THE POWERSHELL CONFERENCE BOOK

-VOLUME 2-

EDITED BY:

PAULA KINGSLEY

MARK KRAUS

MICHAEL T. LOMBARDI

The PowerShell Conference Book

Volume 2

Michael T Lombardi and Mark Kraus

This book is for sale at http://leanpub.com/psconfbook2

This version was published on 2020-09-18

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.

Tweet This Book!

Please help Michael T Lombardi and Mark Kraus 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 #PSConfBook

The suggested hashtag for this book is #PSConfBook.

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

#PSConfBook

Also By These Authors

Books by Michael T Lombardi

The PowerShell Conference Book

Material for pwshop: A PowerShell 101 Workshop

Pentola

Beneath the Canals Collection

Books by Mark Kraus

The PowerShell Conference Book

Contents

Foreword

Back in the PowerShell version 1.0 days, I was providing technical support for three different companies who were all early adopters of Exchange Server 2007. At that point in time, a number of configuration options that needed to be changed and administrative tasks that needed to be performed couldn't be accomplished using the GUI. Incrementally learning the new features of each subsequent version of PowerShell has been manageable.

While attending conferences is not required to learn PowerShell, I can definitely speak from experience that attending the PowerShell + DevOps Global Summit every year since its inception has certainly helped my career, not only from the knowledge that I've gained but also from the connections that I've made while networking with others at the event.

In addition to the numerous new features added with each new version of PowerShell, the ability to run PowerShell on Linux and macOS only compounds the learning curve for someone just getting started. That's where you, this book, and the OnRamp scholarship program can assist. In 2018, The DevOps Collective awarded one scholarship for someone to attend the PowerShell + DevOps Global Summit in Bellevue Washington. The recipient was Andrew Pla. I remember meeting Andrew at the summit that year and his enthusiasm for learning PowerShell was contagious.

Last year, I decided to leverage the connections that I'd made over the years and create Volume 1 of The PowerShell Conference Book. All of the authors paid it forward by donating a small portion of their time by writing a chapter in an area of expertise where they're a subject matter expert. We donated 100% of the royalties from the book to the DevOps Collective OnRamp scholarship program.

This year, while at PowerShell on the River in Chattanooga Tennessee, I personally had a chance to see how the DevOps Collective OnRamp scholarship program maximizes its return on investment. Andrew Pla who was the first scholarship recipient presented a session on how to *Empower your team with ChatOps: Getting started with PoshBot*. I was amazed at his level of expertise and professionalism during the presentation. When it was over, I thought "wow, what a return on investment that was." In just a little over a year, Andrew has gone from receiving a scholarship to attend the PowerShell and DevOps Global Summit to sharing his expertise with others in the industry by presenting at conferences himself.

The authors and editors of this book along with the people who purchase it are facilitating this type of ongoing return on investment by supporting future scholarship recipients.

Mike F. Robbins[1], Creator of The PowerShell Conference Book

[1] https://mikefrobbins.com/

Contributors

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

Graham Beer

Graham is a Platform Engineer living in the south of England. He is a co-founder of a PowerShell user group in the south coast of England and a passionate user of PowerShell for 5 years. Graham previously authored a chapter in the PowerShell Conference Book on 'Extending PowerShell with TypeData Programming' and has contributed many PowerShell articles achieving a spot among the Top 50 Bloggers of 2018[2]. He blogs at graham-beer.github.io/[3] and can be found on twitter @GKBeer[4].

Michael Bender

Michael is a teacher and community leader focused on IT Pros and Operations. Currently, he builds content for learners at Pluralsight.com[5]. You can follow him on Twitter[6] or keep up with his blog[7].

Jacob Benson

Jacob is passionate about how technology empowers individuals to create solutions and solve problems that improve the quality of their lives and the lives of other people. Recently Jacob has been focused on how he can help other people be better humans at and outside of work. You can connect with him on LinkedIn[8] or keep up with his blog[9].

Jessykah Bird

Jessykah is always pursing new methods and concepts around technologies both large and small, and sharing those experiences around the world in both talks and hands-on technical training. With a passion for computational linguistics, she pursues creative methods for human interaction with technology in everyone's day to day lives.

[2] https://www.sqlshack.com/top-50-powershell-bloggers-of-2018/
[3] https://graham-beer.github.io/
[4] https://twitter.com/GKBeer
[5] https://www.pluralsight.com/authors/michael-bender
[6] https://twitter.com/MichaelBender
[7] https://itsallgeek2mike.com
[8] https://linkedin.com/in/jacobbenson
[9] https://jacobbenson.io

Phil Bossman

Phil is an accomplished Windows administrator and is a Citrix architect for a national building materials supplier. Phil has been in the technology industry for over 25 years and has a passion for technology, automation, and learning all things PowerShell. Phil enjoys sharing his wealth of knowledge on PowerShell in ways everyone can relate to and understand. Phil is a co-organizer of the Research Triangle PowerShell User Group[10]. Phil is an active member of the Citrix User Group Community (CUGC)[11]. You can follow him on Twitter[12] and Github[13], or check out his blog[14].

Dave Carroll

Dave has spent more than two decades in IT in various capacities with significant focus on systems administration. For most of this time, he has developed automation through multiple scripting languages on several platforms. Currently, he focuses on Active Directory and identity management. He writes about PowerShell (and other IT related topics) on his blog[15]. You can find him on Twitter @thedavecarroll[16] and Github[17].

Ryan Coates

Ryan has worked in IT since he was 14, and is currently an Enterprise Architect leading API strategy at a large consulting firm. Ryan speaks regularly at national and regional conferences focusing on Cloud and Automation, is frequently active on Twitter[18], and LinkedIn[19], and less frequently on his Blog JustDeclareIt[20].

Tim Curwick

Tim Curwick is an automation consultant, trainer, and speaker with a passion for PowerShell-based automation. He blogs as MadWithPowerShell[21], tweets @MadWPowerShell[22], and speaks at many venues, most frequently as a long-time leader of the Minnesota PowerShell Automation (User) Group[23].

[10] https://rtpsug.com
[11] https://www.mycugc.org
[12] https://twitter.com/Schlauge
[13] https://github.com/pbossman
[14] https://schlauge.com
[15] https://powershell.anovelidea.org
[16] https://twitter.com/thedavecarroll
[17] https://github.com/thedavecarroll
[18] https://twitter.com/ryandcoates
[19] https://linkedin.com/in/ryandcoates
[20] http://justdeclareit.com
[21] https://MadWithPowerShell.com
[22] https://Twitter.com/MadWPowerShell
[23] https://MeetUp.com/Twin-Cities-PowerShell-User-Group

Mateusz Czerniawski

Mateusz is a System Architect at Objectivity[24] who grew his skills in PowerShell on Hyper-V, Active Directory, and everything related. He is also a co-founder of Polish PowerShell User Group[25] and a speaker. His greatest achievement so far is bringing more people to PowerShell community every day. 'But if I can help somebody as I pass along, then my living will shall not be in vain'. You can reach him via his blog[26].

Luc Dekens

Luc is a fan of all things automation, and an early adaptor of PowerShell and VMware PowerCLI. He is a co-author of the PowerCLI Reference[27], a regular speaker at international conferences, and a keen contributor to the VMware Community forum VMTN. He is present on Twitter as LucD22[28], you can read his blog posts at LucD notes[29] or read his community contributions on the VMTN Community[30].

Matthew Dowst

Matthew Dowst is passionate about all things DevOps and automation, with a strong focus on PowerShell and Azure Automation. He works as an automation consultant and is the lead architect on the Catapult Systems Launch automation solution. You can follow him on Twitter[31] or check out his blog[32] where he provides tons of PowerShell snippets, and a weekly round of all things PowerShell.

Saggie Haim

Saggie is an IT Platforms admin who specializes in Microsoft solutions like AD, SCCM, SCOM, and more. He writes about PowerShell in his blog Saggiehaim Blog[33] and teaches MCSA and cybersecurity in local academies.

Jeff Hicks

Jeff is a veteran IT Pro with a long history in the PowerShell community as an author, teacher and conference speaker. You can follow him on Twitter[34] or keep up with his blog[35].

[24] https://www.objectivity.co.uk
[25] https://www.meetup.com/Polish-PowerShell-Group-PPoSh
[26] https://www.mczerniawski.pl
[27] https://www.wiley.com/en-us/VMware+vSphere+PowerCLI+Reference%3A+Automating+vSphere+Administration%2C+2nd+Edition-p-9781118925119
[28] https://twitter.com/LucD22
[29] http://lucd.info
[30] https://communities.vmware.com/people/LucD/activity
[31] https://twitter.com/mdowst
[32] https://www.dowst.dev/
[33] https://www.saggiehaim.net
[34] https://twitter.com/JeffHicks
[35] https://jdhitsolutions.com/blog

Alexander Holmeset

Alexander is a Microsoft MVP, blogger and international speaker. In his day to day work and community contributions he has a huge focus on automation through PowerShell. You can follow him on Twitter at @AlexHolmeset[36] and read his blog[37]. 'If you are going to do something more than once, then automate it!'

Lawrence Hwang

Lawrence Hwang loves PowerShell and the philosophy that sparked PowerShell. He equally loves putting lego blocks together to create something cool and sometimes useful. He thinks PowerShell is a beautiful block that can bridge different realms together. Lawrence is active on Twitter as CPoweredLion@[38].

Mike Kanakos

Mike is a sysadmin and Active Directory Engineer located in Apex, North Carolina. Mike is also the co-leader of the Research Triangle PowerShell users group[39] and active blogger. He focuses on writing, teaching PowerShell fundamentals and sharing tools with the community to make the sysadmins life easier. You can follow Mike's blog at www.networkadm.in[40] and or on Twitter at @MikeKanakos[41].

Bill Kindle

Bill is a sysadmin with 15 years experience in IT currently working at IDEMIA[42], focusing on automation with PowerShell scripts. He is a husband to a wonderful woman that he doesn't deserve and father of two adorable children. Being passionate about PowerShell and various other technology, he also is a content contributor for TechSnips LLC[43].

Josh King

Geek, Father, Walking Helpdesk. Josh is a systems administrator at The Instillery South[44] in regional New Zealand. The bulk of his time is spent in Windows and VMware environments and he has a passion for PowerShell and automation. You can find him online through his blog[45] or on Twitter[46].

[36] https://twitter.com/AlexHolmeset
[37] https://alexholmeset.blog/
[38] https://twitter.com/CPoweredLion
[39] https://rtpsug.com
[40] https://networkadm.in
[41] https://twitter.com/MikeKanakos
[42] https://idemia.com
[43] https://techsnips.io
[44] https://theinstillery.com/
[45] https://toastit.dev
[46] https://twitter.com/WindosNZ

Przemysław Kłys

Przemek is a System Architect with over 15 years of experience. His skill set revolves around Active Directory, Exchange and Office 365. He is avid in PowerShell on everything that's Microsoft related. All his PowerShell modules are downloadable from PowerShellGallery[47] and are available on GitHub[48]. You can follow him on Twitter[49] or keep up with his blog[50].

Cory Knox

Cory Knox is a Deskside support specialist who is passionate about PowerShell. He has spent the last decade looking for ways to automate common tasks with PowerShell. He can be found on Twitter[51] and his Blog[52]. He's active in the PowerShell community Slack[53] and Discord[54] servers. When not working, he enjoys spending time at the lake with his wife and three children. When not spending time at the lake, he can be found on Twitch[55] where he works on the PowerShell extension for VSCode[56].

Mark Kraus

Mark is a Staff Systems Engineer at LinkedIn[57]. He is also a former Microsoft MVP[58], a DevOps and PowerShell Blogger[59], and a Twitch coding streamer[60]. He is the creator of PowerShell Live[61], a PowerShell contributor, and an open source maintainer/developer[62]. You can follow his escapades on twitter @markekraus[63].

Manoj Krishnasamy

Manoj is a PowerShell Fanatic who always find ways to do automation tasks in PowerShell. He loves to write code and solve problems. You can find him on Twitter @manojprabhuk716[64].

[47] https://www.powershellgallery.com/profiles/Przemyslaw.Klys
[48] https://github.com/EvotecIT/
[49] https://twitter.com/PrzemyslawKlys
[50] https://evotec.xyz/hub
[51] https://twitter.com/coryknox
[52] https://knoxy.ca
[53] http://aka.ms/psslack
[54] https://aka.ms/psdiscord
[55] https://twitch.tv/corbob
[56] https://github.com/powershell/vscode-powershell
[57] https://www.linkedin.com/in/markekraus/
[58] https://mvp.microsoft.com/en-us/PublicProfile/5002898?fullName=Mark%20%20Kraus
[59] https://get-powershellblog.blogspot.com
[60] https://www.twitch.tv/markekraus/
[61] https://www.twitch.tv/powershelllive
[62] https://github.com/markekraus/
[63] https://twitter.com/markekraus/
[64] https://twitter.com/manojprabhuk716

Adil Leghari

Adil Leghari is a 15-year Sysadmin who is super-passionate about PowerShell and automation. He can be found on Twitter[65] and his Blog[66]. He's active in the PowerShell community Slack[67] and Discord[68] servers. When not working, he enjoys designing stickers, long walks on the beach, and candlelit dinners.

Michael Lombardi

Mike is a sysadmin-turned-automation-engineer-turned-software-engineer at Puppet[69], passionate about documentation and restorative justice. He is the founder and co-organizer of the St. Louis PowerShell User Group[70] and cohost of the PSPowerHour[71].

Jon M. Junell

Jon leads the Infrastructure and Identity Team at Western Washington University[72]. He's really fascinated with the structures and processes around successful teams and organizations. He enjoys 2nd breakfasts, being outdoors, with a focus on hiking, mountain biking and the World Rally Championship. You can find him on Twitter[73] and LinkedIn[74].

Jeremy Murrah

Jeremy Murrah is an old operations guy from the dark ages of computing. He started automating NT 4.0 installations and hasn't looked back. Classically trained as an Active Directory Administrator and Windows Engineer, he is currently engrossed in all things PowerShell and is eagerly awaiting the death of the GUI. He blogs at murrahjm.github.io[75] and can be found on twitter @JeremyMurrah[76].

Francisco Navarro

Francisco (Cisco) is an Enterprise Consultant focusing on Cloud and Datacenter technologies, with a passion for PowerShell and enthusiasm for automation. He is the co-organizer of the

[65] https://twitter.com/adilio
[66] https://adilio.ca
[67] http://aka.ms/psslack
[68] https://aka.ms/psdiscord
[69] https://puppet.com
[70] https://meetup.com/stlpsug
[71] https://www.youtube.com/channel/UCtHKcGei3EjxBNYQCFZ3WNQ
[72] https://www.wwu.edu
[73] https://twitter.com/jonhikes
[74] https://www.linkedin.com/in/jonjunell/
[75] https://murrahjm.github.io
[76] https://twitter.com/JeremyMurrah

Chicago PowerShell User Group[77] Twitter[78] and a contributor on Techsnips[79]. Francisco prides himself the most on being a loving husband and proud father of his "mini me!" You can find Cisco on the internet through Twitter @ctmcisco[80].

Will Nevis

Will is a former chemical engineer who now does DevOps automation for LinkedIn[81]. He has been working with PowerShell since 2012 and cut his teeth at Tesla right before the release of the Model S. He owes his career to Jeffrey Snover.

Greg Onstot

Husband, Father, Contributor to the PSADHealth Module. Greg has been working in IT Infrastructure and InfoSec Engineering for over 20 years. PowerShell has become an integral part of automating that work.

Andrew Pla

Andrew Pla is a Systems Engineer, blogger, and co-founder of the Gainesville PowerShell User Group[82]. His PowerShell journey was thrust into high gear after receiving the first PowerShell + DevOps Global Summit scholarship in 2018. Since then, he has been automating everything with PowerShell and writing about his experience on his blog[83]. You can find Andrew on Twitter at @AndrewPlaTech[84] and Github[85].

James Pogran

James is a programmer at Puppet[86], co-founder of the Puppet Visual Studio Code Extension[87], and author of Learning PowerShell DSC 2nd Ed[88]. You can find James at @ender2025[89], Github[90] or at https://jamespogran.com[91].

[77] https://www.meetup.com/chgopsug/
[78] https://twitter.com/chgopsug
[79] https://www.techsnips.io/contributors/francisco-navarro/
[80] https://twitter.com/ctmcisco
[81] http://www.linkedin.com
[82] https://gnvpsug.github.io
[83] https://andrewpla.github.io
[84] https://twitter.com/AndrewPlaTech
[85] https://github.com/andrewpla
[86] https://puppet.com
[87] https://pup.pt/vscode
[88] https://www.packtpub.com/networking-and-servers/learning-powershell-dsc-second-edition
[89] https://twitter.com/ender2025
[90] https://github.com/jpogran
[91] https://jamespogran.com

Jess Pomfret

Jess is a SQL Server DBA in Northeast Ohio. One of the most important traits of a DBA, being lazy, means that automating common tasks with PowerShell is a passion of hers. She also contributes to open source projects including dbatools[92], dbachecks[93] and recently has contributed a couple of resources to the SqlServerDsc[94] module.

Thomas Rayner

Thomas Rayner is a Senior Security Service Engineer at Microsoft. His background is in Cloud and Datacenter Management, specializing in DevOps, systems and process automation, public, private and hybrid cloud, and .NET coding. Thomas is a prominent speaker, best-selling author, and instructor covering a vast array of IT topics. Thomas is very active within the technical community and a variety of Microsoft technical and strategic teams. You can get in contact with Thomas by email[95], on Twitter at @MrThomasRayner[96], or on LinkedIn[97].

Joel Sallow

Joel is a curious PowerShell enthusiast, fairly new to the arena. He has been working in PowerShell circles since late 2017, and loves clean & well-written code, frequently spending time helping others untangle the day's unruly code. You can follow him on Twitter[98] and Github[99], or check out his blog[100].

Glenn Sarti

With a penchant for DevOps, Puppet, Neo4j or anything Windows related, Glenn has been in IT Infrastructure and Development Engineering over 20 years. You can find him online through his blog[101] or on Twitter[102].

Mike Shepard

Mike has been actively programming since the early 80's with anything that he could get his hands on. He's the founder of the Southwest Missouri PowerShell User Group, the author of two PowerShell books, and a contributor to the first *PowerShell Conference Book*. Mike is currently a Solutions Architect at Jack Henry and Associates, specializing in build and deployment automation. Current interests include databases and DSLs.

[92] https://github.com/sqlcollaborative/dbatools
[93] https://github.com/sqlcollaborative/dbachecks
[94] https://github.com/powershell/sqlserverdsc
[95] mailto:thmsrynr@outlook.com
[96] https://twitter.com/MrThomasRayner
[97] https://linkedin.com/in/thomasrayner
[98] https://twitter.com/vexx32
[99] https://github.com/vexx32
[100] https://vexx32.github.io
[101] https://glennsarti.github.io/
[102] https://twitter.com/glennsarti

Justin Sider

Mr. Sider has 15+ years of experience as owner of his own tech business and in leadership roles for various tech companies. Currently the Chief Information Officer for Belay Technologies, he leads the development and implementation of a tool utilizing VMware for an automated provisioning & testing solution. Mr. Sider has 10+ years of experience working with VMware products and programming with PowerShell. You can follow Justin on twitter at @jpsider[103], at the PSGallery Gallery[104],on GitHub[105] or read an occasional blog at Invoke-Automation[106].

Daniel Silva

Daniel is a software developer with focus on .NET and SQL Server. He is also a Raspberry Pi and DIY enthusiast. He decided, after several projects (including making an arcade console), to run PowerShell Core on his Raspbery Pi and explore the world of IoT. He is always exploring new things and embracing new challenges. More content can be found on his blog[107].

Fernando Tomlinson

Fernando began developing in PowerShell years ago for defensive purposes and hasn't stopped since! He is the co-developer of Under the Wire[108] and the developer of PoSh Hunter[109]. Both sites help people learn PowerShell or hone the skills they already have. You can follow him on Twitter at @Wired_Pulse[110].

Stephen Valdinger

Stephen Valdinger is a 13 year IT veteran with a penchant for turning the repetitive into PowerShell. He can be found on Twitter[111] and on his Blog[112]. He's active in the PowerShell community Slack[113] and Discord[114] servers.

Stéphane Van Gulick

Stéphane is the co-founder of the French speaking PowerShell User Group (FRPSUG), the co-founder of the SPADUG (PowerShell user group in Strasbourg). He is also a speaker, author and

[103] https://twitter.com/jpsider
[104] https://www.powershellgallery.com/profiles/jpsider/
[105] https://github.com/jpsider
[106] https://invoke-automation.blog/
[107] https://danielssilva.dev/
[108] http://underthewire.tech
[109] https://posh-hunter.com
[110] https://twitter.com/Wired_Pulse
[111] https://twitter.com/steviecoaster
[112] https://steviecoaster.dev
[113] http://aka.ms/psslack
[114] https://aka.ms/psdiscord

blogger and works as Senior DevOps engineer in Switzerland. Stéphane created and maintains several open source frameworks, such as PSHTML[115] and PsClassutils[116] to name a few. You can find Stéphane blogging on PowershellDistrict[117] and on his personal blog Stephanevg.github.io[118]. You can also follow Stéphane on twitter at Stephanevg[119].

Mark Wragg

Mark is a DevOps Engineer who is currently contracting in the south east of England. He contributed a chapter on mastering Pester to the previous incarnation of the *PowerShell Conference Book*[120], which he was subsequently asked to deliver as a talk at the PSDay London conference in 2018[121]. Mark has been using PowerShell since 2013 and is a huge fan of delivering automation with PowerShell and other DevOps tools, particularly within the Microsoft technology space. Mark blogs at wragg.io[122] and can be found on twitter as @markwragg[123].

Michael Zanatta

Michael is a passionate PowerShell scripter, speaker, advocate & streamer working as a Senior Automation Consultant at Insync Technology[124]. You can follow him on Twitter[125] or LinkedIn[126]. Michael is a co-founder of the (Brisbane PowerShell User Group[127]) (YouTube[128]) and author of his livestream on Twitch[129].

[115] https://www.github.com/stephanevg/phstml
[116] https://www.github.com/stephanevg/psclassutils
[117] https://www.powershellDistrict.com
[118] https://www.stephanevg.github.io
[119] https://www.twitter.com/stephanevg
[120] https://leanpub.com/powershell-conference-book
[121] https://youtube.com/watch?v=BbOiQCgDDR8
[122] https://wragg.io
[123] https://twitter.com/markwragg
[124] https://insynctechnology.com.au
[125] https://twitter.com/PowerShellMich1
[126] https://www.linkedin.com/in/michael-zanatta-61670258/
[127] https://www.meetup.com/Brisbane-PowerShell-User-Group
[128] https://www.youtube.com/channel/UCQfLvFYohCCm_gTPEUfaAbw
[129] https://www.twitch.tv/videos/431222542

How to Use This Book

Imagine attending a PowerShell conference where over thirty speakers who are subject matter experts in the industry are each presenting one forty-five minute session. All the sessions are at different times so there's no need to worry about choosing between them. You might be wondering how much a conference like this will cost by the time you pay for the conference, hotel, airfare, and meals? Well, there's no need to worry because this conference doesn't cost a fortune because it's actually a book that's designed to be like a conference.

This book is designed to be like a conference in a book where each chapter is written by a different author who is a subject matter expert on the topic covered in their chapter. Each chapter is also independent of the others so you can read one chapter, ten chapters, or all chapters. You can start with the first chapter, the last one, or somewhere in-between and not miss out on anything related to that particular topic.

This is a Leanpub "Agile-published" book. We completed publishing all chapters for this book. However, as we discover issues or readers report them, we may release additional updates. If you allow Leanpub to do so, you'll get an e-mail each time the book is updated, and you can re-download the new content at no extra charge. To provide feedback, use the "Join the Forum" or the "Email the Authors" link on the book's Leanpub web page[130]. Whether it's a code error, a typo, or a request for clarification, our editors can review your feedback, make updates, and re-publish the book. Unlike the traditional paper publishing process, your feedback can have an immediate effect!

About OnRamp

OnRamp is an annual entry-level education program focused on PowerShell and DevOps. It's a conference within a conference at the PowerShell + DevOps Global Summit. The OnRamp track requires a distinct ticket type which is specifically 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, although some basic knowledge of server administration is useful. You'll network with other Summit attendees who are attending the expert level sessions during keynotes, meals, and evening events.

