthz/trustees
[41] https://learn.microsoft.com/en-us/windows/win32/secauthz/ace-strings

- SECURITY_DESCRIPTOR_CONTROL—Microsoft Docs[42]
- Authentication for Remote Connections—Microsoft Docs[43]
- WinRM Security—Microsoft Docs[44]
- Kerberos Key Distribution Center—Microsoft Docs[45]
- Credential Security Support Provider—Microsoft Docs[46]

[42] https://learn.microsoft.com/en-us/windows-hardware/drivers/ifs/security-descriptor-control
[43] https://learn.microsoft.com/en-us/windows/win32/winrm/authentication-for-remote-connections
[44] https://learn.microsoft.com/en-us/powershell/scripting/learn/remoting/winrmsecurity
[45] https://learn.microsoft.com/en-us/windows/win32/secauthn/key-distribution-center
[46] https://learn.microsoft.com/en-us/windows/win32/secauthn/credential-security-support-provider

Afterword

By Bill Kindle

This book was written during rather interesting times in the world, and with that came unique challenges.

But here we are.

Through all the difficulties and roadblocks, a new PowerShell automation book was born.

Thank you for your support on behalf of the editorial and author teams.

You have just read through a collective work of contributors from the greater PowerShell community. Unlike **The PowerShell Conference Book vol. 1-3**, we aimed to build something different. Many months were poured into these pages to bring you, the reader, a PowerShell book worthy of use in an academic setting.

You may wonder, "What was the reason for the cover art?"

Amy Zanatta designed the cover art and had this to say:

"Initially, I was approached by the Senior Editors to design a cover art image. I used the PowerShell hero image for my color selection, with textures, font, and photography added to provide a technical and professional look and feel."

And she did a bang-up job at that!

So what challenges did the team face?

We had several authors drop from the project, with pressure on the senior editorial staff to fill in the gaps. Michael Zanatta, Matt Corr, and Nicholas Bissell stepped up to write multiple chapters. Thank you! However, considering the timeframe and lifespan of the project, we decided to drop unfinished chapters to focus on releasing the book.

LeanPub continues to present challenges in how markdown is rendered. As many on the team can attest, life sometimes gets in the way. Illness, family obligations, and job changes from authors and editors created delays. However, the resiliency of the editorial and author teams shined.

What's next? A well-earned break! A second edition will be considered later, adding/updating content.

On a personal note, PowerShell was a game changer for my career. It helped me become a better sysadmin and was a gateway for beginning to write. It has been amazing to see so many people come together worldwide to work on a project and see that project become a reality. I liked the first PowerShell Conference Book Volume 1 so much that I decided to help write a chapter for Volume 2. I was then invited to become one of the editors for Volume 3. Michael Zanatta invited me to join this project in the spirit of the first three volumes.

And here I am, standing on the shoulders of giants as an editor. I would have never thought back in 2011, when I first started using PowerShell in my job, that I would be writing the afterword for Modern IT Automation with PowerShell.

Thank you. Until next time…

Index

A, SYMBOLS

B, C

D, E

F, G

H, I

J, K, L, M

N

O

P

R

S

T

U, V, W

Y