Through fundraising and corporate sponsorships, The DevOps Collective, Inc.[131] will be offering a number of 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.

[130] https://leanpub.com/psconfbook2
[131] https://devopscollective.org/

More information about the OnRamp track[132] at the PowerShell + DevOps Global Summit and their scholarship program[133] can be found on the PowerShell.org[134] website.

See the DevOps Collective Scholarships cause[135] on Leanpub.com[136] for more books that support the OnRamp scholarship program.

Prerequisites

Prior experience with PowerShell is highly recommended. This book is written for the intermediate to advanced audience and each chapter assumes that you've completed the OnRamp track at the PowerShell + DevOps Global Summit or have equivalent experience with PowerShell.

A Note on Code Listings

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 things 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 "dri\
vetype=3" -property DeviceID,Size,FreeSpace
```

Here, you can see the default action for a too-long line - it gets word-wrapped, and a backslash inserted at the wrap point to let you know. Attempts have been made to avoid those situations, but they may sometimes be unavoidable. When they *are* avoided in this book, it may be with awkward formatting, such as using backticks (‘)

```
Get-CimInstance -ComputerName $computer `
                -Classname Win32_logicalDisk `
                -Filter "drivetype=3" `
                -property 'DeviceID','Size','FreeSpace'
```

This code is formatted purely for reading purposes, you would never write code in this manner.

If you are reading this book on a Kindle, tablet or other e-reader, then all code formatting bets are off the table. There's no telling what the formatting will look like due to how each reader might format the page.

When *you* write PowerShell expressions, you shouldn't be limited by these constraints. And all downloaded code samples don't have these limitations.

[132] https://powershell.org/summit/summit-onramp/
[133] https://powershell.org/summit/summit-onramp/onramp-scholarship/
[134] https://powershell.org/
[135] https://leanpub.com/causes/devopscollective
[136] https://leanpub.com/

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, login to Leanpub, and click on the "Join the Forum" link in the Feedback section of this book's webpage[137].

[137] https://leanpub.com/psconfbook2

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, making opportunities for folks who otherwise might never see them. That's incredible and needs no qualifiers.

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

We also want to thank Puppet, Inc.[138], who provided support for the editing process.

[138] https://puppet.com

Disclaimer

All code examples shown in this book have been tested by each individual chapter author and every effort has been made to ensure that they're error free, but since every environment is different, they shouldn't be run in a production environment without thoroughly testing them first. 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 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!

Part I: Systems Management

This section highlights using PowerShell and adjacent tools to manage systems - conform their configuration, spin them up, and otherwise make them do the things we need them to.

Automate Patching With PoshWSUS and PowerShell Scheduled Jobs

by Bill Kindle

Introduction

Hello there! If you're a sysadmin, chances are you have experienced both the joy and pain (but mostly pain) of a wonderful Microsoft product known as *Windows Server Update Services* or **WSUS**. My current role involves heavy use of WSUS, patching fleets of servers and workstations. That requires short maintenance windows and a number of tasks that must be performed before reboots. When you have more than one customer environment, being able to perform those tasks automatically, uniformly, and reliably is important. I want to show you how to make working with WSUS a little less painful.

Let's get started!

PoshWSUS

PoshWSUS is a PowerShell module created by *Boe Prox*[139] that manages WSUS. This module contains a plethora of useful cmdlets that allow you to manage & maintain clients and Windows Updates. There's even a cmdlet to help you *clean up* WSUS.

PoshWSUS gives you a little more control than the built-in WSUS cmdlets[140] available in Windows Server 2016 and above do.

PowerShell Scheduled Jobs

A scheduled job is a background job. Starting with PowerShell 3.0, a new module called **PSScheduledJob**[141] was introduced with an additional 16 cmdlets for managing these jobs. You only need to know when, how often, and what command (a script) to run. When you have specific maintenance windows and a litany of maintenance tasks to complete, scheduled jobs can enhance your efficiency in performing those tasks.

[139] https://github.com/proxb/PoshWSUS
[140] https://docs.microsoft.com/en-us/powershell/module/updateservices/?view=win10-ps
[141] https://docs.microsoft.com/en-us/powershell/module/psscheduledjob/?view=powershell-5.1

Example Scenario

This part is important as it will make creating the script a lot easier.

You have a WSUS server and configured computer groups based on your three target groups: Production (**PROD**), User Acceptance Testing, (**UAT**), and Development (**DEV**). To add a little more complication, you also have Primary (**PRI**) and Secondary (**SEC**) servers as subgroups within each target group. On top of all this, each target group has a strict maintenance window and can't be done at the same time. Conservatively, you plan out the monthly patch cycle for each environment to occur on the weekends. Developers want patches to be deployed on Saturday between 8 AM and 12 PM. UAT testing can only be performed Monday through Tuesday during normal business hours of 9 AM to 5 PM. The customer production environment must be patched between 7 AM and 11 PM on the last Sunday of each month according to the Service Level Agreement (SLA).

Microsoft's '*patch Tuesday*' occurs the second Tuesday of each month.

Because you don't like working weekends and have some **PowerShell** knowledge, you decide to build some scripts that will do this task for you. The only manual thing you will do is schedule a maintenance window in your monitoring solution to prevent extraneous alert emails during these times.

Bill, why not use Group Policy?

Personally, it comes down to having specific maintenance windows, special pre-maintenance tasks, and controlled release requirements. When using a release technique called *Deadlining*, based on your client's WSUS check in interval, all local and group policies are overrode. The update installs immediately and will automatically reboot when complete. Group Policy enforces settings that prevent the systems from updating at the wrong time. Going into detail about the precise settings would be outside the scope of this chapter.

Getting Started

PowerShell 5.1 will be used. This has not been fully tested with PowerShell Core 6.2.1 at the time of this writing and there hasn't been a requirement to use it *yet*.

The following code example will download the latest version from the https://powershellgallery.com/[142] repository.

[142] PowerShellGallery.com

```
# It's good practice to put modules in your user context, not the system's.
Install-Module -Name PoshWSUS -Scope CurrentUser
```

Alternatively, you can also clone from the source using Git:

```
# Places the module directly in your user context
Set-Location -Path "C:\users\$env:USERNAME\Documents\WindowsPowerShell\Modules\"

git clone https://github.com/proxb/PoshWSUS/
```

Once installed, verify that the module installed correctly and can be imported.

```
<#
    If you cloned from the source directly, you may be able to skip this part
    as long as you restarted your console session.
#>
Import-Module -Name PoshWSUS

# Show the possibilities!
Get-Command -Module PoshWSUS -All
```

At the time this chapter was being written, there were 95 available cmdlets to choose from in this module.

Now you are ready to begin working with WSUS through the PoshWSUS module.

Connect And Poke Around

Now that you have the correct module loaded, you need to connect to your WSUS instance. To do so you'll need the `Connect-PSWSUSServer` cmdlet.

Take a moment here and explore the built-in help file.

```
Get-Help Connect-PSWSUSServer -ShowWindow
```

The two parameters you will need are `-WsusServer` and the `-Port` WSUS is operating on. By default, WSUS uses HTTP 8530 and for HTTPS/SSL 8531. If your WSUS instance is different, please consult docs.microsoft.com[143] for more information.

[143] https://docs.microsoft.com/en-us/windows-server/administration/windows-server-update-services/deploy/2-configure-wsus

You will need to make sure you are a member of the proper security groups required to manage WSUS or the connection may fail.

Now that you know how to connect to your WSUS instance using the `Connect-PSWSUSServer` cmdlet and a few parameters, consider the following example. Here the lab server's Fully Qualified Domain Name (FQDN) is used and the default port of **8530**. The `-Verbose` parameter issued only to show you the connection messages:

```
Connect-PSWSUSServer -WsusServer 'WSUS.kindlelab.int' -Port 8530 -Verbose
```

Output:

```
VERBOSE: Connecting to WSUS.kindlelab.int <8530>

Name                 Version              PortNumber      ServerProtocolVersion
----                 -------              ----------      ---------------------
WSUS.kindlelab.int   6.3.9600.18838       8530            1.8
```

As you can see, you now have an active connection and can begin issuing other cmdlets supplied by the module. Start by using another cmdlet, `Get-PSWSUSServer` with the parameter `-ShowConfiguration` to gather some documentation.

```
Get-PSWSUSConfig
```

Output:

```
UpdateServer                 : Microsoft.UpdateServices.Internal.BaseApi.Up...
LastConfigChange             : 5/26/2019 2:22:27 PM
ServerId                     : 00000000-0000-0000-0000-000000000000
SupportedUpdateLanguages     : {he, cs, fr, es...}
TargetingMode                : Server
SyncFromMicrosoftUpdate      : True
IsReplicaServer              : False
HostBinariesOnMicrosoftUpdate : False
UpstreamWsusServerName       :
UpstreamWsusServerPortNumber : 8530
UpstreamWsusServerUseSsl     : False
UseProxy                     : False
ProxyName                    :
ProxyServerPort              : 8080
UseSeparateProxyForSsl       : False
SslProxyName                 :
SslProxyServerPort           : 443
AnonymousProxyAccess         : True
```

That's just a small sampling of the available properties. There's no need to go into further details as that would be out of the scope of this chapter. It's just a little something to keep in mind while working with any WSUS server using this module.

> A similar command *`Get-PSWSUSServer -ShowConfiguration`* will also display the same information.

Building A Solution For Automated Patch Management

At this point, it is strongly suggested that you have a documented plan in place for your patching cycle. If you recall in our example scenario, you have known maintenance windows to work with. This makes building a script a lot easier in the long run.

Here's a little *pseudo code* to map out the initial process:

1. Connect to WSUS instance.
2. Set a few variables for groups, Knowledge Base (KB) list, and date/time.
3. Read in KB's that need deployed / installed.
4. Set approval flag and deadline to designated groups.
5. Disconnect from WSUS instance.

The first script will be called `Deploy-PriSecUpdates.ps1` and will be for the **PRI** and **SEC** groups. Now you can code!

The assumption is made that this script will be run from the WSUS instance, not another server. This is why the environment variable is used for the computer name.

```
# Import required tooling. Should already be there but make sure.
Import-Module -Name PoshWSUS

# Begin connection.
# Can use '[IP]' or '[FQDN]' here instead of environment variable.
Connect-PSWSUSServer -WsusServer $env:COMPUTERNAME -Port 8530 -Verbose

# Process
# Set the groups
$PRI = Get-PSWSUSGroup -Name 'PRI'
$SEC = Get-PSWSUSGroup -Name 'SEC'

# Set the deadlines. 2 hours is a good timeframe.
```

```
14 $PRIDeadline = (Get-Date).addHours(2)
15 $SECDeadline = (Get-Date)
16
17 # text file containing KB's
18 $Updates = Get-Content -Path 'C:\PScripts\Maintenance\updates.txt'
19 <#
20     The text file will look like this:
21
22     KB4509000
23     KB4509001
24     KB4509002
25     etc for each KB released that month.
26
27     There's a way to eliminate this step which will be discussed later.
28 #>
29
30 # Profit!
31 # Just as you would select and click on updates to install in the WSUS
32 # management console, this is simply doing the same thing with the
33 # below lines of code:
34
35 Get-PSWSUSUpdate -Update $Updates |
36     Approve-PSWSUSUpdate -Group $PRI -Action Install -Deadline $PRIDeadline
37
38 Get-PSWSUSUpdate -Update $Updates |
39     Approve-PSWSUSUpdate -Group $SEC -Action Install -Deadline $SECDeadline
40
41 # Cleanup. It's best practice to always close connections when they are
42 # no longer needed in a script.
43 Disconnect-PSWSUSServer
44
45 <#
46     Added bonus of having an email alert to let you know the script ran.
47     You'll need an SMTP server in your environment or access to one elsewhere.
48 #>
49
50 #Region Email Alert
51 # building a simple multiple line body here.
52 $Body       = @()
53 $Body       += 'The following updates were released:'
54 $Body       += "$Updates"
55
56 $Body       = $Body | out-string
57
58 $EmailSplat = @{
59     From        = "Automated Patching With WSUS<Strongbad@homestarrunner.com>"
```

```
    To          = "<homestar@homestarrunner.com>"
    Subject     = 'WSUS Release Status - Updates Deployed'
    Body        = "$Body"
    Priority    = 'High'
    SMTPServer  = '1.2.3.4'
    ErrorAction = 'SilentlyContinue'
}

# Now just take the splatted array values and pass them along to the cmdlet
Send-MailMessage @EmailSplat

#EndRegion
```

The email splat is completely optional, but it's nice to have.

If you open up your WSUS management console now, you'll notice that updates that were previously unapproved are now approved and deadlined.

The next script, `Deploy-DevUatUpdates.ps1`, will much of the same code. However, you're going to change couple of lines. To save time, make a copy of the first script and rename it.

```
# Import required tooling. Should already be there but make sure.
Import-Module -Name PoshWSUS

# Begin connection.
# Can use '[IP]' or '[FQDN]' here instead of environment variable.
Connect-PSWSUSServer -WsusServer $env:COMPUTERNAME -Port 8530 -Verbose

# Process
# Set the groups
$PRI = Get-PSWSUSGroup -Name 'DEV'
$SEC = Get-PSWSUSGroup -Name 'UAT'

# Set the deadlines. 2 hours is a good timeframe.
$DEVDeadline = (Get-Date)
$UATDeadline = (Get-Date).addHours(2)

# text file with the KB's
$Updates = Get-Content -Path 'C:\PScripts\Maintenance\updates.txt'
<#
    The text file will look like this:

    KB4509000
    KB4509001
    KB4509002
    etc for each KB released that month.

```

```
    There's a way to eliminate this step which will be discussed later.
#>

# Profit!
# Just as you would select and click on updates to install in
# the WSUS management console, this is simply
# doing the same thing with the below lines of code:

Get-PSWSUSUpdate -Update $Updates |
    Approve-PSWSUSUpdate -Group $DEV -Action Install -Deadline $DEVDeadline

Get-PSWSUSUpdate -Update $Updates |
    Approve-PSWSUSUpdate -Group $UAT -Action Install -Deadline $UATDeadline

# Cleanup. It's best practice to always close connections when
# they are no longer needed in a script.
Disconnect-PSWSUSServer

<#
    Added bonus of having an email alert to let you know the script ran.
    You'll need an SMTP server in your environment or access to one elsewhere.
#>

#Region Email Alert
# building a simple multiple line body here.
$Body       = @()
$Body       += 'The following updates were released:'
$Body       += "$Updates"

$Body       = $Body | out-string

$EmailSplat = @{
    From        = "Automated Patching With WSUS<Strongbad@homestarrunner.com>"
    To          = "<homestar@homestarrunner.com>"
    Subject     = 'WSUS Release Status - Updates Deployed'
    Body        = "$Body"
    Priority    = 'High'
    SMTPServer  = '1.2.3.4'
    ErrorAction = 'SilentlyContinue'
}

# Now just take the splatted array values and pass them along to the cmdlet
Send-MailMessage @EmailSplat

#EndRegion
```

Now all you have to do is sit back and wait for the reboots to occur! This is great and all, but you still have to manually run these scripts. To fully automate this task, you can use PSScheduledJobs.

Tying It All Together With PSScheduledJobs

Now that you've built two fully working scripts that perform all the point and click tasks you once had, it's time to set a schedule so that you can meet SLA requirements.

For this, you will create a script for a one time task.

```
# Since this is a throwaway script, didn't go crazy with getting credentials
$Options = New-ScheduledJobOption -RunElevated -RequireNetwork
$Cred = Get-Credential -UserName KindleLab\SB_WSUS

# Assign the trigger variables
$trigger1 = New-JobTrigger -Once -At "7/13/2019 8:00:00 AM"
$trigger2 = New-JobTrigger -Once -At "7/28/2019 7:00:00 AM"

# Build the jobs using splatting
$Job1 = @{
    Name                = 'Primary and Secondary Server Maintenance'
    Trigger             = $Trigger2
    Credential          = $Cred
    FilePath            = 'C:\PScripts\Deploy-PriSecUpdates.ps1'
    ScheduledJobOption  = $Options
}

$Job2 = @{
    Name                = 'Development and UAT Server Maintenance'
    Trigger             = $Trigger1
    Credential          = $Cred
    FilePath            = 'C:\PScripts\Deploy-DevUatUpdates.ps1'
    ScheduledJobOption  = $Options
}

# Registering the job finishes the script.
Register-ScheduledJob @Job1
Register-ScheduledJob @Job2
```

You only need to run this script once. After the jobs have been successfully registered, you can make alterations to them from within the Task Scheduler management interface just as you would any other scheduled task. The only major difference is the location in which these two jobs are stored. You can find the scheduled jobs under **Task Scheduler Library > Microsoft > Windows > PowerShell > ScheduledJobs**. Now if all goes according to plan, the scheduled jobs will execute the respective script for each of your environments with no interaction required.

Congratulations! You just automated a WSUS update deployment!

How This Script Fitted Into My Process

The driving force behind the creation of this script stemmed from the fact that I have to wake early and sometimes work late to complete a patching schedule on time. The process involves connecting to a VPN, going through two jump servers, and then working with multiple tools and applications.

There is quite a bit of room for error, and a few have been made.

Luckily, the documented procedures to follow which made the creation of scripts for the applications and servers maintenance plan easy. These scripts run as scheduled jobs before and after the `Deploy-Updates.ps1` script runs. All I needed to do was to time myself performing the tasks manually. This gave me a rough estimate of how to set the cadence for execution of the scripts. Over time, these schedules were fine tuned to happen in a fraction of the time it used to take to manually perform the task. One environment that used to take 3 hours, now completes a full maintenance cycle in about an hour.

You may find yourself asking, *"Why all the separate scripts?"*. My reasoning for doing it this way is about separation. Separation allows me to better troubleshoot a failing part of my process. For instance, a shutdown script may run correctly but the deployment script fails. I don't have to troubleshoot one massive script and can also check the Task Scheduler for timestamps and error codes. You may find yourself not having or wanting that much control.

Next Steps

Now that you can automate update deployments on your own schedule, try something else. Here are some suggestions based on common tasks when working with WSUS:

- Try setting up an automatic cleanup schedule using `Start-PSWSUSCleanup`.
- Try adding and removing groups in WSUS using `New-PSWSUSGroup` and `Remove-PSWSUSGroup`.
- Use `Add-PSWSUSClientToGroup` and `Remove-PSWSUSClientFromGroup` to try adding and removing clients in WSUS.

Summary

The PoshWSUS and PSScheduledJobs modules allow you to take control of WSUS instances using PowerShell. Using this module nearly eliminates the need to open another WSUS console again. This will save you time and further automate what can be an annoying, repetitive yet necessary systems administration task. It's with great hope that this chapter has inspired you to do more with PowerShell.

Building SQL Servers with Desired State Configuration

by Jess Pomfret

Desired State Configuration (DSC) is a powerful tool that enables you to build and configure infrastructure with ease. Although this was first available with Windows Management Framework (WMF) 4.0, the release of WMF 5.1 added many useful enhancements. If you are working with an older server, before Windows Server 2016 (or Windows 10), it's recommended that you upgrade WMF manually.

DSC has many use cases, and there are varying degrees of complexity and integration that you can strive towards. This chapter demonstrates creating a reasonably simple DSC setup that will install and configure a SQL Server instance on a target node.

Lab setup

The examples in this chapter are executed against a small lab setup on a laptop. Using Hyper-V to host two Windows Server 2019 virtual machines, one will be used as an authoring station (`DscSvr1`) while the configurations will be enacted against the second server (`DscSvr2`).

Infrastructure as Code

To lay some groundwork on why you might want to use DSC, it's important to talk about *Infrastructure as Code* (IaC). IaC is the idea that our infrastructure can be defined using code. This is hard to wrap your head around if you still picture servers as physical machines living in your data center. Today the majority of our servers are virtual machines (VMs), so it's easier to imagine these when talking about IaC.

A simple example of IaC would be a script that defines your VM. The script defines the number of virtual CPUs, the amount of memory allocated, and the disk layout. This document becomes the artifact in this process, and the first step of embracing IaC is to check that into source control. With the description of how to build your server in source control you now have an auditable log of any changes that are implemented with a clear outline on who made what change when.

The source control repository is also the first step in a continuous integration\continuous delivery (CICD) pipeline. Benefits of using a CICD pipeline to build servers include:

- Security - the pipeline service account needs the high level privileges to build servers rather than people.
- Repeatability - when moving through environments an important checkbox won't be forgotten by a human.
- Built-in documentation - as previously mentioned, any changes are documented which creates an audit log.

With these benefits comes the major risk that the IaC pipeline contains the blueprints of your entire infrastructure. The pipeline should be highly secured to ensure that these blueprints don't fall into the wrong hands.

Desired State Configuration

DSC is a tool that enables us to implement IaC. It provides the framework needed to define the desired state of your infrastructure. DSC is written using a Domain Specific Language (DSL). It's PowerShell but with its own domain of terminology and patterns.

DSC is also based on open standards. DSC uses Windows Management Instrumentation (WMI) and Common Information Model (CIM). WMI is an industry standard for managing and configuring your enterprise environment. CIM is used to represent the objects you are configuring. The standard for configuring CIM classes is the Managed Object Format (MOF). These three standards means that DSC can integrate with other configuration management implementations.

Stages of Desired State Configuration

There are four main stages to consider when talking about DSC. These stages are used as a guide to walk through the entire process of building a SQL Server.

Author

The first stage in this process is to write our configuration. This will define the desired state of our target node or nodes. This configuration document should contain the complete definition of your target node, and as mentioned earlier with IaC, should be source controlled.

Declarative Syntax

PowerShell is usually written using imperative language. This code steps through the actions that should be taken to make the desired changes. For example, the snippet below will create a folder:

```
1 New-Item -Path 'C:\temp' -ItemType Directory
```

Unlike the above code to create a folder, DSC Configurations are written using a declarative syntax. This describes the desired state of our target node. The following describes that a directory should be present at the location `C:\temp`, but it doesn't explain how to complete the task.

```
File CreateDataDir {
    DestinationPath = 'C:\temp'
    Ensure = 'Present'
    Type = 'Directory'
}
```

Using the above declarative syntax removes the need for including any error handling. With the imperative code after the first execution any subsequent runs will be met with a sea of red since the folder already exists. On the other hand, when using declarative language if the folder exists, the desired state is met and no further action is required.

Idempotent

Another concept to understand when working with DSC is idempotency. Being idempotent means that the same configuration can be applied multiple times with the same end result. If the node is already in the desired state, the configuration will note that and carry on. If it's not in the desired state, code will be executed to "make it so." This means that incremental changes can be made to our configuration. When the configuration is reapplied, only the changes will be executed.

Resources

The main building block used to write a configuration document are resources. These come packaged as PowerShell modules and contain the code needed to get your target node into the defined desired state. If your machine has WMF installed, you will also have the `PSDesiredStateConfiguration` module which contains 22 resources already built-in.

PowerShell likes to be helpful. Armed with `Get-Command` and `Get-Help` you can find functions and cmdlets for whatever you need. DSC is no different. It comes with `Get-DscResource`, which can be used to find resources on your local machine, and `Find-DscResource` to search the PowerShell Gallery. First, use `Get-DscResource` to investigate the built-in module. Running the following will list out all the resources included:

```
Get-DscResource -Module PSDesiredStateConfiguration
```

ImplementedAs	Name	ModuleName	Version	Properties
Binary	File			{DestinationPath, Attributes, Ch...
PowerShell	Archive	PSDesiredStateConfiguration	1.1	{Destination, Path, Checksum, Cr...
PowerShell	Environment	PSDesiredStateConfiguration	1.1	{Name, DependsOn, Ensure, Path...}

> Not full output

When using the `-Name` parameter to get a single resource, the `-Syntax` parameter can also be used as shown below. This gets the syntax needed to use the resource within your configuration. You can copy and paste the output returned and begin filling in the properties you need to define.

```
Get-DscResource -Name File -Syntax
```

```
    File [String] #ResourceName
    {
        DestinationPath = [string]
        [Attributes = [string[]]{ Archive | Hidden | ReadOnly | System }]
        [Checksum = [string]{ CreatedDate | ModifiedDate | ...}]
        [Contents = [string]]
        [Credential = [PSCredential]]
        [DependsOn = [string[]]]
        [Ensure = [string]{ Absent | Present }]
        [Force = [bool]]
        [MatchSource = [bool]]
        [PsDscRunAsCredential = [PSCredential]]
        [Recurse = [bool]]
        [SourcePath = [string]]
        [Type = [string]{ Directory | File }]
    }
```

The built-in resources aren't the only ones available. There are many more modules full of resources on the PowerShell Gallery. To search for those, use `Find-DscResource`. In the example below a specific resource name is passed in, but wildcards can also be used.

```
Find-DscResource -Name SqlSetup
```

Name	Version	ModuleName	Repository
SqlSetup	13.0.0.0	SqlServerDsc	PSGallery

On June 20th 2019, there were 1,506 resources available in the PowerShell Gallery. They can be counted using `(Find-DscResource).Count`.

Writing the Configuration

Once the required resources have been identified, it's time to create a configuration document. The first line is similar to creating a function in PowerShell. Since DSC uses a DSL, the rest of the code looks different. The configuration has been named *CreateSqlFolder*. Within the configuration, the `Import-DscResource` keyword will be used to pull in the modules that contain the resources needed to configure the target node.

It's important to note that the modules used within the configuration must be available on both the authoring station and the target node with matching versions.

Then the *Node* block is defined. There can be one or more node blocks per configuration. Each node block can contain more than one target node. Below, two target node names have been passed in using array notation. Nested within the node blocks are where the resources are defined. Two *File* resources are added which create the directories needed for the data and log files once SQL Server is installed. Each resource is followed by a name. This friendly name must be unique within the configuration. It's a good idea to use meaningful names for these resources as they will be shown in the output. Having names that describe what they do will help when troubleshooting.

```
Configuration CreateSqlFolder {

    Import-DscResource -ModuleName PSDesiredStateConfiguration

    Node @('dscsvr1', 'dscsvr2') {
        File CreateDataDir {
            DestinationPath = 'C:\SQL2017\SQLData\'
            Ensure          = 'Present'
            Type            = 'Directory'
        }
        File CreateLogDir {
            DestinationPath = 'C:\SQL2017\SQLLogs\'
            Ensure          = 'Present'
            Type            = 'Directory'
        }
    }
}
```

Once the above code is executed, you can see a command has been created with the special type of configuration.

```
Get-Command -CommandType Configuration
```

CommandType	Name
Configuration	CreateSqlFolder

Managed Object Format Files

Once our configuration has been written, it must be compiled into a MOF file. This occurs by executing the configuration:

```
CreateSqlFolder -Output .\Output\
```

When you execute this, the defined output folder will now contain two MOF files, one per node.

There is an option to have multiple MOF files per node when using a technique called *Partial Configurations*. At the time of writing this chapter, these cause more trouble than they're worth. For now, stick with the notion of one MOF per node.

Configuration Data

The configuration written in the last example was a simple example and not realistic for a real world situation. The next step to enhance the configuration is using *Configuration Data* to separate the "data" from the "code." The *Configuration Data* is written as a hash table, either within the same file as the configuration or as an external psd1 file. If the following is saved as a psd1 file. The filename can then be passed into the configuration.

```
@{
    AllNodes = @(
        @{
            NodeName = "DSCSVR1"
            Environment = "Test"
        },
        @{
            NodeName = "DSCSVR2"
            Environment = "Production"
        }
    )
    NonNodeData = @{
        DataDir = "C:\SQL2017\SQLData\"
        LogDir = "C:\SQL2017\SQLLogs\"
        TestDir = "C:\TestForJess"
    }
}
```

The hash table must contain an 'AllNodes' key. It can also include other 'NonNodeData' key if desired. The above hash table defines two target nodes: one test and one production. It then defines three directories within the 'NonNodeData' that can also be accessed from within the configuration.

To use the *Configuration Data* within the *Configuration*, it will need to be passed in using the *Common Parameter* of `-ConfigurationData`. The `$AllNodes` special variable is used to access the defined nodes. The *NonNodeData* is accessed using the `$ConfigurationData` variable.

```
Configuration CreateSqlFolder {

    Import-DscResource -ModuleName PSDesiredStateConfiguration

    Node $AllNodes.NodeName {
        File CreateDataDir {
            DestinationPath = $ConfigurationData.NonNodeData.DataDir
            Ensure          = 'Present'
            Type            = 'Directory'
        }
        File CreateLogDir {
            DestinationPath = $ConfigurationData.NonNodeData.LogDir
```

```
            Ensure          = 'Present'
            Type            = 'Directory'
        }
    }

    Node $AllNodes.Where{$_.Environment -eq "Test"}.NodeName {
        File CreateTestDir {
            DestinationPath = $ConfigurationData.NonNodeData.TestDir
            Ensure          = 'Present'
            Type            = 'Directory'
        }
    }
}

CreateSqlFolder -Output .\Output\  -ConfigurationData .\03a_ConfigurationData.psd1
```

There is a mix of DSC and regular PowerShell in the second node block. The `Where` method is used to apply this block to only certain nodes. An extra test directory will be created on servers that have the environment defined as test in the configuration data. This provides flexibility. The same configuration can be used for all environments while still allowing some differences.

Publish

In the publish phase of DSC, the MOF files that were just created are shipped out to the target nodes. There are two modes that DSC can be used in: push and pull Push mode is simpler to setup and therefore will be used in these examples. Push mode involves actively pushing configurations to the target node. In pull mode, the nodes are registered to a pull server that contains the modules and configurations needed.

There are three options for how to setup a pull server: using the pull service on a Windows Server, setting up an SMB share, or using the Azure Automation platform. Once a pull server is set up you then register your target nodes with it. The nodes will check in to determine if there's a configuration that should be applied. The frequency of checking in is based on a configurable setting.

There are two commands that can be used to push out MOF files to the target node. First, `Publish-DscConfiguration` can be used to deliver the MOF to the node but not immediately enact it. Instead after a certain amount of time (configurable setting) has passed the configuration will be applied.

```
Publish-DscConfiguration -Path .\output\ -ComputerName dscsvr2 -Verbose
```

The other option is to use `Start-DscConfiguration`. This will push the MOF file and immediately enact to reach the desired state.

```
Start-DscConfiguration -Path .\output\ -ComputerName dscsvr2 -Wait -Verbose
```

In this example, use the `-Wait` and `-Verbose` parameters to be able to see the output returned to your console. If you don't specify these parameters, the console will return immediately and the execution will take place in the background within a PowerShell job.

Enact

Once the MOF file gets to the target node, the Local Configuration Manager (LCM) takes over. This is the engine of DSC. It's job is parsing and enacting the MOF. There are many settings that can be used to configure the LCM. Several of the settings available are described below. For a full list of settings, reference the Microsoft Docs[144].

Setting	Description
ActionAfterReboot	What should happen after a reboot. ContinueConfiguration or StopConfiguration.
CertificateID	Thumbprint of the certificate used to encrypt the MOF file.
ConfigurationMode	What the LCM does with the configuration document. This setting can be used to automatically keep your node in the desired state. ApplyOnly, ApplyAndMonitor or ApplyAndAutoCorrect.
ConfigurationModeFrequencyMins	How often should the LCM check configurations and apply them. If the ConfigurationMode is ApplyOnly this is ignored.
RebootNodeIfNeeded	If during the configuration a reboot is required should the node automatically reboot.
RefreshMode	Does the LCM passively wait for configurations to be pushed to it (push), or actively check in with the pull server for new configurations (pull).

If you don't use a certificate to encrypt the MOF file, passwords will be stored in plain text.

These settings can be changed by enacting a meta configuration. First, check the current settings using `Get-DscLocalConfigurationManager`:

```
Get-DscLocalConfigurationManager -CimSession dscsvr2  |
Select-Object ActionAfterReboot, RefreshMode, ConfigurationModeFrequencyMins
```

ActionAfterReboot	RefreshMode	ConfigurationModeFrequencyMins
ContinueConfiguration	Push	15

The *ConfigurationModeFrequencyMins* is how often the LCM will check for configurations to apply you can use the following configuration to define the desired settings. To change it:

[144] https://docs.microsoft.com/en-us/powershell/dsc/managing-nodes/metaconfig

```
[DSCLocalConfigurationManager()]
configuration LCMConfig
{
    Node dscsvr2
    {
        Settings
        {
            ActionAfterReboot = 'ContinueConfiguration'
            RefreshMode = 'Push'
            ConfigurationModeFrequencyMins = 20
        }
    }
}

LCMConfig -Output .\output\
```

The final line of the above snippet will create a meta MOF file for the target node. Once that has been generated, it can be applied by using `Set-DscLocalConfigurationManager`:

```
Set-DscLocalConfigurationManager -Path .\output\ -ComputerName dscsvr2 -Verbose
```

Monitor

The final step in the process is to review the configuration and report on any configuration drift. This is when nodes are no longer in the desired state that was defined.

This is the part of DSC that's lacking slightly. Hopefully, as DSC continues to be developed, there will be more work around the reporting aspect. Currently, the options for more complete reporting are to pair DSC up with a 3rd party tool, or to write your own tooling around your DSC implementation.

There is some information available though. It makes sense if you are following along to come back to this section after the last section of the chapter where you will install a SQL Server. The output below is from post SQL Server install using DSC.

First, check the current configuration of the node using `Get-DscConfiguration`:

```
Get-DscConfiguration -CimSession dscsvr2
```

```
...
ConfigurationName    : InstallSqlServer
DependsOn            : {[SqlSetup]InstallSql}
ModuleName           : SqlServerDsc
ModuleVersion        : 12.3.0.0
PsDscRunAsCredential :
ResourceId           : [SqlDatabase]CreateDbaDatabase
SourceInfo           :
Collation            : SQL_Latin1_General_CP1_CI_AS
Ensure               : Present
InstanceName         : MSSQLSERVER
Name                 : DBA
ServerName           : DSCSVR2
PSComputerName       : dscsvr2
CimClassName         : MSFT_SqlDatabase
...
```

> Not full output

It's important to note that this isn't stating the desired state, just the current configuration. For example, if the *DBA* database is dropped and then the above code rerun, you will get the following:

```
...
ConfigurationName    : InstallSqlServer
DependsOn            : {[SqlSetup]InstallSql}
ModuleName           : SqlServerDsc
ModuleVersion        : 12.3.0.0
PsDscRunAsCredential :
ResourceId           : [SqlDatabase]CreateDbaDatabase
SourceInfo           :
Collation            : SQL_Latin1_General_CP1_CI_AS
Ensure               : Absent
InstanceName         : MSSQLSERVER
Name                 : DBA
ServerName           : DSCSVR2
PSComputerName       : dscsvr2
CimClassName         : MSFT_SqlDatabase
...
```

> Not full output

It now shows that the the `Ensure` property is `Absent`, but it doesn't note that this isn't the desired state.

The next command available is `Get-DscConfigurationStatus`. This will return detailed information on completed configuration runs.

```
Get-DscConfigurationStatus -CimSession DscSvr2
```

Status	StartDate	Type	Mode	RebootRequested	NumberOf Resources	PSComputerName
Success	6/23/2019 7:45:40 AM	Initial	Push	False	11	DscSvr2

To determine if our node is still in the desired state or not, you can use `Test-DscConfiguration`. If you run this with just the `-CimSession` parameter it will just return true or false.

To get more information you can use the `-Verbose` switch, which outputs all the DSC verbose output. This is more useful than the first option, but means you will have to read through all the output to find which resources aren't in the desired state.

Our final option is to use the `-Detailed` parameter. This returns a PowerShell object that can be manipulated to find the information needed. For example, the code below just shows resources that aren't in the desired state are selected:

```
Test-DscConfiguration -CimSession DscSvr2 -Detailed | Select-Object ResourcesNotI\
nDesiredState
```

ResourcesNotInDesiredState
{[SqlDatabase]CreateDbaDatabase}

Troubleshooting

Although separate from monitoring, another important concept is being able to troubleshoot your configuration if something fails or the expected results don't occur. Along with the commands highlighted above in the monitoring section the Windows event logs can provide a lot of details on what might have gone wrong. These logs can be found on the target node within the event viewer at the following path `Application and Services Logs > Microsoft > Windows > Desired State Configuration`. There are two logs, "Operational" and "Admin," that are turned on by default. If these don't provide enough detail you can enable two additional logs, "Analytic" and "Debug."

Desired State Configuration and SQL Server

With a whistle-stop tour of the DSC architecture under your belt, it can now be applied to a real world problem. If you install SQL Server often, you have probably developed a checklist similar to the one below to make sure each server is built to the same standard.

1. Install Windows Features - .NET Framework
2. Create directories for Install/Data/Logs/Tempdb
3. Install SQL Server
4. Enable TCP/IP
5. Set Windows Firewall
6. Server Configuration Options (sp_configure)
    1. Backup compression
    2. CTOP
    3. MAXDOP
7. Create DBA Database

Each time a request comes in to build a server, this list can be followed and you'll end up with a server that meets your needs. However, it can be a tedious process to manually follow this list. Instead, let's translate this to use DSC.

For each step on my list there is a DSC resource available that will be used instead.

Step	Module	DSC Resource
Install Windows Features	PSDesiredStateConfiguration	WindowsFeature
Create directories	PSDesiredStateConfiguration	File
Install SQL Server	SqlServerDsc	SqlSetup
Enable TCP/IP	SqlServerDsc	SqlServerNetwork
Set Windows Firewall	SqlServerDscNetworkingDsc	SqlWindowsFirewall
	NetworkingDsc	Firewall
Server Configuration	SqlServerDsc	SqlServerConfiguration
Create DBA Database	SqlServerDsc	SqlDatabase

You can see that there are some options available. For example, the task of setting up the windows firewall can be completed one of two ways. Either the `SqlWindowsFirewall` resource from within the `SqlServerDsc` module can be used to open up the firewall for the features installed, or, if more flexibility is needed, the `Firewall` resource from the `NetworkingDsc` module can be used. This resource has more properties that can be set than those that are made available with the `SqlWindowsFirewall` resource.

Once the resources have been chosen, they will form the basis of the configuration. A full version of both the configuration document and some sample configuration data are available in the downloads for this book.

Using the lab setup described earlier, SQL Server was installed and configured as defined in the above checklist in less than 5 minutes. That's pretty impressive!

There are a few things to note within the sample scripts. First, note the `NonNodeData` within the configuration data. You can structure your configuration data in more than one way to define the properties that will be used in the configuration. For the directory structures, each folder is listed directly under the `NonNodeData` key. However, the SQL Server configuration settings that will be used by the `SqlServerConfiguration` resource are provided in a nested array called `ConfigOptions`.

```
NonNodeData = @{
        DataDir = "C:\SQL2017\SQLData\"
        LogDir = "C:\SQL2017\SQLLogs\"
        InstallDir = "C:\SQL2017\Install\"
        InstanceDir =  "C:\SQL2017\Instance\"
        ConfigOptions = @(
            @{
                Name    = "backup compression default"
                Setting = 1
            },
            @{
                Name    = "cost threshold for parallelism"
                Setting = 25
            },
            @{
                Name    = "max degree of parallelism"
                Setting = 4
            }
        )
    }
```

The difference in how the configuration data is structured means that it will be used differently when these properties are accessed within the configuration. This is a design point that you will want to think about as you build out your own solutions. For the folder structures, the resources will be built out as already highlighted, accessing the folder paths as so:

```
File CreateInstallDir {
    DestinationPath     = $ConfigurationData.NonNodeData.InstallDir
    Ensure              = 'Present'
    Type                = 'Directory'
}
File CreateInstanceDir {
    DestinationPath     = $ConfigurationData.NonNodeData.InstanceDir
    Ensure              = 'Present'
    Type                = 'Directory'
}
```

Since our configuration options are structured as an array, PowerShell can be used to generate the individual resources by looping through each item. This will create a resource for each setting in

the `ConfigOptions` array. As an example, the resource to set backup compression will be named "`SetConfigOption_backup compression default`".

```
$ConfigurationData.NonNodeData.ConfigOptions.foreach{
    SqlServerConfiguration ("SetConfigOption_{0}" -f $_.name) {
        DependsOn       = '[SqlSetup]InstallSql'
        ServerName      = $Node.NodeName
        InstanceName    = 'MSSQLSERVER'
        OptionName      = $_.Name
        OptionValue     = $_.Setting
    }
}
```

The `Get-Credential` function is used to set the "sa" user password. When this configuration is executed, a popup will appear to prompt for the password to be entered.

```
$saCred = (Get-Credential -Credential sa)
```

For this example, the following section has been added to the `AllNodes` key of the configuration data. Specifying the `NodeName` as `*` means that these settings will be applied to all nodes. In this case `PSDscAllowPlainTextPassword` has been set to allow plain-text passwords to be stored in the MOF file. **This isn't recommended for production use!**

```
@{
    NodeName = '*'
    PSDscAllowPlainTextPassword = $true
}
```

After you run this, if you inspect the MOF files that were generated, you can see the super secure sa password. For production, you should be encrypting your MOF file with a certificate. This is outside the scope of this chapter. More information on how to set this up can be found in the Microsoft Docs[145].

```
instance of MSFT_Credential as $MSFT_Credential1ref
{
Password = "Password1234";
 UserName = "sa";

};
```

Another useful property is `DependsOn`. This is a common property that's available on all resources. In the resource definition below, this property ensures that the configuration won't attempt to create the DBA database if the `InstallSql` resource isn't in the desired state. This is vital if you're using WMF 4.0, as DSC didn't follow the order laid out in the configuration. It, instead, enacted resources in any old order. It's still important with WMF 5.1. If the `InstallSql` resource fails for any reason, this will stop the resource to create a database from being enacted.

[145] https://docs.microsoft.com/en-us/powershell/dsc/pull-server/securemof

```
SqlDatabase CreateDbaDatabase {
    DependsOn           = '[SqlSetup]InstallSql'
    ServerName          = $Node.NodeName
    InstanceName        = 'MSSQLSERVER'
    Name                = 'DBA'
}
```

Running the configuration, the `-wait` and `-verbose` switches are set for `Start-DscConfiguration` so that as the configuration runs the output is returned to the console. Below is a snippet of this output showing the LCM enacting the `CreateDbaDatabase` resource. You can see it starts by running a test to see if the node is in the desired state. In this case it returns false so the set is called to 'make it so'.

```
VERBOSE: [DSCSVR2]: LCM:  [ Start  Resource ]  [[SqlDatabase]CreateDbaDatabase]
VERBOSE: [DSCSVR2]: LCM:  [ Start  Test     ]  [[SqlDatabase]CreateDbaDatabase]
VERBOSE: [DSCSVR2]:                            [[SqlDatabase]CreateDbaDatabase]
  Checking if database named DBA is present or absent
VERBOSE: [DSCSVR2]:                            [[SqlDatabase]CreateDbaDatabase]
  Information: PowerShell module SqlServer not found, trying to use older SQLPS m\
odule.
VERBOSE: [DSCSVR2]:                            [[SqlDatabase]CreateDbaDatabase]
  Importing PowerShell module 'SQLPS' with version '14.0' from path 'C:\Program F\
iles
  (x86)\Microsoft SQL Server\140\Tools\PowerShell\Modules\SQLPS\SQLPS.psd1'.
VERBOSE: [DSCSVR2]:                            [[SqlDatabase]CreateDbaDatabase]
  Connected to SQL instance 'DSCSVR2'.
VERBOSE: [DSCSVR2]:                            [[SqlDatabase]CreateDbaDatabase]
  Getting SQL Databases
VERBOSE: [DSCSVR2]:                            [[SqlDatabase]CreateDbaDatabase]
  SQL Database name DBA is absent
VERBOSE: [DSCSVR2]:                            [[SqlDatabase]CreateDbaDatabase]
  2019-06-26_03-33-51: Ensure is set to Present. The database DBA should be creat\
ed
VERBOSE: [DSCSVR2]: LCM:  [ End    Test     ]  [[SqlDatabase]CreateDbaDatabase]
  in 0.1870 seconds.
VERBOSE: [DSCSVR2]: LCM:  [ Start  Set      ]  [[SqlDatabase]CreateDbaDatabase]
VERBOSE: [DSCSVR2]:                            [[SqlDatabase]CreateDbaDatabase]
  Found PowerShell module SQLPS already imported in the session.
VERBOSE: [DSCSVR2]:                            [[SqlDatabase]CreateDbaDatabase]
  Connected to SQL instance 'DSCSVR2'.
VERBOSE: [DSCSVR2]:                            [[SqlDatabase]CreateDbaDatabase]
  Adding to SQL the database DBA.
VERBOSE: [DSCSVR2]:                            [[SqlDatabase]CreateDbaDatabase]
  2019-06-26_03-33-51: Created Database DBA.
VERBOSE: [DSCSVR2]: LCM:  [ End    Set      ]  [[SqlDatabase]CreateDbaDatabase]
  in 0.7350 seconds.
```

```
VERBOSE: [DSCSVR2]: LCM:  [ End    Resource ]  [[SqlDatabase]CreateDbaDatabase]
VERBOSE: [DSCSVR2]: LCM:  [ End    Set      ]
VERBOSE: [DSCSVR2]: LCM:  [ End    Set      ]    in  227.8890 seconds.
VERBOSE: Operation 'Invoke CimMethod' complete.
VERBOSE: Time taken for configuration job to complete is 228.496 seconds
```

The last line shows it took 228 seconds, or just under 4 minutes to configure the target node.

As mentioned previously, since DSC is idempotent you can make incremental changes to this configuration and reapply it. If you needed to create another database, you could add another resource to your configuration and rerun the script. The LCM will go through the MOF file testing each resource to determine if it's already in the desired state. If it's not, it will call the set function for that specific resource. The same goes for if you decide to change one of the configuration options. If, for example, you wanted to change the `cost threshold for parallelism` to 50, you would update the configuration data and rerun the configuration.

There are many ways to expand on this example, but this should give you an idea on how you can build a SQL Server using PowerShell DSC with ease.

Next Steps

If this chapter has interested you and you'd like to know how to expand on this idea, there are a few recommended topics. Azure Automation has capabilities to work as a pull server for your cloud or on-prem machines. This can help with the management of configurations and modules.

The use of configuration data is also an area that can be expanded on. As you build out a full list of settings and options for different environments, or even types of servers, you will find that this document can get large and overwhelming. Gael Colas has written a PowerShell module called Datum[146] that enables you to build out a hierarchy to manage this problem.

Finally, although not fully built out, there is the concept of ReverseDsc[147], This is a collection of PowerShell modules that you would point at an already configured node. The resulting output is the configuration that would have been needed to put that node in its current state. This could be a good method of creating the artifacts for IaC for existing servers.

[146] https://github.com/gaelcolas/datum
[147] https://github.com/Microsoft/ReverseDSC

Part II: PowerShell Tips & Tricks

This section highlights topics that make your use of PowerShell a little easier, a little friendlier; topics to improve your quality of life as a PowerShell user and developer.

Cross Platform PowerShell: Notes from the Field

by James Pogran

For years, PowerShell has been referred to as a thing that can only be used to manage Windows hosts and nothing more. It's always `if linux:bash` or `if windows:powershell`. But with PowerShell Core, you can do more! Wait, did I just rhyme? And in the introduction too. Oh the shame.

Now that it installs on most platforms, that means it all just works like magic and you can just sit back and relax! Right? Well, for the most part, PowerShell Core works everywhere - but like most things, the devil is in the details.

This chapter provides an overview of daily use operations like profiles, prompts, and modules. We then dive into hard-won best practices from the field: covering platform detection, aliases, environment variables, file access and encoding, newlines and case sensitivity, and path handling. It will end with recommendations about using the PowerShell Visual Studio Code extension to ensure your scripts work no matter the platform.

Daily Use

The goal of PowerShell Core is to be as backwards compatible as possible with Windows PowerShell, while also expanding the possible install base to Linux and MacOS platforms. This is a huge effort and there are some spots where PowerShell Core doesn't hold up to those goals, but for the most part PowerShell Core can be a daily driver.

You will find that the 'built-in' PowerShell modules and cmdlets work the same as they did in Windows PowerShell, just improved to handle other platform concerns. In many cases, these cmdlets and modules have benefited from community fixes and contributions that make them faster or easier to use in PowerShell Core than they were in Windows PowerShell.

In other cases, you will find gaps with modules or cmdlets that relied on Windows specific APIs, where either some functionality was removed or entire features absent. This is due to the goal of being cross-platform, things like Windows Management Instrumentation (WMI) or Component Object Model (COM) just couldn't be ported to other platforms.

Profile Differences

If you are familiar with Windows PowerShell, you know all the different profiles you can have:

```
$profile | Format-List * -force
```

Output:

```
AllUsersAllHosts       : C:\Program Files\PowerShell\6\profile.ps1
AllUsersCurrentHost    : C:\Program Files\PowerShell\6\Microsoft.
  PowerShell_profile.ps1
CurrentUserAllHosts    : C:\Users\james\Documents\PowerShell\profile.ps1
CurrentUserCurrentHost : C:\Users\james\Documents\PowerShell\Microsoft.
  PowerShell_profile.ps1
```

On Mac and Linux, they're in slightly different places:

```
$profile | Format-List * -force
```

Output:

```
AllUsersAllHosts       : /opt/microsoft/powershell/6/profile.ps1
AllUsersCurrentHost    : /opt/microsoft/powershell/6/Microsoft.
  PowerShell_profile.ps1
CurrentUserAllHosts    : /home/james/.config/powershell/profile.ps1
CurrentUserCurrentHost : /home/james/.config/powershell/Microsoft.
  PowerShell_profile.ps1
```

Notice the `.config` directory. That may seem strange if you aren't familiar with the X Desktop Group (XDG) Base Directory Specification[148]. The XDG standard defines specifications for file and file formats, and where they should be located. This is common among most macOS and Linux platforms, with some application level support on others. This standard looks familiar if you've ever worked with Windows `APPDATA` files and folders, as it separates configuration files for applications from the data that the applications store on the filesystem. For this scenario, it just means PowerShell will locate its configuration, history, and user data according to XDG specifications.

What does this mean in practice? If you are a Windows PowerShell user coming to PowerShell Core, your profile knowledge can be mapped directly to PowerShell Core profiles on other platforms, with only having to account for slight folder location differences. If you are a *nix user coming to PowerShell, the profile locations should look familiar and not require much effort to adopt. In conclusion, something for everyone.

Module Differences

Where are my Modules? What Modules can I use?

[148] https://specifications.freedesktop.org/basedir-spec/basedir-spec-latest.html

Module Installation Paths

PowerShell Core Modules are located in different directories than in Windows PowerShell. They're installed to `$home/Documents/PowerShell/Modules`. PowerShell Core Modules are also located in different directories on non-Windows platforms. PowerShell Module installation locations follow XDG Base Directory specifications[149]. This means that they're installed to `~/.local/share/powershell/Modules`.

Module Loading Differences

Since PowerShell Core installs modules to different locations than Windows PowerShell, it also loads modules differently than Windows PowerShell. It does still use the `$env:PSModulePath` environment variable, but it's populated with different values in PowerShell Core.

On Windows:

```
$env:PSModulePath -split ';'
```

Output:

```
C:\Users\james\Documents\PowerShell\Modules
C:\Program Files\PowerShell\Modules
c:\program files\powershell\7-preview\Modules
C:\WINDOWS\system32\WindowsPowerShell\v1.0\Modules
```

On Mac and Linux, they're in slightly different places:

```
$env:PSModulePath -split ':'
```

Output:

```
/home/james/.local/share/powershell/Modules
/usr/local/share/powershell/Modules
/opt/microsoft/powershell/6/Modules
```

> Note how we had to use the `:` path separator with `$env:PSModulePath` instead of the `;` separator. We'll get to how to handle this difference later on in the chapter when we talk about handling paths in Beware Paths.

[149] https://specifications.freedesktop.org/basedir-spec/basedir-spec-latest.html

Using Windows PowerShell Modules in PowerShell Core

So if PowerShell Core installs and stores modules in different locations than Windows PowerShell, how do you use Windows PowerShell Modules in PowerShell Core? Turns out it's harder than just prepending `$env:PSModulePath` with a valid Windows PowerShell path and telling PowerShell Core to ignore Windows PowerShell based modules with `SkipEditionCheck`.

Most Windows PowerShell modules can work in PowerShell Core with minimal to no modification, if they kept to using PowerShell cmdlets or base .NET API. If the Windows PowerShell modules used Windows specific API like COM or WMI, then the module won't run in PowerShell Core. Those aren't supported as they couldn't be run cross-platform. So are we out of luck if we want to continue to use our existing Windows PowerShell Modules?

Enter the WindowsCompatibility Module. This module lets PowerShell Core access Windows PowerShell Modules that aren't yet natively available on PowerShell Core. How can it do that if the module isn't supported? Through the power of Implicit Remoting! This feature has been around since PowerShell version 2, but isn't used as much as it should. It uses PowerShell Remoting to provide wrapper cmdlets and functions that serialize the requests and responses. This allows PowerShell Core to invoke the Windows PowerShell module, without requiring you to do anything differently.

All good right? Well, there are some caveats. This is a solution for Windows platforms only as it uses Windows Remote Management (WinRM), you won't be able to use this to run Windows PowerShell cmdlets on macOS or Linux platforms. Since it's a serialized remoting approach, no GUI applications or use of Windows Presentation Foundation (WPF) or Windows Forms are allowed. It also requires the latest version of PowerShell Core version 6.1. Even with all of the caveats, this is a major improvement and lights up Modules that aren't strictly supported. This is an important bridge to the time when the Module finally becomes PowerShell Core compliant.

Notes From the Field

Ok, we've gotten past the shiny new cross-platform stuff and some of the gotchas, what are some things to be aware of using PowerShell Core?

Aliases Aren't Your Friend

If you've been a PowerShell user for any length of time, you've used aliases in the terminal and in scripts. You've developed opinions on their use and when to use them, and recognized that the community has a large array of opinions on whether or not to use them. PowerShell Core makes this topic even harder because of its multi-platform nature.

Aliases are great in interactive use, when you're typing fervently at the console trying to make something work. They're not so great when you're reading the production deploy script at 4 AM trying to figure out what that three letter abbreviation means. It's a long held stricture in the community, first echoed by Jeffery Snover, that you should be pithy at the command line and verbose in your script. Jeffery first coined the phrase to describe what PowerShell allows you to do, compared to other programming languages, but in this scenario it's a good idea to follow.

Why the long preamble? In the case of PowerShell Core, it's even more important to be deliberate in what you want executed because you aren't running on 'just Windows' anymore. A command like `ls` doesn't behave the same way in macOS as it does on Windows. It points to the binary `ls` which has different parameters and text output. Other aliases conflict with names of standard Unix utilities like `cat` and `curl`. In the past `cat` mapped to `Get-Content`, which would cause problems when running on Linux when it returns newlines with carriage returns instead of just newlines.

So, what do we do about it? In general, it will be a personal choice, but there is some community discussion on paths forward. RFC #129[150] has initial discussions on removing or dealing with aliases. #8970[151] is among many requests asking for decouple/removal.

Platform Variables

Some PowerShell Core cmdlets may have abilities to detect and handle different platforms, but what if you really really need to know what platform you're running on in order to do decide to do one thing or another? PowerShell Core has you covered with some built-in variables that are automatically populated for each platform it supports. These variables are all `boolean`, in that they're `true` or `false` depending on what the platform is.

```
PS C:\Users\james> $IsWindows
True
PS /home/james> $IsLinux
True
PS C:\Users\james> $IsMaxOs
False
```

You would use these variables to determine which custom logic is needed for your environment. For example you would use these to determine which locations to pull binaries for, or determining which command to run depending on the OS. This avoids having to know how to query Common Information Model (CIM) for the Windows OS or `uname` for linux, in simple `switch` or `if-else` statements.

And easy snippet to keep handy is below. This `switch` statement will determine which platform you are on, and only run the code in appropriate scriptblock.

```
switch($true){
  $IsWindows { 'PSCore on Windows!' }
  $IsLinux   { 'PSCore on Linux!' }
  $IsMacOS   { 'PSCore on MacOS!' }
}
```

These built-in platform variables won't work for more complicated scenarios where you need to know which linux distribution you're running in or whether you're running on Windows 2012R2 or 2016. When you get to those scenarios, you drop down to the raw query commands for your platform.

[150] https://github.com/PowerShell/PowerShell-RFC/pull/129
[151] https://github.com/PowerShell/PowerShell/issues/8970

Case Sensitivity

Dealing with case sensitivity is a recurring theme when working cross-platform, no matter the language you use. Case sensitivity means differentiating between upper case and lower case characters. For example, 'Windows' is different that 'windows'.

You probably know that Windows is case *insensitive* and Linux is case *sensitive* already. What you may not know is that Windows is case *insensitive* while *still* preserving case. To further complicate things, in general PowerShell is case *insensitive*.

All of this means that PowerShell is case *insensitive* but your platform may *not be*, and you have hidden bugs waiting to happen! What does this mean for how you use PowerShell Core? It means you have to start accounting for human error or variance in everything from file paths to user input that can depend on character casing. What's a person to do?

We'll cover two areas to be most concerned about regarding case sensitivity: Environment variables and File Paths.

Environment Variables

PowerShell tries very hard to 'Do the Right Thing', but in the case of `Environment Variables` there are some things it can't shield you from regarding case sensitivity. On linux and macOS all environment variables are upper case with exception of `PSModulePath`. This variable was deemed important enough to break the convention and set as camel-cased no matter the operating system. This allowed scripts to work using `PSModulePath` no matter the platform it was running on.

File Paths

Again, PowerShell tries very hard to 'Do the Right Thing', and in the case of file paths it does a pretty good job of protecting you from implementation details.

Here are some general rules to follow:

- In general, rely on the PowerShell Core system to validate paths by using the built-in cmdlets.
- `Resolve-Path` is your friend when trying to ensure the case in paths is correct no matter the platform.
- When it doubt, use `Test-Path` to determine both if a path is present and valid case at the same time.
- If you have to compare paths as `strings` yourself for some reason, use case insensitive comparisons like `-ieq`.
- Always remember to check your regex statements for case sensitivity.
- When using the interactive shell, PowerShell Core on Linux won't tab complete incorrect cases

When to Use Shebangs

A shebang, or hashbang, is a sequence of characters beginning with a number sign and exclamation mark that indicate the file is to be used as if it's an executable. In general, the characters after the `#!` are parsed as a path to the thing that will execute the file, and the contents of the file as the code to execute. There are some variants of this workflow, but for most systems that's how it works.

In PowerShell Core 6.0, support for shebangs was added by changing the first positional parameter for `pwsh` to `-File` from `-Command`. This allowed you to execute `pwsh foo.ps1` or `pwsh someFile` without specifying `-File` like this `pwsh -File foo.ps1`.

In order for `pwsh foo.ps1` to work, you have to add `#!/usr/bin/env pwsh` as the first line of your PowerShell script:

```
#!/usr/bin/env pwsh

# awesome pwsh code here
Write-Host 'awesome'
```

Let's break this down a bit, using our definition above. We know that `#!` indicates we're using a shebang, that the path to the executable follows it `/usr/bin/env`, and that the executable to run is last `pwsh`.

Why use `/usr/bin/env`? For most platforms, that's where the symlink to the PowerShell install will be. As a bonus, running it on Windows works with that path too:

```
Microsoft Windows [Version 10.0.18362.239]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\james>

C:\Users\james>pwsh foo.ps1
awesome

C:\Users\james>
```

Having a shebang allows you to have a truly cross-platform script that runs anywhere PowerShell Core is installed.

Git Hooks

So, even with all this, when would you want to use shebangs with PowerShell Core? One increasingly common use case is git hooks. Git hooks allow you to run custom scripts whenever certain events occur (committing, merging, pushing, etc.) when using git. When git was written, an unfortunate assumption was made that all git hooks would be bash scripts. This is unfortunate because it restricted the ability to use git hooks on different platforms, as not all platforms come with bash, never mind bash not being prevalent on Windows.

A workaround was to have your git hook call the bundled bash shell inside Git, then spawn PowerShell from there:

```
#!C:/Program\ Files/Git/usr/bin/sh.exe
exec powershell.exe -NoProfile -ExecutionPolicy Bypass \
  -File ".\.git\hooks\pre-commit.ps1"
```

Since Git and bash doesn't understand PowerShell, we have to store the PowerShell code in a separate PowerShell script file: `.\.git\hooks\pre-commit.ps1`.

```
# Verify user's Git config has appropriate email address
if ($env:GIT_AUTHOR_EMAIL -notmatch '@(non\.)?acme\.com$') {
    # super cool powershell code
    exit 1
}
exit 0
```

This is sub-optimal for several reasons. This requires multiple files, both the git hook itself *and* the PowerShell script. The path to Git's `sh` is hardcoded, so it has to be the same every system that runs this, and it relies on `powershell.exe` to be in *PATH*, although you could have hardcoded that path too, but that isn't any better.

With PowerShell Core, there is a cleaner alternative:

```
#!/usr/bin/env pwsh

# Verify user's Git config has appropriate email address
if ($env:GIT_AUTHOR_EMAIL -notmatch '@(non\.)?acme\.com$') {
  # super cool pwsh code
  exit 1
}
exit 0
```

With PowerShell Core, we can put the PowerShell code directly in the git hook file, and place a shebang to instruct the system where to find the shell to run the code. In the end, this could prove to be a better solution than bash, as this is truly cross platform code. Install PowerShell core on all of your build and development systems, and the git hooks work no matter where they're run.

Dealing with Files

File Encoding

When you start to create files that have to be able to be read on multiple platforms, you quickly realize that reading the content of a files is much harder than it sounds.

When creating files `UTF-8` is the best choice for cross-platform user, with `ASCII` being your last best hope. `UTF-8` is readable by most modern applications, and handles the most characters in a

predictable way. `ASCII` is the best fallback for interoperability, as it restricts what characters it supports so less things go wrong.

If you set the default parameters for cmdlets that use encoding like

```
1 $PSDefaultParameterValues["Out-File:Encoding"] = "UTF8"
```

in your profile or at the beginning of your scripts, you can avoid having to remember to set encoding correctly for cmdlets that use encoding.

File Newlines and Line Endings

The only thing as hotly debated as line endings is tabs vs spaces. We won't get into value judgments here, just an exploration of what to consider.

There are two choices to use when writing newlines in files: `\n` and `\r\n`.

Most Mac and Windows applications will accept `\n` but Unix shell interpreters will fail to read `\r\n`. This is especially important in the `shebang`, as it will prevent the entire file from being read instead of just having weirdly terminated lines in output.

If you are writing scripts that will be read or executed on multiple platforms you will have to start standardizing on Unix-style line feeds. It ensures that all platforms will read and understand your text, and most modern editors allow you to configure linefeed defaults. This, among many other reasons, is a reason to move off of the PowerShell Integrated Scripting Environment (ISE).

File Access

Continuing on the theme, like with the path cmdlets and environment variables, PowerShell Core knows how to handle file access across platforms. For most cases, use the built-in cmdlets for reading and writing files as they will know the variances between OS. For high performance read/write scenarios, you'll have to drop down to native APIs, but for your typical input/output workloads the content cmdlets work fine.

However, beware the differences in platform support of cmdlets like `Get-ChildItem`. You will get back filesystem objects, but they won't have paths you expect if you're used to Windows Paths. Also be careful of aliases, if you still have `ls` defined somewhere and a `ls -la` is used you'll have a bad time.

Beware Paths - Building Them

PowerShell Core has adapted itself to running on platforms that use different path indicators and separators, and so you need to start using the built-in functionality to do the same.

When building paths, use `Join-Path` and other path cmdlets like `Test-Path`, `Split-Path`, and `Resolve-Path` to do the hard work of knowing what path separator to use on which platform. Don't build the path strings yourself using string interpolation like `"C:\$examplePath\$anotherExamplePath"`. I can almost guarantee you that you will miss that one corner case where it's a backslash and not a forward slash. Besides being brittle, using string

interpolation makes you do all the work. Why add extra custom logic to detect if it's linux or windows when a simple `Join-Path` works the same, is less code, and is reliable?

When coding your scripts, be aware that the platform you're sitting on may not be the platform the code is run on. Your code has to be path agnostic, so stick to concepts like adding [FileInfo] typed parameters to your function or scripts. Don't assume the primary drive is `C`, use environment variables like `$env:SystemDrive` to determine if it's not `/` instead. While we're discussing default drives, don't assume all programs are installed in 'C:Program Files', use environment variables like `$env:ProgramFiles` to determine where things are.

Beware Paths Part Deux - Accessing

`[System.Environment]::CurrentDirectory` isn't set in .Net Core, so you can't rely on it for the correct directory.

If you need to call path-sensitive .Net APIs, you need to use

```
[System.IO.Directory]::SetCurrentDirectory(
    (Get-Location -PSProvider FileSystem).ProviderPath
)
```

first to update the .NET environment. This will set .NET's location to the same thing that PowerShell thinks is the location. This is similar to the problems that have existed in PowerShell since version 1.

Note - PowerShell sets the working directory correctly when launching applications.

VS Code Config

Visual Studio Code[152] (VSCode) is a great PowerShell editor, and with a few customizations can be an efficient tool to use.

Extensions

First off, install the PowerShell Extension[153]. If you need help installing the extension, this link has resources for the many ways to install the extension. This VSCode extension provides syntax highlighting, intellisense, and many more modern Integrated Development Environment (IDE) features. If you're a PowerShell ISE fan, it even comes with a PowerShell ISE theme to make you feel at home.

Outside of PowerShell, you should take a look at these other VSCode extensions that will improve your day to day coding experience:

- Gitlens[154]

[152] https://code.visualstudio.com
[153] https://marketplace.visualstudio.com/items?itemName=ms-vscode.PowerShell
[154] https://marketplace.visualstudio.com/items?itemName=eamodio.gitlens

- Bracket Pair Colorizer[155]
- Docker[156]

VSCode has something for everyone, it even has something for vim users[157]!

Configuration

VSCode works for PowerShell coding out of the box, but to truly get the most of out your time here are some configuration suggestions for cross-platform use. The examples below are all JSON output, which you can reach inside VSCode by executing `Ctrl+Shift+P` and typing `settings`, then choosing `Preferences: Open Settings (JSON)`. You could use the graphic settings editor `Preferences: Open Settings (UI)` if you prefer.

A general purpose VScode settings looks like the following code block. It uses all the information we've covered in this chapter to configures VSCode for cross-platform coding.

```
{
  // Put a line number in the gutter for the last line
  "editor.renderFinalNewline": true,
  // Show all whitespace. Alternatively could use boundary to still show
  // spaces, but not leading spaces
  "editor.renderWhitespace": "all",
  // Removes trailing whitespace that was added by auto complete
  "editor.trimAutoWhitespace": true,
  // Use cross-platform compatible UTF8 file encoding
  "files.encoding": "utf8",
  // Set newlines to Linux format
  "files.eol": "\n",
  // Add an empty line at the end of a file
  "files.insertFinalNewline": true,
  // Ensure there is only one empty line at the end of a file
  "files.trimFinalNewlines": true,
  // Remove any extra whitespace to make git diffs easier
  "files.trimTrailingWhitespace": true,
}
```

In summary, the above configures VSCode to use `UTF8` encoding, Linux newlines, insert an empty newline at the end of every file, show all whitespace (non character space), and trim extra whitespace and newlines.

If these config items conflict with ones you previous set, you can scope these to the language level in VSCode. This will make VSCode use these settings for PowerShell, while applying the others for other languages.

[155] https://marketplace.visualstudio.com/items?itemName=CoenraadS.bracket-pair-colorizer-2
[156] https://marketplace.visualstudio.com/items?itemName=ms-azuretools.vscode-docker
[157] https://marketplace.visualstudio.com/items?itemName=vscodevim.vim

```
{
  "[powershell]":{
    "editor.renderFinalNewline": true,
    "editor.renderIndentGuides": true,
    "editor.renderLineHighlight": "all",
    "editor.renderWhitespace": "all",
    "editor.trimAutoWhitespace": true,
    "files.encoding": "utf8",
    "files.insertFinalNewline": true,
    "files.trimFinalNewlines": true,
    "files.trimTrailingWhitespace": true,
  },
}
```

> Notice that we can't set `"files.eol": "\n"` in the PowerShell language scoped section. Setting line endings is a global VSCode setting, and can't be set per language.

The PowerShell VSCode extension can be used without configuration, but just like VSCode, can be optimized to your usage patterns. The following settings ensure that the PowerShell extension will load when a PowerShell file is opened, and set the integrated console to not show on startup. This is a personal preference of mine, as I find the terminal popping open every time a PowerShell file is opened jarring.

```
{
  "powershell.startAutomatically": true,
  "powershell.integratedConsole.showOnStartup": false,
}
```

Choosing what version of PowerShell the extension uses when it's parsing your scripts and in the integrated console used to be difficult. The settings required you to type the paths to the PowerShell binary by hand, and were error prone. You can still do that, or you can click the `PowerShell` icon in the lower right click corner which will present you with a popup window showing you the different PowerShell versions installed on your system. This results in the following config entry:

```
{
  "powershell.powerShellExePath":
    "C:\\Program Files\\PowerShell\\7-preview\\pwsh.exe",
}
```

One great feature of the PowerShell extension is the code formatting feature. This automatically fixes your code to align with PowerShell coding standards. If you find you are disagreeing with some of those choices, there are formatting options to consider:

```
{
  // Use correct casing for cmdlets.
  "powershell.codeFormatting.useCorrectCasing": true,
  // Adds a space after a separator (',' and ';').
  "powershell.codeFormatting.whitespaceAfterSeparator": true,
  // Adds spaces before and after an operator ('=', '+', '-', etc.).
  "powershell.codeFormatting.whitespaceAroundOperator": true,
  // Adds a space before and after the pipeline operator ('|').
  "powershell.codeFormatting.whitespaceAroundPipe": true,
  // Adds a space between a keyword and its associated scriptblock expression.
  "powershell.codeFormatting.whitespaceBeforeOpenBrace": true,
  // Adds a space between a keyword and its associated conditional expression.
  "powershell.codeFormatting.whitespaceBeforeOpenParen": true,
  // Adds a space after an opening brace and before a closing brace
  "powershell.codeFormatting.whitespaceInsideBrace": true,
}
```

The `"powershell.codeFormatting.useCorrectCasing"` setting is an interesting feature. It's disabled by default, but once enabled with automatically convert all of the references to cmdlet to their proper case and full name!

Another useful feature is the real-time script analysis from the PowerShell Script Analyzer. You can set the path to a version you have installed, so you can control what version your project uses.

```
{
  // Enables real-time script analysis from PowerShell Script Analyzer.
  "powershell.scriptAnalysis.enable": true,

  // Specifies the path to a PowerShell Script Analyzer settings file
  "powershell.scriptAnalysis.settingsPath": "",
}
```

Wrap-up

In this chapter we've covered PowerShell Core daily use operations and some hard-won notes from field. Through it all, we've examined how to do things with an eye for how to use PowerShell Core on different platforms. While whether you can use PowerShell Core as your daily use shell largely is determined by your use cases, I think what we've covered here shows that PowerShell Core is definitely a shell and language that's successfully designed for use cross-platform.

PowerShell Development on Containers (PowerShell Core)

by Saggie Haim

With PowerShell V7 on the verge, the importance of building solutions that are cross-platform in PowerShell is greater than ever. One issue running tests is that your local PowerShell environment isn't "clean."

You install modules and software and change configurations that your target audience or servers won't share. Your solution can act differently in those environments, and you also risk not understanding the full dependencies list and limitation of a target environment. That's why you need a clean PowerShell environment.

Sometimes you may just want to try the new PowerShell versions or maybe test how it's affecting your tools before making the switch. While installing a new server, whether virtual or physical is a time-consuming task, running a clean PowerShell core in a container can take two minutes or less.

What Containers Are

Containers provide the ability to run an application in an isolated environment. The isolation and security of containers allow you to run many containers simultaneously on your PC. Containers are lightweight because they don't require the extra load of a hypervisor and run directly within your machine's kernel. This means you can run more containers on a given hardware combination than if you were using virtual machines.

Prerequisites

Before you can run containers, you need a containers engine. The most popular one is Docker. However, you can't use Docker on Windows 10 home edition due to Hyper-V limitations.

Docker for Windows

- Windows 10 64-bit: Pro, Enterprise or Education (1607 Anniversary Update, Build 14393 or later).
- Virtualization enabled in BIOS. Typically, virtualization is enabled by default. This is different from having Hyper-V enabled.

- CPU SLAT-capable feature.
- At least 4 GB of RAM.

Read more about Windows Prerequisites and installation guide on the Docker website[158].

Docker For Linux

- A 64-bit installation
- Version 3.10 or higher of the Linux kernel. The latest version of the kernel available for your platform is recommended.
- `iptables` version 1.4 or higher
- `git` version 1.7 or higher
- A `ps` executable, usually provided by `procps` or a similar package.
- XZ Utils 4.9 or higher
- A mounted `cgroupfs` hierarchy; a single, all-encompassing `cgroup` mount point isn't sufficient.

Read more about Linux Prerequisites and installation guide on the Docker website[159].

If you meet the prerequisites, install the Docker engine from the Docker site.

Starting with Docker

Docker uses images to build containers. A container is a runnable instance of an image. An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization. In our example, the PowerShell image is based on Linux, with PowerShell installed.

[158] https://docs.docker.com/docker-for-windows/install/
[159] https://docs.docker.com/install/linux/docker-ce/binaries/

```
PS /> $PSVersionTable

Name                           Value
----                           -----
PSVersion                      6.2.1
PSEdition                      Core
GitCommitId                    6.2.1
OS                             Linux 4.9.125-linuxkit
Platform                       Unix
PSCompatibleVersions           {1.0, 2.0, 3.0, 4.0…}
PSRemotingProtocolVersion      2.3
SerializationVersion           1.1.0.1
WSManStackVersion              3.0
```

You can create, start, stop, move, or delete a container. You also can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state. A container is defined by its image and any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that aren't stored in persistent storage disappear.

Pull Docker Image

Before you can create any containers, you need to pull the image you want to use from the Docker hub repository. Each image link is built by two parts:`image URL:tag`. Image tag defines the version or configuration variation of the image. You can find more info about them in the image repository source. Begin with opening PowerShell and use the `Docker pull` command to get the image you want:

Latest *Stable* Edition of PowerShell

```
Docker pull mcr.microsoft.com/powershell:latest
```

Latest *Preview* Edition of PowerShell

```
docker pull mcr.microsoft.com/powershell:preview
```

You can choose what best fits your needs. When the Docker engine finishes downloading the image, you can move to the next step.

Creating a Container Based on PowerShell Image

Now, you can create containers based on the image and image tag you pulled. Create a new container using the `Docker run` command:

```
Docker run --name ps-core --interactive --tty mcr.microsoft.com/powershell:latest
```

You can use the **preview** tag if you prefer.

Because you used the `--interactive --tty` switches, when the command executes, the session is switched to the container session context.

Managing Existing Containers

Finally, when you finish your tests, you need to decide what to do with the container. The container is persistent, which means you can continue work on it, even if its stopped. All the data will stay until you delete the container. Do you want to keep it for future purpose? Or you don't need it anymore and want to delete it?

Delete the Container

If you don't need the container anymore, you can remove it. Pass the ID or the name of the container to the `Docker rm` command:

```
Docker rm 'Container name'
```

If you don't remember the container ID or Name, you can get it with the command `Docker ps`.

```
Docker ps
```

The default behavior for Docker ps command is to show only running containers. You can add the `-a` or `--all` switch to list both running and stopped containers.

Keep The Container

If You decide to keep the container, you can re-use it at any time. First, you need to make sure the container is running. You can use the `Docker ps` command to check it. If it's not running, you can start it with the command `Docker start`.

```
Docker start 'Container Name'
```

Now to actively interact with it, you use the `Docker attach` command.

```
Docker attach 'Container Name'
```

You can also use the container ID if you prefer.

You can create a temporary container with: `docker run --rm -it mcr.microsoft.com/powershell:latest`. The `--rm` switch tells the Docker engine to automatically delete it on exit.

Use Visual Studio Code to Connect and Develop on a Container

Now you know how to create containers with PowerShell core inside. But that PowerShell session isn't a development environment. You want to properly develop in an environment that supports development operations like debugging, task running, version control, and more.

The Remote—Containers Extension

The Remote—Containers extension lets you attach VS Code to a running container. The extension lets you work with VS Code as if everything were running locally on your machine, except now they're isolated inside a container. To install the Remote-Containers extension, open the Extensions view by pressing `Ctrl+Shift+X` and search for "Remote-Containers" to filter the results.

Attach VS Code to a Container

You can attach VS Code to a container in three steps:

1. Press F1 to open the command pallet.
2. Type "Remote-containers: Attach To Running Container..."
3. Choose the Container you want to attach to.

The container must be running to attach it.

Now you're connected to the container. You can run your scripts, modules, or any other solution on a clean and isolated environment.

The Docker Extension

The Docker extension makes it easy to build, manage, and deploy containerized applications from VS Code. To install the Docker extension, open the Extensions view by pressing `Ctrl+Shift+X` and search for "Docker" to filter the results. After installing the extension, you can add it to the activity bar by right-clicking on the activity bar and choosing Docker. You can use it to manage your containers, Images, and more.

Manage Containers

In the Docker view, the first box is used for containers. In this box, you can see all the containers available in your PC. Each container has a status icon next to it. Green Play is for running containers and Red Stop is for stopped Containers. By right clicking on a container, you can Attach, Start, Stop, Restart, or remove it.

Manage Images

Under the Images box, you can see which images are available on your PC. Those are the images you can use to create containers. You can expand each image to see which tags you pulled. By right-clicking on the tag of the images, you can remove it, or run a temporary instance container.

When you run an instance by right-clicking on the image tag, it's running it with the `--rm` switch.

Summary

The combined abilities of VS Code as a development environment and the Docker containers as a test environment are endless. When one test environment fails, a new one rises. You may find a new world of possibilities with containers. You can try the newest preview version of PowerShell on containers. You can also try different modules from the internet, without messing up your environment. Most importantly, you can develop and test your solutions much better. Hopefully this chapter will help you start with containers and help you test your solutions better.

Part III: PowerShell Internals

This section highlights the functionality and usability of the language itself, including advanced topics on leveraging it to tackle harder problems than ever before.

Writing Your First Compiled PowerShell Cmdlet

by Thomas Rayner

You're a person who writes PowerShell, right? You write tools and scripts in PowerShell which you and other people run all the time to do cool things. Probably, since you bought this book about PowerShell, you like the language, are looking to learn more about it, and want to take your code and contributions to your company to the next level.

This chapter introduces you to PowerShell cmdlets in C# by going through some relatable examples in .NET Core. You're not learning best practices, necessarily, but you *are* learning some cool and interesting things that you can use right away.

The Whys

You might wonder to yourself, "Why would I ever want to write a PowerShell cmdlet in C#?" and I would simply reply with "Why not?"

There is a long-standing attitude in the operations community that there's some specific difference between a developer and an operator. People believe, for some reason, that developers are these magical people who must have Computer Science degrees, or work in roles titled "Software Engineer." None of that's true.

If You Write Code, You're a Developer

Period. You might not be a *good* developer or a *professional* one, but you're a developer. The act of creating code to run on a computer system is *development*. Therefore, whether or not you want to admit it, when you're writing code you're *developing*. Don't let anybody tell you otherwise.

You don't need unit tests to be a developer. You don't need to write in Python, Javascript, SQL, Rust, or any other language to be a developer. All you have to do is create code, and that's what you're doing pretty much any time you open your editor, right? So, why limit yourself to just PowerShell? There's a whole world of languages out there, and you shouldn't keep yourself from diving in just because you don't meet some arbitrary definition of a "developer."

Why You Should Use C# to Write PowerShell Tools

C# (pronounced "see sharp") is a .NET language just like PowerShell. That makes it easier to pick up than a language that has nothing in common. C# is considered a "lower level" language than PowerShell. What does that mean? PowerShell abstracts a lot of programming concepts away from the user writing code. It was initially built to help system administrators with limited coding skills to automate mundane tasks, which it does! PowerShell, in my opinion, is a gateway language and inevitably leads people towards other languages.

C# is much more flexible because it was designed to work in many more scenarios.

- Linq[160] is much more straightforward in C# and allows you to interact with collections more conveniently.
- Async APIs and APIs that require generics usually work better in C#.
- Don't get me started on multithreading. In PowerShell, it's about as much fun as slamming your hand in a car door, but in C#, multithreading is much more convenient.

Writing your PowerShell cmdlets in C# also offers the benefit of delivering one compiled artifact instead of a bunch of text (by which I mean `.ps1`) files. These compiled artifacts, usually Dynamic-Link Libraries (DLLs), are harder for "power users" to tamper with and can often make it easier for you to work with C# code written by others. That's especially important when you're writing a wrapper for a library someone else created or contributing to an open source project. For example, maybe you're the person adding a PowerShell module to a cool C# tool!

Prerequisites

Before you get started writing any C# code, you need to install a couple things first. There are a lot of options for editors out there. Visual Studio[161] is popular but, if you're into PowerShell, you're probably already using VS Code[162]. Installing the C# extension[163] is all you need to get started in VS Code.

You also need to install a version of the .NET Core Software Development Kit (SDK)[164]. You can use whatever version of the .NET Core SDK you prefer, but be aware that some features in newer versions of .NET Core aren't backwards compatible.

Hello World

As with all things, you have to crawl before you walk and walk before you run. In that regard, this first example will have you crawling all over the place.

[160] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/
[161] https://visualstudio.microsoft.com/
[162] https://code.visualstudio.com/
[163] https://marketplace.visualstudio.com/items?itemName=ms-vscode.csharp
[164] https://dotnet.microsoft.com/download

When you start a new C# project, you need to create a little bit of scaffolding. If you've ever used something like the Plaster module[165], or some other template system to create new PowerShell modules, then this will be a somewhat familiar concept. Basically, you use the `dotnet` command line interface (CLI) tool to create some new artifacts.

First, you create a new `classlib` which is, as the name suggests, a class library.

In C#, a class library is essentially just a collection of related files that compose your entire solution.

Create your new class library, and give it a meaningful name related to what you're working on.

```
dotnet new classlib --name ThomasWritesPerfectCode
cd ThomasWritesPerfectCode
```

In this example I named my class library `ThomasWritesPerfectCode` which, I assure you, is *absolutely* true. Name yours something that's more relevant to your project. Then use `cd` (or `Push-Location` if you're fancy) to navigate to the directory that was created by `dotnet` when you created a new class library. If you don't provide a `--name` value, your class library will be created in your present working directory, with the same name as that directory.

Now, you perform an optional step. It's time to add another artifact to the class library using `dotnet`. This next artifact is a `globaljson` which, while optional, you should include. It's used for a lot of things, including tracking the version of .NET Core used in your project.

```
dotnet new globaljson
```

You don't have to do anything else to your new `global.json` file that you see in the root of your class library folder.

Now, you're ready to write some C#! But, wait. You're writing a PowerShell cmdlet. In PowerShell, when you're writing code that's for another system, often you'll end up having to import a module or specify a `using` statement to include commands and other artifacts from that module in your code. C# requires the same sort of statement.

Rub Some PowerShell On It

Using `dotnet` again, you must add the PowerShell library to your project so you can use all the tools, methods, classes, and other artifacts that the PowerShell team at Microsoft created for people writing their own cmdlets in C#.

```
dotnet add package PowerShellStandard.Library
```

This command will modify your `.csproj` file and instruct your computer to retrieve and add a dependency. Specifically, a dependency on the bits needed to write a PowerShell cmdlet.

[165] https://github.com/PowerShell/Plaster

Writing Some Code

Open the folder you created in VS Code (or your preferred editor of choice). There are already a few files in here, some with bad names. Specifically, rename `Class1.cs` to something more related to the specific cmdlet you're making. In this example, it's renamed to `GetGreeting.cs`.

Think of the name of your class library as the name of your module, and the names of your `.cs` files as the cmdlets inside that module.

Open your newly renamed file, and change the line that reads

```
1 public class Class1
```

to

```
1 public class GetGreeting
```

Or, more specifically, to match the name of your `.cs` file. It's in your best interest to use meaningful names and keep yourself well organized.

Now, it's time to include the specific pieces of code that allow the `.dll` that you're creating to run in PowerShell. At the very top of your currently open file, it says `using System;`. Immediately following that line, add `using System.Management.Automation` so that the first two lines look like this:

```
1 using System;
2 using System.Management.Automation;
```

Don't forget your semicolons. C# needs them to denote the end of a line, unlike PowerShell.

Next, it's time to identify the class `GetGreeting` as a PowerShell cmdlet within this module. First, append `: PSCmdlet` to the end of the line that instantiates the class, you edited earlier.

Next, insert a line immediately above that one—this is where you name your cmdlet. The name for this cmdlet is going to end up as `Get-Greeting`.

```
1 [Cmdlet(VerbsCommon.Get, "Greeting")]
2 public class GetGreeting : PSCmdlet
```

There's a lot to unpack here. The first line adds an attribute to your class, the `Cmdlet` attribute. It takes a few arguments, but you only need two to give your cmdlet a name. The first is a verb. This is one of the approved verbs that come with PowerShell, which have been conveniently organized for you. For this example, you want to choose the `Get` verb. The second argument is a noun, which should be singular and doesn't come from an approved list. My noun here is `Greeting`. Hence, the name of this cmdlet will be `Get-Greeting` since the verb and noun will be combined to fit in with all the other PowerShell cmdlets you know and love.

Right now, your `GetGreeting.cs` file should look like this:

```
using System;
using System.Management.Automation;

namespace ThomasWritesPerfectCode
{
    [Cmdlet(VerbsCommon.Get, "Greeting")]
    public class GetGreeting : PSCmdlet
    {
    }
}
```

Ship It

This module is *technically* ready to be built and deployed. In a PowerShell console, in the directory that has your class library, run the following command to build your solution:

```
dotnet build
```

Unless you made a typo somewhere, you should have output that describes where it created `ThomasWritesPerfectCode.dll` (or whatever you named your class library), and a portion that lets you know that you have zero errors or warnings.

You can even import this module and run your new cmdlet (remember to specify the full or relative path to the DLL):

```
Import-Module $pathToYourDLL
Get-Greeting
```

The only problem is that nothing happens when you run `Get-Greeting`. What happened there? Well, you haven't actually made your cmdlet do anything yet. After your class declaration (where you appended `: PSCmdlet`), there's an opening curly brace (`{`), and that's where code gets added.

Make Some Magic Happen

It's time to make your new cmdlet do something. This is where it pays to be a little familiar with C#, but it's alright if you're not. C# is .NET based, just like PowerShell, so you'll find yourself looking up syntax and finding your way around the basics, but you'll pick it up quickly.

Classes in C# can have properties. A property is just a place holder for a variable. You've worked with properties before on objects in PowerShell. For instance, the `Length` property on a variable that holds a string tells you how long the text is.

```
$myString.Length
```

In C# PowerShell cmdlets, parameters are properties, although they're a little special. Add the following inside of the curly brace that comes right after the class declaration.

```
[Parameter()]
public string Name { get; set; }
```

For now, you can safely ignore the curly braces and the `get` and `set` parts. You'll learn more about that as you get to know C#. The first line is an attribute that applies to the line that comes after it. That attribute simply identifies the property on the second line as a parameter for this cmdlet. The second line states that the `Name` parameter is of type `string` and is `public`, which is normal.

In PowerShell you might have written this instead:

```
[CmdletBinding()]
param (
    [Parameter()]
    [string]
    $Name
)
```

But you didn't, because this is C#. Your `GetGreeting.cs` file should now look like this:

```
using System;
using System.Management.Automation;

namespace ThomasWritesPerfectCode
{
    [Cmdlet(VerbsCommon.Get, "Greeting")]
    public class GetGreeting : PSCmdlet
    {
        [Parameter()]
        public string Name { get; set; }
    }
}
```

The point of adding this parameter for this example is so that your cmdlet can either write out `Hello World` or it can write out `Hello Name` where the name passed to the cmdlet is used. You didn't make that parameter mandatory (you could have put `Mandatory = true` inside the round brackets in the `Parameter` attribute), so you must handle cases where the user doesn't specify a `Name`.

Now it's time to construct the message that will be sent back to the user. You'll need to break it down into its parts. First, determine the subject. Did the user provide a value for the `Name` parameter? Or did they leave it blank, therefore implying the valid output is `Hello World`? Add this line below the parameter declaration to fill in the subject being greeted.

```
string subject = string.IsNullOrEmpty(Name) ? "World" : Name;
```

This line declares a new variable of type `string`. It's being set equal to a value that looks a little weird if you've only ever worked in PowerShell. This is called a *ternary* statement and it works a lot like an `if/else` does in PowerShell. The first part, `string.IsNullOrEmpty(Name)` returns true or false depending on if the `Name` variable contains a value or not. If `Name` doesn't contain a value, it will return true, and the part between the `?` and the `:` will be returned, in this case "World" is what would be returned. If `Name` does contain a value, the part after the `:` will be returned, in this case, the value for `Name`.

But Wait, There's a Bug

A bug already? If you're following along with this example, you noticed that your editor is unhappy with you. In C#, operations like the one you're doing right now to create variables and programmatically determine their values need to be inside of a *method*. Methods in C# are just like functions in PowerShell. There's a specific method that you need to use here, and it might look a little familiar.

Above the line you just made to declare the `subject` variable, add the following.

```
protected override void ProcessRecord()
{
}
```

Then, put the line declaring `subject` inside the curly braces. This is roughly equivalent to the `process` block of a PowerShell function. You can't get away without using begin, process, and end blocks here like you can in PowerShell.

Your `GetGreeting.cs` file should now look like this.

```
using System;
using System.Management.Automation;

namespace ThomasWritesPerfectCode
{
    [Cmdlet(VerbsCommon.Get, "Greeting")]
    public class GetGreeting : PSCmdlet
    {
        [Parameter()]
        public string Name { get; set; }

        protected override void ProcessRecord()
        {
            string subject = string.IsNullOrEmpty(Name) ? "World" : Name;
        }
    }
}
```

Now that you've got the *who* you're greeting figured out, it's time to construct the greeting and send it to the user.

Start by declaring another variable right below the one that holds the subject, and use string concatenation to make your greeting.

```
string greeting = "Hello " + subject;
```

Then, send that greeting to the user. In PowerShell, you'd normally use `Write-Output` to send this greeting to the user, but since this isn't PowerShell, you need to use something else. You're still *in* PowerShell when you're running this cmdlet, so you still want your greeting to go to the output stream. There's just a different way of doing that in C# than there is in PowerShell.

```
WriteObject(greeting);
```

There is also `WriteVerbose()` and `WriteDebug()` and methods to write to all the different output streams you're used to. Since this is just standard output that's destined for the output stream, you can just use `WriteObject()`.

Your `GetGreeting.cs` file should now look like this if you've been following along:

```
using System;
using System.Management.Automation;

namespace ThomasWritesPerfectCode
{
    [Cmdlet(VerbsCommon.Get, "Greeting")]
    public class GetGreeting : PSCmdlet
    {
        [Parameter()]
        public string Name { get; set; }

        protected override void ProcessRecord()
        {
            string subject = string.IsNullOrEmpty(Name) ? "World" : Name;
            string greeting = "Hello " + subject;
            WriteObject(greeting);
        }
    }
}
```

Ship It for Real

Now, run `dotnet build` again to recompile your project.

Did you get an error that you couldn't build your solution? The DLL that was created earlier, that you imported into your PowerShell session is probably still locked. Close that PowerShell window and open a new one.

Once your code compiles without error, import the new DLL (in the same location), and you can witness the glory of your very first compiled PowerShell cmdlet.

```
dotnet build
# No errors!
Import-Module $pathToDLL
Get-Module
```

Running the `Get-Module` command returns the usual suspects (things like `Microsoft.PowerShell.Utility` and `PSReadLine`), and also shows your new module and the exported commands. Run
`Get-Command -Module ThomasWritesPerfectCode`
and explicitly see the *Get-Greeting* command that your compiled module makes available.

Take This Puppy for a Spin

Run `Get-Greeting` in your PowerShell window you imported your new module into. If your code looks like my code, it returned `Hello World`! Happy days! Rejoice!

But wait, you added another feature. Now call `Get-Greeting` again, but this time provide a value for `-Name`.

```
Get-Greeting -Name Thomas
# returns "Hello Thomas"
```

Amazing! You did it. You wrote code in C# that executes in a PowerShell console. You've written (and run) your very first compiled PowerShell cmdlet.

Summary

This chapter showed you the basics of writing compiled PowerShell cmdlets in C# You saw how to get up and running, how to get started writing C# for PowerShell, and how to build and run your compiled modules. This is just the tip of the iceberg, though!

From here, you're invited to dig deeper on your own. Earlier in this chapter, there is a list of things that are generally easier, or at least better suited for C# than PowerShell. Did any of those things tickle your fancy? Try them out! Worst case scenario, you learn something new.

Writing compiled PowerShell cmdlets isn't a one-size-fits-all solution, just like PowerShell isn't a one-size-fits-all solution. The point of learning how to do this is to expand your skill set and add tools to your proverbial toolbox. There's no reason to let a job title hold you back.

There's no reason you can't get started right now.

PowerShell's Tokenizer

by Joel Sallow

The PowerShell language is made up of atomic units called **tokens**. The `Tokenizer` class is responsible for scanning all text given to PowerShell, whether directly via the command line or from a script file, and creating tokens from the text. This chapter peeks into how the tokenizer works and how knowledge of the tokenizer's functionality and limitations can help you gain a more effective understanding of the PowerShell language.

This will merely scratch the surface to give you an overview of how the tokenizer operates, and how to recognize the different parsing modes and when each applies. Further study is highly recommended; the source code is openly available on Github for you to examine. However, should you wish to have a really thorough understanding, it's recommended you take a moment to clone the PowerShell repository on Github. This will allow you to easily follow references throughout the PowerShell engine internals as the chapter progresses.

Please note that all links will reference commits that are recent as of the time of writing. Feel free to jump ahead to the most recent commit to ensure the code you're looking at is current if you wish. If you do, be aware that all code referenced in this chapter may refer to slightly different line numbers, and parts of the code may have changed since this chapter was written.

Context

For the tokenizer to *work* it needs something to work *with*. For context's sake here is a brief overview of the context the tokenizer primarily works within, so that when one of these things is referenced you have a basic idea of what's discussed.

- `Token.cs`[166]
- Contains the definitions of the `Token` type itself, along with the necessary helper data types like `TokenKind` and `TokenFlags` which define the various distinct types of tokens and how each token behaves.
- `CharTraits.cs`[167]
- Defines the basic character traits for all "special" characters recognized by PowerShell, which directly influences how the tokenizer is able to group and separate characters into distinct tokens.
- `Parser.cs`[168]
- Takes the sequence of tokens from the tokenizer and builds them into the Abstract Syntax Tree, an abstract representation of PowerShell code, which is able to be compiled into executable code.

The tokenizer itself is contained within `tokenizer.cs`[169].

It's recommended that you follow along as this chapter goes through the code, as line numbers are referenced frequently to keep things as succinct and clear as possible. Enough chit-chat; time to look at how the tokenizer works!

Helper Classes and Enums

DynamicKeyword (Ln. 19–481)

The first thing you'll notice when you pull up `tokenizer.cs` in the PowerShell repo is the `DynamicKeyword` class, and a few associated enums related to the necessary parsing of these tokens. Very few people have really heard of — much less used — a `DynamicKeyword`. They're quite old in terms of PowerShell features, but they'e rather tricky to work with, and perhaps because of this (at least in part) you won't see them used frequently.

`DynamicKeyword` code will crop up throughout the tokenizer, but this chapter won't get too deep into that. All you need to worry about here is that these things exist and can be used, and the tokenizer needs to be able to recognize them. They work very similarly to regular keywords, just with a little extra complexity to allow some room for customization, thus the enums and the relative complexity of the `DynamicKeyword` type.

TokenizerMode (Ln. 483–489)

The `TokenizerMode` enum is very simple and to the point. The tokenizer has 4 core modes that it can be in as it steps through text:

[166] https://github.com/PowerShell/PowerShell/blob/9dfec6ca84d8bcdfdddc92c86f50dcc480c77d59/src/System.Management.Automation/engine/parser/token.cs

[167] https://github.com/PowerShell/PowerShell/blob/9dfec6ca84d8bcdfdddc92c86f50dcc480c77d59/src/System.Management.Automation/engine/parser/CharTraits.cs

[168] https://github.com/PowerShell/PowerShell/blob/9dfec6ca84d8bcdfdddc92c86f50dcc480c77d59/src/System.Management.Automation/engine/parser/Parser.cs

[169] https://github.com/PowerShell/PowerShell/blob/9dfec6ca84d8bcdfdddc92c86f50dcc480c77d59/src/System.Management.Automation/engine/parser/tokenizer.cs

1. `Command`
2. `Expression`
3. `TypeName`
4. `Signature`

The default mode is `Command` mode, where the tokenizer will be looking for command names, language keywords and the subsequent tokens that defines those language features or command parameters. However, if it recognizes other input types (for example, the `class` keyword, a variable or literal value, the opening `[` bracket of a type declaration), it can immediately switch out of `Command` mode. The rules by which it switches to each mode are reasonably straightforward, and each mode has several stages it will cycle through as it creates tokens. These are covered in more detail as you step through the methods themselves.

NumberSuffixFlags (Ln. 491–549)

The `NumberSuffixFlags` enum is used when the tokenizer is parsing numerical input. PowerShell actually recognizes *several* type suffixes when it reads in a number, which affect the final data type of the value.

A number in PowerShell may be suffixed with:

- Nothing - no suffix may indicate either a typical 32-bit integer or a floating-point double value (depending on whether decimal points are present).
- `u` — `Unsigned` — the `u` suffix indicates an unsigned integer (.NET has no unsigned floating-point types).
- `y` — `SignedByte` — the `y` suffix indicates an `sbyte` (signed byte) value.
- `uy` — `UnsignedByte` — the `uy` suffix indicates a regular (unsigned) `byte` value.
- `s` — `Short` — the `s` suffix indicates a `short` value, a 16-bit integer.
- `us` — `UnsignedShort` — the `us` suffix indicates a `ushort`, an unsigned 16-bit integer.
- `l` — `Long` — the `l` suffix indicates a `long` value, a 64-bit integer.
- `ul` — `UnsignedLong` — the `ul` suffix indicates a `ulong` value, an unsigned 64-bit integer.
- `d` — `Decimal` — the `d` suffix indicates a `decimal` value, a 128-bit high-precision floating point number.
- `n` — `BigInteger` — the `n` suffix indicates an arbitrarily large signed integer.

All suffixes are case-insensitive.

NumberFormat (Ln. 554–570)

Another enum utilized when parsing various forms of numbers, this enum is used to classify the format of a number. Numbers can come in 3 basic forms: decimal (base 10, no prefix), hexadecimal (base 16, `0x` prefix), or binary (base 2, `0b` prefix).

TokenizerState (Ln. 576–586)

This mini class is effectively in-memory storage for the state of the tokenizer. Finding nested scriptblocks within the script it has been asked to parse is pretty common, and in order to parse these, it will store its state using this class. Then, it proceeds to reset its own state and perform a whole new tokenizing operation on the scriptblock. Once that's completed it can then restore its prior state, skip to the end of the now-tokenized scriptblock, and resume where it left off in the main script.

The Tokenizer Class (Ln. 5–88)

The tokenizer itself takes up the remainder of the file, around 4500 lines.

Core Private Members (Ln. 591–613)

The following runs through some of the key private members and outline their job so you can retain your bearings as you go.

Static Members

```
private static readonly Dictionary<string, TokenKind> s_keywordTable // Ln. 591
private static readonly Dictionary<string, TokenKind> s_operatorTable // Ln. 593
```

These two dictionaries handle the mapping between text input and language keyword `TokenKind` values, and the similar mapping between operator strings and operator `TokenKind` values. These are actually populated during the tokenizer's constructor with all the values in these two arrays:

```
private static readonly string[] s_keywordText // Ln. 618
private static readonly TokenKind[] s_keywordTokenKind // Ln. 634

internal static readonly string[] _operatorText // Ln. 650
private static readonly TokenKind[] s_operatorTokenKind // Ln. 669
```

As you can see, each keyword and operator is stored initially in two arrays; one containing the text, and the other containing the corresponding `TokenKind` values. These have a 1:1 relationship, as the values at each position correspond to the same token type.

Instance Members

```
private string _script; // Ln. 611
private int _tokenStart; // Ln. 612
private int _currentIndex; // Ln. 613
```

As you might imagine, `_script` stores the full text of the script the tokenizer is currently examining, `_currentIndex` stores exactly what point it's at in the script, and `_tokenStart` is used to remember the starting position of the current token it's examining. These last two are extremely helpful when there could be a few possible `TokenKind` values as it scans a token; sometimes you do need to backtrack a little if the initial assumption didn't pan out.

For example, if you enter `300_spartans` into PowerShell as a bare string, at first it will see the number `3` and start scanning for more numbers. Then, once it reaches the `_` it will immediately realize something isn't right, stop scanning for numbers, and fall back to scanning what it calls a **generic** token. Generic tokens are most often parsed into command names or unquoted command arguments by the Parser.

Constructor (Ln. 690–720)

The tokenizer has a **static** constructor which does 2 main things:

1. Verifies that the keyword and operator tables mentioned above are both in sync, and the same size.
2. Generates a very basic hash to be used when checking for a script signature block.

The only instance constructor takes a single argument — a `Parser` instance, which is the instance of the parser that the tokenizer will provide with the generated tokens. This is the **only** way to create an instance of the tokenizer. In other words, you *must* have a Parser instance to pair with it — you can't instantiate the tokenizer on its own.

Auxiliary Private & Internal Members (Ln. 722–7–39)

Quickfire round! Most of these are relatively self-explanatory. For the ones you don't understand yet, don't worry; their meaning will become more apparent in context.

```
internal TokenizerMode Mode          // Stores the current tokenizer mode

internal bool AllowSignedNumbers     // Are signed numbers valid in the current co\
ntext?

internal bool WantSimpleName         // Restrict allowed characters in an identifi\
er

internal bool InWorkflowContext      // Are we in a PS Workflow context? (Deprecat\
ed)

internal List<Token> TokenList       // The list of tokens generated

internal Token FirstToken            // The first token in the script

internal Token LastToken             // The last token in the script
```

```
17
18 private List<Token> RequiresTokens  // The tokens in the script's #Requires state\
19 ment
```

There are also a few mode-checking methods which mainly serve to make code more expressive. For example:

```
1 private bool InCommandMode() { return Mode == TokenizerMode.Command; }
```

There's one method corresponding to each tokenizer mode.

Initialize Method (Ln. 741–7–76)

```
1 internal void Initialize(string fileName, string input, List<Token> tokenList)
```

This method is used to set the stage, make sure everything the tokenizer needs is in place. It does 2 main things:

- Ensures that any stored tokens are discarded, and the token list is set to the input list.
- Runs through the entire input very quickly to determine where each line starts.

Nested Scaning Methods (Ln. 778–8–00)

```
1 internal TokenizerState StartNestedScan(UnscannedSubExprToken nestedText);
2
3 internal void FinishNestedScan(TokenizerState ts);
```

I mentioned `TokenizerState` earlier; this is where it's used. The `StartNestedScan()` method creates takes a snapshot of the current tokenizer state, then proceeds to reconfigure the tokenizer to scan the nested script section.

Once this scan is completed, the `FinishNestedScan()` method will be called to restore the tokenizer to the stored state.

This pattern might seem a little baffling at first, but this method of approaching nested scanning is quite clever. It allows the tokenizer to scan nested scriptblocks to effectively any depth as needed, without ever needing to spin up a second tokenizer instance. One is enough. This helps immensely with speed and memory usage, and also very neatly sidesteps what could be a sticky question: how multiple tokenizer instances might interact with the one `Parser` instance.

Utility Methods

You'll frequently see `Debug.Assert()` method calls in a **lot** of utility and main tokenizer functions. These have **no effect** in a release build of PowerShell as the compiler can be configured to ignore them completely. However, during development these are built-in checks that will completely crash PowerShell if the check fails. This helps to ensure that critical functionality isn't broken.

In addition to the `InCommandMode()` method and its cousin, the tokenizer has a slew of helper methods that avoid an otherwise significant amount of duplicate code. These are also extremely helpful in terms of maintaining readability in most cases. This is a quick summary of their purposes, but a few rarely-used methods may be skipped to avoid overwhelming the reader.

Navigation and Scanning (Ln. 816–8–55)

These methods are used for navigation in the script input. It's stored as one huge block of text, and the tokenizer uses these methods to navigate within it.

```
1  private char GetChar(); // Ln. 816
```

`GetChar()` has one pretty simple job — increment the current index value (in other words, step forward one character) and then return that character. It also performs checks to ensure you're not trying to read outside the bounds of the input.

```
1  private void UngetChar(); // Ln. 831
```

Even simpler than its pair, `UngetChar()` simply decrements the current index value (in other words, it steps backwards one character).

```
1  private char PeekChar(); // Ln.838
```

`PeekChar()` checks the character at the current index without modifying the index value. It's a quick "what's it currently looking at" so that the tokenizer can *conditionally* step forwards, or check the same character against a few possible cases, or simply recognize something's wrong and either report an error or try another possibility.

PowerShell's flexibility as a language is partly made possible by these tiny helper methods and how they're used.

```
1  private void SkipChar(); // Ln. 850
```

You'll often find this paired with `PeekChar()`, and its role is extremely simple. It simply increments the current index value, without returning anything.

```
1  internal static bool IsKeyword(string str); // Ln. 862
```

The `IsKeyword()` method simply checks a given string to determine if it's a valid keyword. It will return `true` for both regular language keywords and dynamic keywords, and it simply searches the keyword and dynamickeyword tables for a match.

```
internal void SkipNewlines(bool skipSemis); // Ln. 877

private void SkipWhiteSpace(); // Ln. 955

private void ScanNewline(char c); // Ln. 969
```

These methods all handle skipping whitespace or generating newline tokens.

`SkipNewlines()` may seem a bit convoluted, but in a nutshell all it does is skip whitespace characters (including newlines) until it finds something more interesting to look at. It also skips comments, though it scans them and stores the tokens as it finds them. This process loops until either the script ends or another character is recognized.

`SkipWhiteSpace()` is a little more strict than `SkipNewLines()`, and *only* skips whitespace.

Finally, `ScanNewLine()` simply returns a token representing a newline character. It also normalizes CRLF (carriage return, line feed) sequences by stripping out the CR character, as PowerShell internally only typically uses LF to represent newlines.

```
private void ScanSemicolon(); // Ln. 981
```

There is a *specific method* to scan semicolons. The tokenizer needs to be as performant and lightweight as it can be, so this method verifies whether `TokenList` is available to store tokens in before opting to create the token.

```
private void ScanLineContinuation(char c); // Ln. 992
```

If you're not familiar with line continuations, in PowerShell a line continuation can be performed with a backtick character (` ) followed **immediately** by a newline. This method is almost identical to the above `ScanNewline()` method; the only difference you'll note is that this method is expecting a two-character newline: the backtick, **and** the line feed character.

```
internal void Resync(Token token); // Ln. 1010

internal void Resync(int start); // Ln. 1018
```

The `Resync()` method is called by the Parser when it determines that it needs to backtrack. This can be a fairly expensive operation for the tokenizer, so uses are kept to a minimum. When this method is called, the tokenizer discards any tokens generated after the token given to this method, and then resumes scanning from this point again.

```
internal void CheckAstIsBeforeSignature(Ast ast); // Ln. 1112
```

In PowerShell, signature blocks must be at the bottom of a file. If this isn't the case, this method reports a parse-time error and stops tokenizing the file.

Error Reporting (Ln. 1125–1153)

```
private void ReportError(IScriptExtent extent, string errorId, string errorMsg);

private void ReportIncompleteInput(int errorOffset, string errorId, string errorM\
sg);
```

The `ReportError()` and `ReportIncompleteInput()` methods, which have a few overloads, call back to the similarly-named methods on the Parser instance. This will cause a parsing error to be created, and parsing of the script or input will halt.

Creating Extents (Ln. 1155–1191)

All AST objects created by the parser have an `Extent` property which details both position in the overall script, and the literal text that was turned into the token.

These methods are necessary in order to generate this information, which can be later used in the `Resync()` method, for example. Extents are also used in reporting parse errors, which helps a great deal with locating the problem!

Generating Tokens (Ln. 1193–1309)

There are a wide variety of tokens used in PowerShell. The methods here are all very similar. Each method calls the private `SaveToken()` method after creating the token that each method is named after. Depending on the kind of token, the method and token creation may require different input values. Here are a few of the more commonly-used token-generating methods:

```
private Token NewNumberToken(object value);

private Token NewParameterToken(string name, bool sawColon);

private VariableToken NewVariableToken(VariablePath path, bool splatted);

private StringToken NewStringLiteralToken(
    string value,
    TokenKind tokenKind,
    TokenFlags flags);

private StringToken NewStringExpandableToken(
    string value,
    string formatString,
    TokenKind tokenKind,
    List<Token> nestedTokens,
    TokenFlags flags);

private Token NewGenericToken(string value);
```

Special Line Continuations (Ln. 1345–1416)

There are a few recent additions to the tokenizer that allow for additional line continuations, which will be included for the first time in PowerShell 7.0.

```
internal bool IsPipeContinuance(IScriptExtent extent);

internal bool IsPipeContinuance(IScriptExtent extent);
```

These methods allow the tokenizer to read pipelines written like this, which wasn't valid syntax in PowerShell 6.x and below:

```
PS> Get-ChildItem -Path $HOME -Directory
>> | ForEach-Object -MemberName Name
>> | Get-Random -Count 5
>> | Sort-Object -Descending
```

At the time of writing, there are several other proposals for additional line-continuation behaviours that may end up being incorporated into the tokenizer in a later PowerShell version.

Skipping Comments (Ln. 1418–1446)

```
private int SkipLineComment(int i);

private int SkipBlockComment(int i);
```

These methods simply step along through the comments in a script, looking for the character(s) that indicate their end. Once they find it, they return the index just after the end of the comment.

Special Characters (Ln. 1448–1577)

```
private char Backtick(char c, out char surrogateCharacter);

private char ScanUnicodeEscape(out char surrogateCharacter);

private static char GetCharsFromUtf32(uint codepoint, out char lowSurrogate);
```

These methods are all parsing backtick-escaped characters and giving back the appropriate character. For example, `"``n"` is transformed into a literal newline, and `"``u{0x25}"` becomes a `%` sign.

`Backtick()` specifically handles all the standard characters like a tab stop, newline, etc., and then the ‘u case handles any and all unicode characters by calling out to the `ScanUnicodeEscape()` method, which turns the input value into the appropriate Unicode character.

Signed Scripts and `#Requires` (Ln. 1579–1714)

```
private void ScanToEndOfCommentLine(out bool sawBeginSig, out bool matchedRequire\
s)
```

This method is *mainly* just for reading to the end of a single-line comment. However, both `#Requires` and a signature block begin the same way as a single-line comment, so this method has a fair bit of additional logic in order to be able to determine whether it's something important or just a comment.

Reusable StringBuilders (Ln. 1718–1745)

The tokenizer maintains its own private `Queue` of `StringBuilder` objects, as you can see here.

```
private readonly Queue<StringBuilder> _stringBuilders = new Queue<StringBuilder>(\
);

private StringBuilder GetStringBuilder();

private void Release(StringBuilder sb);

private string GetStringAndRelease(StringBuilder sb);
```

This allows the tokenizer to avoid a lot of unnecessary allocations. Many of its parsing methods require the use of `StringBuilders` in order to efficiently create tokens. This setup creates a framework where a `StringBuilder` is only created if it's needed; `StringBuilders` can be cleared out and reused instead of expending resources instantiating a new one.

Scanning Comments (Ln. 1747–2244)

```
private void ScanLineComment();

private void ScanBlockComment();
```

What's that, you thought you were done with comments? There is still a bit more here. While some simple *recognition* of `#Requires` and signature blocks have already been covered, the methods already seen don't cover actually parsing a `#Requires` statement, much less evaluating the validity of a signature block.

```
internal ScriptRequirements GetScriptRequirements();

private void HandleRequiresParameter(CommandParameterAst parameter,
    ReadOnlyCollection<CommandElementAst> commandElements,
    bool snapinSpecified,
    ref int index,
    ref string snapinName,
    ref Version snapinVersion,
    ref string requiredShellId,
    ref Version requiredVersion,
    ref List<string> requiredEditions,
    ref List<ModuleSpecification> requiredModules,
    ref List<string> requiredAssemblies,
    ref bool requiresElevation);

private List<string> HandleRequiresAssemblyArgument(
```

```
    Ast argumentAst,
    object arg,
    List<string> requiredAssemblies);

private List<string> HandleRequiresPSEditionArgument(
    Ast argumentAst,
    object arg,
    ref List<string> requiredEditions)
```

The code contained in these methods is pretty dry, but does about what you'd expect. It uses the methods you've already seen to determine if a comment is a `#Requires` or a signature, and then it evaluates them appropriately. If you're particularly interested, looking at the `#Requires` code is recommended, though you will end up scrolling by a lot of `ReportError()` calls for all the failure cases.

Tokenizing Strings (Ln. 2246–2834)

PowerShell has a few neat tricks up its sleeve when it comes to strings, but by and large the biggest difference you'll see will come down to whether it starts with single or double quotes.

Strings in Command Arguments

Sometimes, however, you can have a string value that doesn't have quotes. These are only available when entering command arguments. When you enter a command and then pass an argument to it that's a simple string, that string will remain a string token, instead of a generic token (generic tokens are what commands start as).

```
# Despite not enclosing this in quotes, it remains a string value
PS> Get-ChildItem -Path .\pwsh
```

Here is the method which is called upon to handle this:

```
internal StringToken GetVerbatimCommandArgument(); // Ln. 2250
```

This is pretty straightforward for a tokenizer method. It checks for disallowed characters, stops on a space (spaces aren't included in an unquoted command argument string), and has a bit of extra handling for the stranger edge cases where you want to put quotation marks in the middle of the unquoted argument.

Single-Quoted Strings

As for single-quoted strings, those of you familiar with PowerShell are probably aware, but a `'single-quoted string'` is assumed to be **literal**. The engine won't bother checking it for variables or subexpressions to expand.

```
private TokenFlags ScanStringLiteral(StringBuilder sb); // Ln. 2284

private Token ScanStringLiteral(); // Ln.2322
```

These are pretty straightforward, simply stepping through and adding every character it sees to the StringBuilder that's being used to create the final string value. The only additional handling here is checking for an escaped quote in the middle of the string. As a quick example:

```
PS> 'this string contains ''escaped quotes,'' you see?'
this string contains 'escaped quotes,' you see?
```

Double-Quoted Strings

Double-quoted strings have a lot more going on: expanding variables, entire subexpressions, etc. In the interest of keeping it relatively easy to digest, this chapter doesn't get too in-depth with the actual code itself.

```
private Token ScanSubExpression(bool hereString); // Ln. 2329

private TokenFlags ScanStringExpandable( // Ln. 2516
    StringBuilder sb,
    StringBuilder formatSb,
    List<Token> nestedTokens);

private bool ScanDollarInStringExpandable( // Ln. 2485
    StringBuilder sb,
    StringBuilder formatSb,
    bool hereString,
    List<Token> nestedTokens);

private Token ScanStringExpandable(); // Ln. 2535
```

Essentially what's going on here is:

1. The tokenizer recognizes an expandable string (starts with `"`)
2. `ScanStringExpandable()` is called.
3. `TokenFlags` are determined, and any subexpressions are examined, and sub-tokens are retrieved.
4. An expandable string token is generated, with all this information.

Once the token is created, the Parser and Compiler can use it to evaluate the final string content.

Here-Strings

Here-strings in PowerShell look like this:

```
$Literal = @'
String contents
'@

$Expandable = @"
String $contents
"@
```

In terms of the tokenizer, it now simply means that it changes what it looks for. The tokenizer here behaves similarly to scanning through other strings, except that it's now searching for a valid ending sequence for the string, and no longer caring about quotation marks that would otherwise end a normal string (unless they comprise part of the ending sequence).

```
private bool ScanAfterHereStringHeader(string header);

private bool ScanPossibleHereStringFooter(
    Func<char, bool> test, Action<char> appendChar,
    ref int falseFooterOffset);

private Token ScanHereStringLiteral();

private Token ScanHereStringExpandable();
```

As you can see, there are specific methods to scan for the header or footer of the here-string. In addition, here-strings have their own version of the `ScanStringLiteral` / `ScanStringExpandable` which mainly only differs in the handling of quotation marks as mentioned a moment ago.

Tokenizing Variables (Ln. 2836–3149)

Variables in PowerShell are very powerful, but tokenizing them correctly is no easy feat. The intricate details here are well worth a look (linked above!) but in terms of understanding the primary method used here, efforts were made to keep it simplified for you.

```
// Scan a variable - the first character ($ or @) has been consumed already.
private Token ScanVariable(bool splatted, bool inStringExpandable);
```

As noted by the comment here, by the time it gets to this method, the tokenizer's already consumed the `$` or `@` character. As a result, this information is passed by the method parameters instead.

This method checks for **braced variables** as well, as braced variables are permitted to use just about any possible character in their names. Unbraced variables are limited to basic alphanumeric names, with only a few allowed symbols.

```
${my braced\\variable} = "ted"
$unbraced_variable = "not ted"
```

You'll notice that, like many of the tokenizer's `Scan` methods, this method actually uses a `while (true)` loop — a deliberately infinite loop. As ill-advised as this can be, arguably it's one of the only straightforward ways to get this done.

What often makes these methods a lot more difficult to follow, however, is their use of the `goto` keyword to break out of the loop and move on once the initial scanning is completed. Some of the methods use `goto` in order to re-start their scans when necessary as well. It can sometimes be more clear if you follow it one character at a time; trying to figure out how it'll operate across even one whole word can be a little tricky.

The long and the short of it for this method is:

1. It checks if the variable is braced or not.
2. It loops through all the characters, until it reaches the end (or an invalid character).
3. It checks for errors, and report any found.
4. If no errors are found, it creates our variable token and return it.

Note that this is the tokenizer — it's just reading the input text. The tokenizer hasn't the faintest idea what's **in** the variable, if anything at all. It just knows there's a variable token here.

It's also worth noting that PowerShell has some very cleverly reserved variables (namely, `$$` and `$^`) which would normally be considered actually invalid as variable names. These variables are hard-coded into the tokenizer as an additional case, ignoring the standard variable name rules.

Generic Tokens (Ln. 3330–3446)

```
private Token ScanGenericToken(char firstChar); // Ln. 3330

private Token ScanGenericToken(char firstChar, char surrogateCharacter); // Ln. 3\
337

private Token ScanGenericToken(StringBuilder sb); // Ln. 3349
```

These methods are the tokenizer's "catch-all" bucket. If it looked like it was going to be some other token, and then an unexpected character was found, it will end up being passed to these methods.

It's actually rather difficult to summarize it better than the original code comments in the method here convey it:

> On entry, it's already scanned an unknown number of characters and found a character that didn't end the token, but made the token something other than what it thought it was.
>
> Examples:

- `77z` — it looks like a number, but the 'z' makes it an argument.
- `$+` — it looks like a variable, but the '+' makes it an argument.

A generic token is typically either command name or command argument (though a generic token is accepted in other places, such as a hash key or function name.) A generic token can be taken literally or treated as an expandable string, depending on the context. A command argument would treat a generic token as an expandable string whereas a command name would not. It optimizes for the command argument case - if it finds anything expandable, it continues processing assuming the string is expandable, so it'll tokenize sub-expressions and variable names. This would be considered extra work if the token was a command name, so it assumes that will happen rarely, and indeed, '$' isn't commonly used in command names.

Tokenizing Numbers (Ln. 3448–4118)

The tokenizer has a few different methods for parsing numbers. First of all, it has the basic scanning methods:

```
1  private void ScanHexDigits(StringBuilder sb); // Ln. 3450
2
3  private int ScanDecimalDigits(StringBuilder sb); // Ln. 3461
4
5  private void ScanBinaryDigits(StringBuilder sb); // Ln. 3476
6
7  private void ScanExponent( // Ln. 3487
8      StringBuilder sb,
9      ref int signIndex,
10     ref bool notNumber);
11
12 private void ScanNumberAfterDot( // Ln. 3506
13     StringBuilder sb,
14     ref int signIndex,
15     ref bool notNumber);
```

These methods handle the string input directly, making sure that the subsequent methods only receive useful data.

`ScanHexDigits()`, `ScanDecimalDigits()`, and `ScanBinaryDigits()` all operate similarly, looping over the input and adding valid numbers to the string. Numbers can also have suffixes in PowerShell, so they don't immediately assume that an unwanted character means an error.

`ScanExponent()` is called when a number like `2e6` is passed in; `ScanDecimal()` handles it normally until the `e` is hit, then returns. Then, `ScanExponent()` is called to finish up and get the exponent value on the end. These numbers will shortly be evaluated, and `2e6` becomes `2,000,000` before the final data type is decided. Note that for exponents, it's normal and acceptable to have a negative sign in the exponent, so `2e-6` is also a valid number token for PowerShell's tokenizer to accept.

`ScanNumberAfterDot()` does exactly what it says on the tin; when you give the tokenizer a number with a decimal place, `ScanDecimal()` would stop at that point. Then, `ScanNumberAfterDot()` will be called to finish the scan.

```
private static bool TryGetNumberValue(
    ReadOnlySpan<char> strNum,
    NumberFormat format,
    NumberSuffixFlags suffix,
    bool real,
    long multiplier,
    out object result);
```

This method is called after the scans are done, and the type is decided. It bears the brunt of the logic to determine the actual type of the resulting value.

If the `d` decimal suffix is provided, it can proceed to parse it as decimal and return early. Otherwise, it will either attempt to parse it as a `double` value or a `BigInteger` depending on whether or not the number is considered **real**. All numbers input with a decimal point or an exponent are automatically considered real numbers.

Before checking types, any multiplier suffixes (for example, `10gb`, `5mb`) are applied first to ensure that the target type can actually hold the requested value. Then, any provided suffix is checked at this point to determine if the value can be contained in the specified type.

If no suffixes are provided, the default type is `double` for real values, and `int` or `long` for non-real values, depending on whether the value is small enough to be stored in an `int` or not.

If all goes well, the number can be returned and stored in the resulting number token. Below is a brief example of the process of determining number value in the tokenizer with a few example cases.

```
10d                              -> decimal
11.4d                            -> decimal
12e4    -> real                  -> double
12      -> non-real              -> BigInteger -> int
10gb    -> non-real -> 10 * 1GB  -> BigInteger -> long
12.0    -> real                  -> double
1.5sKb -> real     -> 1.5 * 1KB -> double     -> short
```

The methods that does a lot of the actual decision making to get to that point, however, are `ScanNumber()` and `ScanNumberHelper()`.

```
private Token ScanNumber(char firstChar); // Ln. 3838

private ReadOnlySpan<char> ScanNumberHelper(  // Ln. 3886
    char firstChar,
    out NumberFormat format,
    out NumberSuffixFlags suffix,
    out bool real,
    out long multiplier);
```

`ScanNumberHelper()` is responsible for:

- Determining the format of the number (hex, binary, decimal).
- Handling the number strings from the input.
- Identifying decimal points, exponents, and suffixes.
- Determining the type and multiplier suffix values.

At this point, the information is fed out to `ScanNumber()` and then back into `TryGetNumberValue()` to determine the final values.

Member Access / Invocations / Necessary Characters (Ln. 4120–4216)

```
internal Token GetMemberAccessOperator(bool allowLBracket); // Ln. 4120

internal Token GetInvokeMemberOpenParen(); // Ln. 4197
```

These two methods handle syntax like this:

```
PS> $variable.Property
PS> $type::StaticMethod()
PS> $variable.Method()
```

They actually only handle the `.`, `::`, and `(` items in the above examples, all of which are key. When calling a member of an instance or a type, PowerShell needs to know whether the access method is static or not, and it also needs to know if you're actually calling the method or requesting information about it.

A method in PowerShell called without its parentheses will actually list out the metadata for the method, including the required parameters to invoke it. Additionally, the distinction between access with `.` and `::` is extremely important.

All these methods need to do is check that the context they're being queried in is valid, and then examine the separator characters in order to generate the correct tokens for the Parser to utilize later on.

The actual member *names* themselves are separately scanned as generic tokens.

Identifiers (Ln. 4335–4391)

The tokenizer uses **Identifiers** for a few different purposes:

- Keywords
- Command Names
- Command Arguments
- Method / Property Names

The basic rule here is that identifiers are pretty much universally alphanumeric, with only a scant few symbols permitted.

```
1  private Token ScanIdentifier(char firstChar);
```

Depending on whether the tokenizer is currently in command or expression mode, the generated token here can be a generic token instead of a true `Identifier`, but this method will handle both.

Type Names (Ln. 4393–4505)

```
1  private Token ScanTypeName(); // Ln. 4395
2
3  private void ScanAssemblyNameSpecToken(StringBuilder sb); // Ln. 4428
4
5  internal string GetAssemblyNameSpec(); // Ln. 4452
```

These methods are called after you've already got an open L-bracket, and are expecting a type name of some form. The permitted symbols are much more restricted, as the allowable characters in a type name in the .NET CLR is pretty restricted.

The only allowable characters are letters, digits, and a few special characters.

Label Names (Ln. 4507–4543)

Labels are rarely used in PowerShell, and only valid in the context of labelling a specific loop in a script. They follow the same rules as any other identifier / generic token in PowerShell in the tokenizer, except that they start with a `:` character. For example:

```
1  :loop1 foreach ($a in 1..10) { if ($_ -in 3, 6, 7) { break } $_ }
```

The tokenizer doesn't retain sufficient context awareness to determine if a label is actually valid in context, however — for example, the loop that's meant to come after it could be missing.

NextToken (Ln. 4545–4992)

This is the primary method that the parser will interact with the tokenizer. It will simply call `_tokenizer.NextToken()` and then see what it gets, and that's fantastic when you consider the relative complexity of this method.

```
internal Token NextToken();
```

It's essentially a giant `switch` statement, with a *lot* of bells and whistles, plus a sprinkling of `goto` statements here and there. These are mainly here so that the tokenizer can recognize there's whitespace, handle it, and then continue scanning until it gets the next token that the Parser actually wants to look at. It saves an additional series of checks and messy code in the Parser itself by simply tucking that away here at this point.

The long and the short of it: this method determines what the tokenizer is looking for based on its current position in the file, and what it looks for from there. If you were to try adding a completely new style of syntax to PowerShell, that's certainly one place you might need to look at adding a few extra special cases for.

Part IV: Handling Data

This section highlights aggregating, visualizing, and acting on incoming data, whether it be logs, events, or environmental information.

Keeping Your Users in the Loop with Toast Notifications

by Josh King

I love toast notifications. They're a great way of getting information out to people without necessarily demanding their attention in the moment. As you dive into them further, you'll find that their potential use cases and customizability is limitless.

Need to tell your users to log out at the end of the day? Use a toast!

Need to remind users that there's an all staff meeting today at 10, and the CEO will be personally upset if they aren't there? Use a toast!

Waiting on a script and want to know when it's done without keeping the console open? Use a toast!

My hope is that by the end of this chapter, you'll love—and use—toast notifications as much as I do.

Toasts

Toast? But this isn't breakfast!

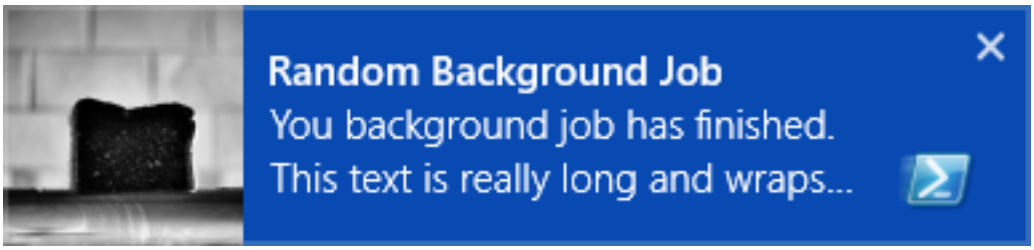
You may not know them by the name "toast," but it's a guarantee that you've seen them. Chances are you've heard of toast notifications referred to as "pop-up notifications" or "notification bubbles," but for the remainder of this chapter I'll be referring to them simply as "toasts."

These toasts are small graphical elements that pop on your screen to communicate some information to you. One of their defining features is that they're meant to automatically disappear after a certain amount of time. After disappearing they should then be retrievable from a common location.

All major current Operating Systems support toasts, even those running on your phone. This chapter will only be focusing on their implementation in Microsoft Windows, though you will be left with some breadcrumbs to follow in the final section for Mac and Linux notifications.

You might find yourself asking, "focusing on Windows is fair enough, but why just Windows 10?"

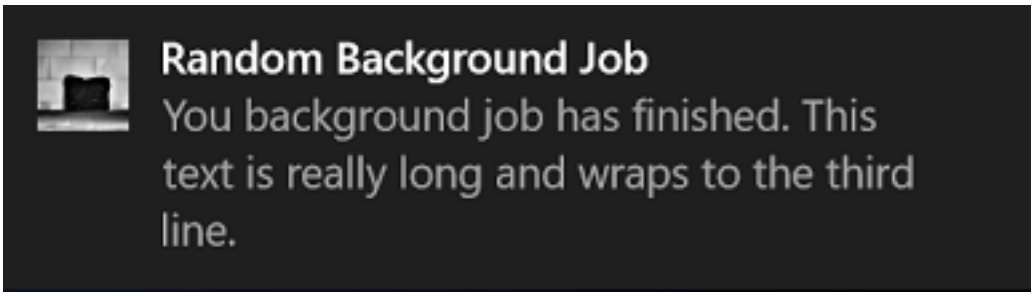
Toasts as a concept were added to Windows in Windows 8. They were the notifications that would pop up towards the top right of your screen every time you inserted some sort of removable media.



An example toast from Windows 8.1

These toasts were heavily template based, meaning you'd have to select the best option for your use case from a catalogue.

Windows 10 shook things up by adding adaptive toasts and moving the notifications to the lower right hand corner of your screen. The term adaptive toast sounds a lot more intimidating than it really is. They're simply a more free form method for creating toasts, using a generic template, giving you much more creative freedom.



An example toast from Windows 10

As a matter of opinion, the Windows 8 style toasts look much nicer than the new variety. The new flexibly more than makes up for this, but that's enough history. Time to dive into your first toast notification!

Cooking Your First Toast

Your first toast notification is going to be a whirlwind tour of the basics. It's going to look a little complicated straight off the blocks, but don't be put off. The next section in this chapter introduces you to a module that helps with the parts of this that seem arcane.

Before that, hurry up and get your first toast cooking.

The first thing to know is that behind the scenes, a toast notification is XML. When working directly with the XML, it's preferable to store them in a here-string[170] so that they're easy to read and understand.

[170] https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_quoting_rules?view=powershell-5.1#here-strings

```
$XmlString = @"
<toast>
  <visual>
    <binding template="ToastGeneric">
      <text>Default Notification</text>
    </binding>
  </visual>
  <audio src="ms-winsoundevent:Notification.Default" />
</toast>
"@
```

There's a deceptively large amount of information to unpack about this XML schema. If you're interested in getting a deeper understanding than what this chapter can give you, there's some good information available from Microsoft Docs[171].

The main thing to know about this example is the template, "**ToastGeneric**." This is where you would have listed the specific template you wanted to use in Windows 8, but now you should just leave it as the generic option. In fact, you have to leave this as the generic option if you want to make use of any of the new adaptive features from Windows 10.

Next is the text wrapped in `<text>` tags. You can have more than one of these, and the first one will always be the bold heading of your toast. It's beyond the scope of this chapter, but this is also where you can include an image.

Finally, is the audio that plays when the toast is displayed. This example used the default sound, but there is a small library[172] of other operating system provided sounds you can use in its place.

Next you need to pick an "**AppUserModelID**", referred to as AppID from this point forward. This is the string that identifies applications installed on Windows, and it's use with toasts tells Windows which application generated a given notification. You can get a list of all valid AppIDs by running `Get-StartApps` and you're free to pick any of them.

For this example, you will use Windows PowerShell's AppID.

```
$AppId =
 '{1AC14E77-02E7-4E5D-B744-2EB1AE5198B7}\WindowsPowerShell\v1.0\powershell.exe'
```

Next you need to load some Windows runtimes into your PowerShell session. There's not much that can be written about this sample code without diving into insane detail. Just know that this has to be run into to display your toast, and it allows you to work with some built-in .NET classes.

[171] https://docs.microsoft.com/en-us/windows/uwp/design/shell/tiles-and-notifications/toast-xml-schema
[172] https://docs.microsoft.com/en-nz/uwp/schemas/tiles/toastschema/element-audio

```
$null = @(
    'Windows.UI.Notifications.ToastNotificationManager'
    'Windows.UI.Notifications'
    'ContentType = WindowsRuntime'
) -join ',' -as [type]

$null = @(
    'Windows.Data.Xml.Dom.XmlDocument'
    'Windows.Data.Xml.Dom.XmlDocument'
    'ContentType = WindowsRuntime'
) -join ',' -as [type]
```

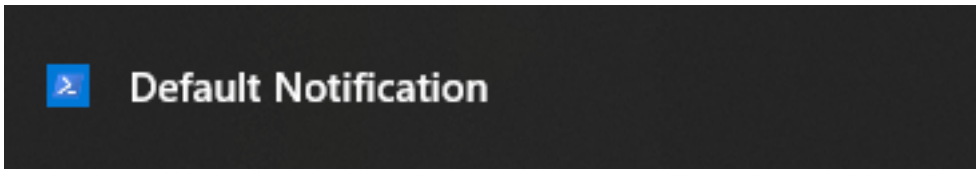
The second to last step is to load the here-string into a "real" XML object and then use that to create a new "ToastNotification" object.

```
$ToastXml = [Windows.Data.Xml.Dom.XmlDocument]::new()
$ToastXml.LoadXml($XmlString)
$Toast = [Windows.UI.Notifications.ToastNotification]::new($ToastXml)
```

Finally, pass your toast to the "Toast Notification Manager." You need to provide the AppID that you defined earlier and the toast object itself.

```
$NotificationManager = [Windows.UI.Notifications.ToastNotificationManager]
$NotificationManager::CreateToastNotifier($AppId).Show($Toast)
```

All going well, you'll see your toast pop up above everything else on your screen. It's not much to look at at this stage, but take pride in your very first toast notification!



Your first toast notification

Troubleshooting Your First Toast

Unfortunately, toasts don't always go as planned on your first go. Often because a setting has been toggled on your system to limit the display of notification from other application.

Firstly, double check that Focus Assist[173] isn't enabled. This is a great feature to ensure you're not being distracted when trying to get work done, however it will also suppress the notifications you're working on.

Next, open up the **Settings** app and head to "**System**" then "**Notifications & actions**." Scroll down and you'll see a list of apps that have, or can, send toasts. Look for the application you

[173] https://blogs.windows.com/windowsexperience/2018/05/09/windows-10-tip-how-to-enable-focus-assist-in-the-windows-10-april-2018-update/

chose for your AppID and ensure it's toggled on. You can also click on the application to change some more granular settings. In this area you could decide that you do want to see toasts from a specific app, but don't want it to be able to make noises. You can also customize how many past notifications are visible in the Action Center without having to click the "see more" button.

Introduction to BurntToast

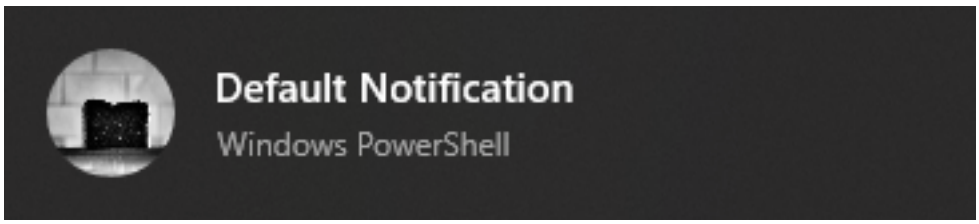
Now that you've seen how to do toast the hard way, it's time to leverage an existing module to make the whole process easier. The first thing you'll need to do is download the BurntToast[174] module from the PowerShell Gallery.

```
1 Install-Module -Name BurntToast -Scope CurrentUser
```

This command installs the module to your user directory. You can omit the scope parameter to install the module system wide, but you will need to be running the PowerShell console as administrator.

With the BurntToast module installed, you can very quickly display a new notification with a single command.

```
1 New-BurntToastNotification
```



Default BurntToast notification

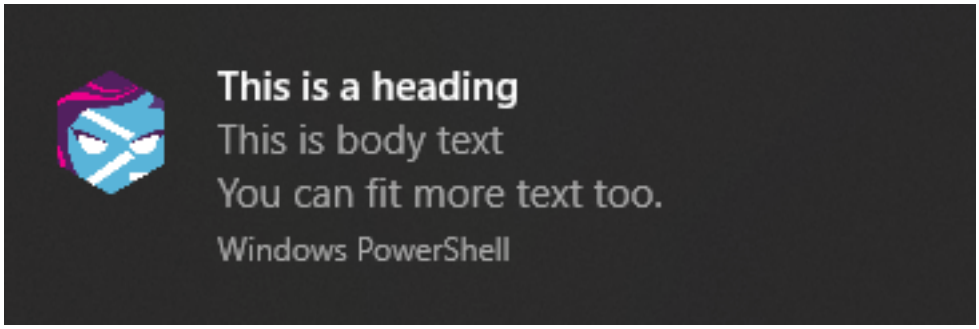
Now it's time to iterate quickly so that you can finally make a toast notification that's 100% your own. First, decide what text you want displayed on your toast and provide that against the `-Text` parameter. Remember that the first string you provide will be the heading and everything thing else will be displayed underneath that.

You'll also notice that the default BurntToast notification had a larger picture instead of the PowerShell icon that was on your first notification. This is known as an "App Logo Override" and it can be any image you desire. To change the logo, provide the path to the desired image to the `-AppLogo` parameter.

[174] https://www.powershellgallery.com/packages/BurntToast

```
$ToastSplat = @{
    AppLogo = 'C:\Toast\pwsh-hex.png'
    Text = 'This is a heading', 'This is body text', 'You can fit more too.'
}

New-BurntToastNotification @ToastSplat
```



Customized BurntToast notification

Note that this example included three different string objects, each of which appear on their own line. You can perform any of the string manipulation or variable substitution you'd normally do in PowerShell to make these toasts more dynamic.

You're now at the point where you can display toast notifications with whatever text you require. Now it's time to start diving into a few of the more advanced customization options you have at your disposal.

Listening to Your Toast Sing

So far you've only customized the appearance of your toasts. Sometimes the best way to get a user's attention, if that's the aim of your notification, is to change the sound that accompanies it.

In the "Cooking Your First Toast" section, you saw that there was a small library of built-in sounds you can use. To access them, use the `-Sound` parameter. This parameter gives you tab completion of all the available sounds, so choose from the list and you're good to go.

If you really do want your user's attention, it's best to pick one of the "Call" or "Alarm" sounds. Choosing any of these will loop the sound while the toast is onscreen and all other sounds, such as videos, will be turned down.

```
New-BurntToastNotification -Sound Alarm10
```

As a bonus, using any of these looping sounds will also result in the toast staying onscreen longer than a standard toast. Use this power sparingly, it can be incredibly annoying if overused and the last thing you want is your users getting annoyed enough to learn how to suppress your notifications.

On the other end of the annoyance spectrum, if you don't want your toast to make any noise you can use the `-Silent` switch.

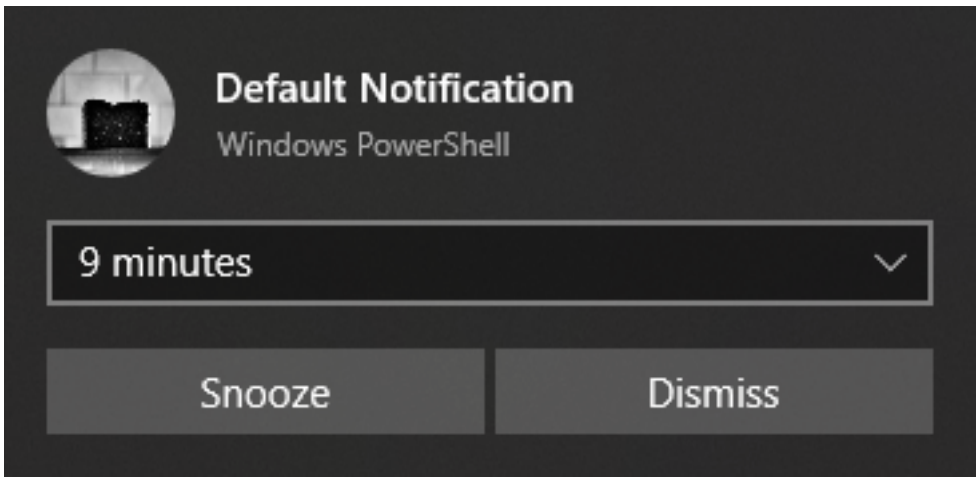
```
New-BurntToastNotification -Silent
```

Taking Action, or Just Snooze and Dismiss

Now that you know how to make your toast display the information you need it to, and grab your user's attention with different sounds, it's time to look at enabling your toast to *do* more.

The first example in this regard is enabling "Snooze and Dismiss." This gives the user a series of buttons and a drop down list for dismissing the notification or snoozing it for a selectable number of minutes.

```
New-BurntToastNotification -SnoozeAndDismiss
```



Example of a Snooze and Dismiss toast

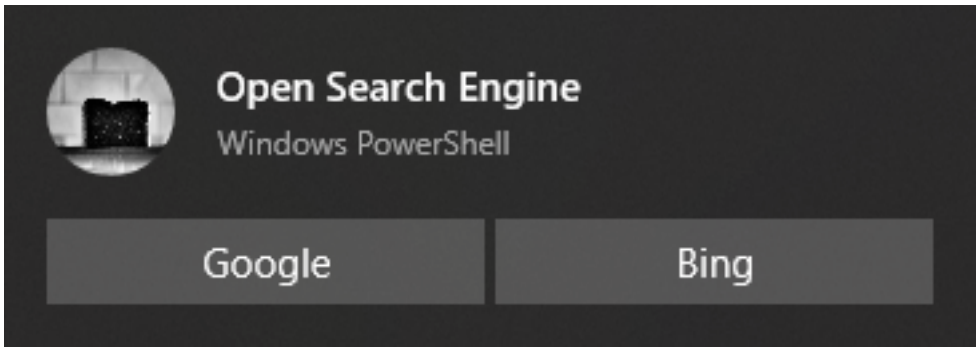
If the user decides to snooze the toast, it will reappear after the amount of minutes shown in the dropdown list.

What if you want to create your own buttons?

You can create "Button" objects with the `New-BTButton` function, and provide one or more of them to the `-Button` parameter.

```
$Button1 = New-BTButton -Content 'Google' -Arguments 'https://www.google.com'
$Button2 = New-BTButton -Content 'Bing' -Arguments 'https://www.bing.com'

New-BurntToastNotification -Text 'Open Search Engine' -Button $Button1,$Button2
```



Example of toast with buttons

When creating your button objects, the "Content" is the text that displays on the button itself. The "Arguments" is the link to be opened when the button is clicked. If this is a URL, it will open in your default browser, but it doesn't necessarily need to be a website.

You can instead provide a path to a file. Doing this will open the file in the default program for the file type. For example a *.docx* will open in Microsoft Word and a *.jpeg* may open in the Photos app.

But Why? Exploring Scenarios and Use Cases for Toasts

You've learned a lot of the "how" regarding toasts, and with any luck your mind is racing with all the places you'll be able to use them. To tie a pretty bow on your introduction to these notifications, it's time to spend a little bit of time going through some of the "why."

One of the main uses for toasts in PowerShell scripts is to let you know when a long running process is finished. Maybe you have a report that takes 20 minutes or longer to generate and you'd really like to know when it's done. You don't, however, want to keep your console open the whole time.

At the end of your report generation script, add the code for a toast notification. You may even want to add a button that points to the *.pdf* or *.csv* file that was the end result of the script so that you can open it directly.

If you've got a job running in a background job, you could setup an object event[175] to fire a toast when the job completes.

Finally, instead of sending out password expiry warning emails to users, you could use toasts. As the expiry time gets closer, you could make your toasts more ~~annoying~~ attention grabbing by using an alarm sound.

But Don't Get Too Excited, There's a Limitation

After all this, you probably have one burning question: "how do you trigger a PowerShell script on a button press?"

[175] https://github.com/Windos/BurntToast/blob/master/Examples/Example06/Get-ToastJobNotification.ps1

The unfortunate answer is that at the time of writing, you can't.

This is a topic with a lot of depth to it and a deep dive that's out of scope for this chapter. The very quick explanation is that toasts are part of the "WinRT" APIs in Windows which PowerShell doesn't completely support. Toasts do fire off events, but PowerShell can't register actions against them. This means there is a lot of untapped potential still waiting to be unlocked.

Go Forth and Cook Toast

There's more to learn about toast notification than what could be included in this chapter without making it a book in and of itself.

Do consider rummaging around the BurntToast notification to find out what else is possible.

Some future topics for you to dive into include:

- Progress bars
- Clickable toast
- Images (inline and hero)
- Custom sounds
- Exporting toast XML for portability

It's time for you to explore toasts at your own leisure and find your own use cases. Have fun!

ReportCardPS - Create Custom HTML Reports with VMware's Clarity UI Styling

by Justin P. Sider

This chapter focuses on how to use the ReportCardPS PowerShell module to create customized HTML reports that leverage VMware's Clarity UI project. It covers the structure of creating a report manifest and custom functions used by the report manifest file.

PowerShell has been used to create HTML reports for years. System administrators have written hundreds and hundreds of lines of custom code that create visually pleasing reports in an HTML format. Many of those lines exist solely to format data in a pleasing HTML format. Utilizing an external library for HTML styling allows for a much simpler approach to creating these customized reports.

This chapter assumes you are already at a moderate skill level with PowerShell and have a basic knowledge of HTML elements, PowerShell function structure, and JSON objects. To follow along with the commands in this chapter, Windows PowerShell 5.1 is required. No other versions of PowerShell were tested with the code referenced in this chapter. The ReportCardPS PowerShell module version used for this chapter is `0.0.14`.

How ReportCardPS Works

The Basics

The purpose of the ReportCardPS module is to turn output from multiple PowerShell functions into a consumable HTML document. Each function should be represented by an area on the page that's distinct. The HTML document should be able to support multiple screen sizes and a random number of functions or cards.

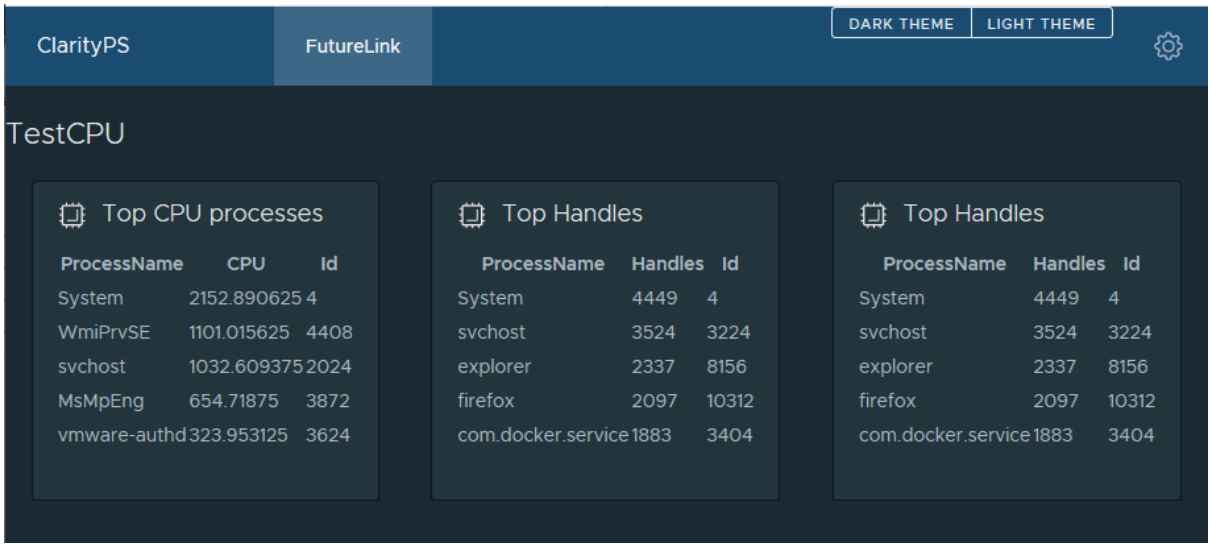
The `New-ReportCard` function iterates through the PowerShell functions provided in the report manifest file. First, a new HTML document is created. Second, each function is executed and the output of the function is transformed into HTML elements. As each function is executed the HTML elements are added to the HTML document. Finally, after all functions have been executed, the HTML document is returned to the PowerShell pipeline. This function also inserts all the required HTML header information to use the VMware Clarity HTML and Cascading Style Sheets (CSS) elements.

VMware's Project Clarity

The HTML/CSS framework that supports ReportCardPS is VMware's Clarity Design System[176]. Clarity is an active open source project and is licensed under the MIT License. This design system provides rich features and an exceptional experience to end users. ReportCardPS consumes the HTML elements of Project Clarity through ClarityPS[177], a PowerShell module specifically aimed at exposing all Clarity HTML elements. This allows for the ReportCardPS module to focus solely on providing the functionality to generate reports. ReportCardPS also takes advantage of having built-in support for Light and Dark themes.

Understanding HTML Cards

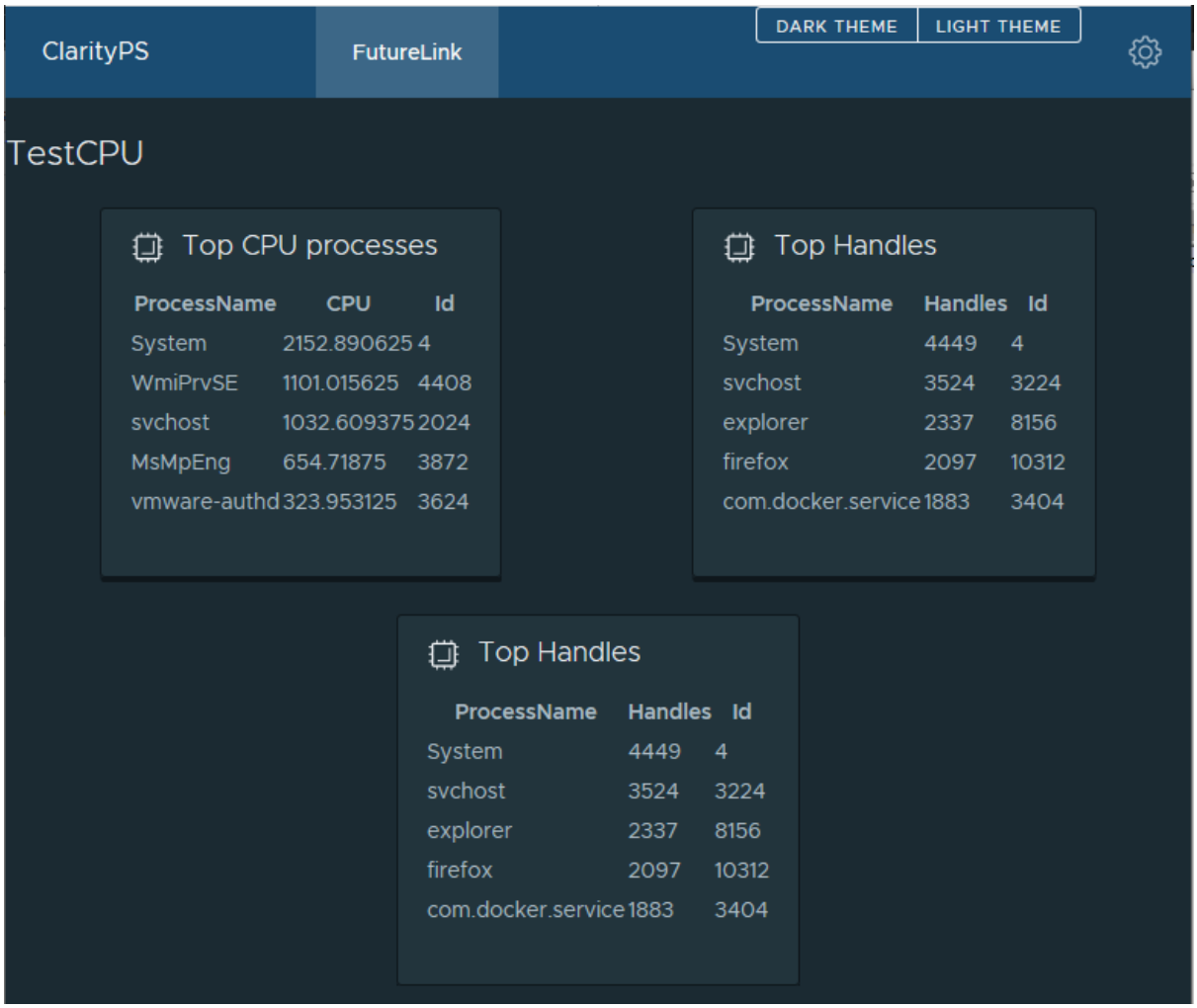
HTML cards are a user interface object that organizes data clearly and concisely. When a HTML document utilizes multiple cards on the same webpage the cards are placed on a balanced grid to allow for auto scaling.



Single row of cards on a wide browser page.

ReportCardPS utilizes a `flex-container` element which allows for the grid of cards to be generated dynamically based on the card width and the page width. See the image below where the browser window has been dragged to the left and the result is a card being bumped to a second row.

[176] https://clarity.design/
[177] https://github.com/jpsider/ClarityPS



Additional row due to narrow page width.

The two images used above were rendered with the same exact HTML—only the browser window's width was changed.

Getting Started

Configuring Your Environment for This Chapter

Creating a clean environment is important when running any script or application.

The `Import-Module` command requires the script execution policy be set to Remote-Signed[178] or a less restricted policy[179].

[178] https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.security/set-executionpolicy?view=powershell-6

[179] https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_execution_policies?view=powershell-6&viewFallbackFrom=powershell-Microsoft.PowerShell.Core

```
# Determine the current script execution policy.
Get-ExecutionPolicy
```

Output:

```
Restricted
```

```
# Execute the following command to change the PowerShell execution policy.
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned
```

```
# Verify the command updated the script execution policy.
Get-ExecutionPolicy
```

Output:

```
RemoteSigned
```

Install ReportCardPS

First, ensure you can find the module in the PowerShell Gallery[180]. This will require the computer have a connection to the internet.

```
Find-Module -Name ReportCardPS -Repository PSGallery
```

```
Version    Name                                Repository           Description
-------    ----                                ----------           -----------
0.0.12     ReportCardPS                        PSGallery            PowerShe...
0.0.14     ReportCardPS                        PSGallery            PowerShe...
```

Next, install the ReportCardPS PowerShell module:

[180] https://www.powershellgallery.com/packages/ReportCardPS/0.0.14

```
$InstallParams = @{
  Name = 'ReportCardPS'
  RequiredVersion = '0.0.14'
  Repository = 'PSGallery'
  Scope = 'AllUsers'
  Force = $true
}
Install-Module @InstallParams
# Import the ReportCardPS module
Import-Module -Name ReportCardPS
```

Then, verify the ReportCardPS module is installed by utilizing `Get-Module`:

```
Get-Module -Name ReportCardPS -ListAvailable
```

Output:

```
    Directory: C:\Program Files\WindowsPowerShell\Modules

ModuleType Version    Name                                ExportedCommands
---------- -------    ----                                ----------------
Script     0.0.14     ReportCardPS                        {Get-ReportCardTemp...
```

Finally, verify the ClarityPS module is installed by utilizing `Get-Module`:

```
Get-Module -Name ClarityPS -ListAvailable
```

Output:

```
    Directory: C:\Program Files\WindowsPowerShell\Modules

ModuleType Version    Name                                ExportedCommands
---------- -------    ----                                ----------------
Script     0.3.0     ClarityPS                           {Add-Branding, Add...
```

Inspect the ReportCardPS Module

You can use the `Get-Command` function to review the available commands that are provided in the ReportCardPS module.

```
Get-Command -Module ReportCardPS
```

Output:

```
CommandType     Name                                Version    Source
-----------     ----                                -------    ------
Function        Get-ReportCardTemplateSet           0.0.14     ReportCardPS
Function        New-ReportCard                      0.0.14     ReportCardPS
```

The ReportCardPS Manifest File

The report manifest file is a JavaScript Object Notation (JSON) document which details the order of and information about the cards that will be in the HTML report document. Each card is described with the following information:

- `CardTitle`: Brief description of the card
- `CardFunction`: PowerShell function to be used to generate the card HTML
- `CardArguments`: Parameters to be used when executing the PowerShell function
- `Order`: Order of the cards on the HTML document.

Here is an example of a basic report manifest file:

```
[
  {
    "CardTile": "CPU",
    "CardFunction": "Get-LocalProcUsage",
    "CardArguments": "-Count 5",
    "Order": "1"
  },
  {
    "CardTile": "Handles",
    "CardFunction": "Get-LocalHandleCount",
    "CardArguments": "-Count 5",
    "Order": "2"
  }
]
```

Locate ReportCardPS Manifest Files

ReportCardPS includes example report manifest files in the module's `lib` folder. You can locate the example files with the included function `Get-ReportCardTemplateSet`:

```
Get-ReportCardTemplateSet
```

Output:

```
# Full paths are not displayed.
FullName
C:\...\Modules\ReportCardPS\0.0.14\lib\Generic\LocalCPUReport.json
C:\...\Modules\ReportCardPS\0.0.14\lib\VMware\vCenterHomeReport.json
```

This chapter will utilize `LocalCPUReport.json` as the example report manifest file.

Creating a Basic Report

ReportCardPS includes the functions and report Manifest file to create a basic report on CPU usage and performance for your local machine. To create a report execute the `New-ReportCard` command with the following parameters:

- `Title`: Name of the report that will be displayed on the HTML Document
- `JsonFilePath`: Full Path to the report manifest file

By default, ReportCardPS returns the HTML document back to the PowerShell pipeline. In the example below the HTML file is output to a file in the `C:\temp` directory.

```
$NewReportParams = @{
  Title = 'ExampleReport'
  # Full path is not displayed.
  # Use the full path returned from Get-ReportCardTemplateSet.
  JsonFilePath = 'C:\...\ReportCardPS\0.0.14\lib\Generic\LocalCPUReport.json'
}
$OutputParams = @{
  FilePath = 'C:\temp\exampleReport.html'
  Encoding = 'ascii'
}
New-ReportCard @NewReportParams | Out-File @OutputParams
```

You can open the newly created file `C:\temp\exampleReport.html` in a web browser.

Creating Custom Functions

The key to ReportCardPS is the custom PowerShell functions. These functions are used to gather the data and create the HTML used to render the cards. Each of these custom functions must be imported into the PowerShell session running the `New-ReportCard` function. The main steps that each function needs are:

1. Accept parameters.
2. Gather raw data.
3. Convert the raw data to HTML.
4. Assemble card HTML.
5. Return the html to the pipeline.

Below is a snippet from an example function similar to one found in the ReportCardPS module.

```
# Gather the raw data.
$rawData = Get-Process | Sort-Object -Property CPU -Descending

# Convert the raw data to HTML
$DataHTML = $rawData | ConvertTo-Html -Fragment

# Build the HTML Card, with a Title and Icon
$Card = New-ClarityCard -Title "Top CPU processes" -Icon cpu -IconSize 24
$CardBody = Add-ClarityCardBody -CardText "$DataHTML"
$CardBody += Close-ClarityCardBody
$Card += $CardBody
$Card += Close-ClarityCard -Title "Close CPU Card"

# Return the HTML to the pipeline
$Card
```

The example provides the basics for gathering data and creating a simple card. It's recommended to explore the Clarity Documentation[181] to learn about additional HTML elements you can add to the cards. You may also want to review the latest functions from ClarityPS[182].

Summary

The ReportCardPS PowerShell module was created to give system administrators another tool to create custom reports utilizing VMware's Clarity HTML elements. The functions included in ReportCardPS[183] and ClarityPS[184] can be used to create elaborate and customized reports to display a variety of data points. Contributions, bug reports, and suggestions are welcome to both PowerShell modules as they're open source projects.

[181] https://clarity.design/documentation
[182] https://clarityps.readthedocs.io/en/latest/
[183] https://github.com/jpsider/ReportCardPS
[184] https://github.com/jpsider/ClarityPS

Part V: Cloud Operations

This section focuses on using PowerShell in against public cloud providers, leveraging their APIs and offerings to get things done.

Part VI: Culture

This section focuses on all of the parts of work that aren't in a console or web browser - how work has and will change, how to be a better teammate, how to grow as an engineer.

Where PowerShell Has Taken Work

by Jon Junell

It's foundational to PowerShell to be declarative. As folks who work, we benefit from that same declaration of intent. PowerShell has removed much of the meandering from our technical workflows, and we can now share our declarative work with others in a consumable, repeatable, and collaborative way.

Equally importantly, the introduction of PowerShell hit a reset button for getting into our trade. Familiarity with the GUI has given way. What you did with NT 3.51 is no longer so important; it's about what you're doing today, and by leveraging PowerShell, you've supercharged what you can accomplish. A script isn't a written note about a process or procedure, it's a structured, source-controlled document that can be applied and reapplied easily in an intended environment.

Reminiscing

What's new was old, and what was old is new again. If you look through the internet archive or (perhaps) a really well-stocked thrift store you might be able to find early computer magazines from the 1980s. One staple of these magazines was the inclusion of programs that you could type into your computer by hand.

Now, a thousand lines of BASIC might make for a pretty dull read today, but it's an example of shared code thrown out into the broader world: to be shared, tested, mistyped, and—most importantly—modified by the reader. This gave you the ability to do something yourself. You had quite an opportunity if you were lucky enough to share a program that you typed in with your friends, peers, or computer club; you could gain a bit of notoriety by sharing your hard work with others who didn't have the magazine, the time, or the ability to focus on tiny printed lines of code.

GUI

Jokingly, I *could* say that Xerox ruined computers with a GUI. By making computers more accessible, Xerox opened up computing to a whole new audience—and early Macintosh computers did much the same. Programming moved into the background, becoming something that only a subset of users engaged in, instead of all. VisiCalc and its compatriot programs started defining what a computer application could be. And over time, the conversation morphed from *make*, to *buy*. "I have a task, so I'll build a tool" became "I have a task, so I'll buy the tool that fits the job."

It's true that although buying a tool *can* restrict creativity, it's unwise to talk in absolutes. However, we know from research with children and perception[185] that humans rely on information (books, blogs, etc.) and each other to form opinions and gain understanding. If you're told that you have to look at a packaged solution, you may never take the opportunity to build something on your own.

Click Next

Here's an experiment: Tell a friend what you think are the best directions to your favorite restaurant, and don't let them write anything down. Note the despair in their eyes when you don't give them the street address, probably the most critical piece of information they'll need in order to actually get to the restaurant! Leaving out this level of information leads to confusion, misinterpretation, and editorializing. If you want to make sure that they get to the restaurant, and at the right time, you'll need to transfer the important information succinctly. For example, the address can't work only in MapQuest; it needs to work wherever the listener gets their directions. (Okay, maybe I'm making an argument for GPS coordinates.)

If your instructions are "click next in the dialog box" instead of declarative as `Repair-DbaDbMirror`, you're going to run afoul of context really quickly.

Defined Patterns and Slowed Innovation

If everyone has agreed that Excel is the only way you'll ever manage a spreadsheet, does it limit the conversation? Digging more, if the only accepted ways to automate a task are through Perl or C, then why would there be any reason to do something different? There isn't. This has parallels to competition in business; if it's accepted that there's only Choice A and Choice B, it becomes very difficult to create Choice C.

Moving Forward

Communication can often be the prime mover. For example, in my organization, Active Directory exists only because of the implementation of Microsoft Exchange, which required the adoption of a directory service, and hence a move from Novell (or Banyan for the cool kids). In this scenario, Active Directory was almost a byproduct of Exchange.

Well, fast forward, and Exchange is now integral to communication in of lots of companies. Unfortunately, it's a huge product, with lots of sprawl. You've got Mailbox, Transport and Client Access servers, not to mention the underlying SQL Server architecture. One might argue that Exchange circa 2007 was the start of a small cloud, contsisting of similarly configured, load-balanced servers. And we learned that you couldn't efficiently manage an environment like that with clicks, but you could with that newfangled PowerShell. This moved configuration and management from Microsoft Installers (MSIs) and answer files to readable, reproducible code—just like our goal when we first opened an issue of Byte magazine. Now, as an IT Pro, you can spin

[185] How Language Shapes Our Perception of Reality

up a test environment consistently and accurately. Combine this with the rise of virtualization (VMware went public in 2007), and we've got the beginnings of repeatable infrastructure.

In an early demonstration of PowerShell, from October 2004[186], Jeffrey Snover talks about how naming and consistancy are critical to the quick adoption of new technologies. After showing some of the commands and syntax to find system processes, he turns to the camera and tells a story. He says, there's a joke in the VMS community, about how you can sit a VMS operator down, tell them that there's a fish finder attached to the VAX, and question whether they can operate it. The answer from the VMS operator is, of course, `show /fishfinder`, `get /fishfinder`, etc.

While a fish finder on a minicomputer is pretty absurd, change and new products aren't; replace fish finders with SQL Server, or Azure functions, and you see the importance. True, it's all part of the perfect storm: a rapid rise in access to computing resources, the sprawl of feature requirements, changes in virtualization technology—so many little things gave PowerShell and its community a great place to start and thrive. However, these are just pieces of technology, and they do very little to address the folks that sit between the keyboard and that chair.

Nod to Cultural Change

PowerShell and its community wouldn't be where they are today without the rise of companies like FaceBook, Google, Twitter and Microsoft. By taking on the primordial challenge of *big change* these companies had to transform existing business models to adapt them to customer demands. These needs at scale gave us the cloud as we understand it today–and for us to manage that scale we had to change the way we work, moving from treating our servers as pets, to treating them as *cattle*. We had to adopt DevOps practices as we know them, with all of their transparency, source control, and team collaboration. (While these are all great topics for another time, we're here to talk about what DevOps did for PowerShell—this is a PowerShell Conference Book after all!)

Current Location

That amazing shift set the stage for a reboot of our environments. I would argue that, as IT Pros, we've been granted a fresh start. But many of us still run legacy systems–SMB v1 still haunts us despite Ned Pile[187]–and we might need to run a few hosts on old versions of software, for various shaky reasons. However—and this is the real point—the rest of our infrastructure is working in ways that are more accessible and designed in that spirit of transparency. In changing the way we work, we've removed the biggest barrier to change: gatekeeping.

Lords of the Gate

Think back to your first installation of SQL Server or CentOS. You probably installed it one step at at time, line by line, click by click. If you were to describe to someone else how to perform

[186] Jeffery Snover - Monad demonstrated

[187] Running SMB1 is like taking your grandmother to prom: she means well, but she can't really move anymore. Also, it's creepy and gross

the installation, it would be much like those anecdotal directions to your favorite restaurant: a meandering tale of "First do this, then this, and oh, then do that…"

Compare that to what you'd do today, which is (I hope) more like: "Here's what I wrote for my environment; you can see where I reference some of our internal practices, but read through it and hit me up on Slack if you have questions."

Even better, there are numerous public repositories containing examples of what other people have done, plus blogs and social media posts that let folks find things on their own. There's an easily accessible *community* of people who share your trade and are willing to put themselves out there as a resource for others. There's been a big reduction in "who you need to know" to get access to this knowledge. It's now easier (although not always easy) for folks with some internal motivation, and there are plenty of ways around traditional groups.

The point is, we've gotten back to "here's the code; take a look."

"But They Have Suspenders!"

We've got a second reason to be thankful for changes resulting from the adoption of cloud computing: it provides a strong incentive for companies to look at the environment (read: market), see where growth is, judge what their competitors are doing, and make some decisions. If part of your portfolio of services involves hosting Windows VMs or .NET-related services, you're going to find PowerShell in the ecosystem.

Mix this with a Microsoft that's hard at work going back to its roots—providing really great integrated development environments (Visual Studio Code) and making sure their products are everywhere (.NET Core)— and you end up with PowerShell running on Linux, and partners who make sure their modules work on the cross-platform implementation of PowerShell. You see things like AWS Tools for Windows PowerShell[188] and VMware's PowerCLI[189].

When this kind of expansion across environments occurs, combined with an open tradition of sharing, you create a broad-based appeal that allows people with diverse experiences to join the conversation. Been a Unix admin for 30 years, but are trying some cloud deployments? Here's some PowerShell. Are you a content designer and you want easier deployments—maybe even send your customers a demonstration environment? Here's some PowerShell. These folks come to the *community* and, because of the adopted norms, they can start contributing—which, in turn, enriches the content that makes up the *community's* body of knowledge. They stop being **people in suspenders**, or **people with MacBook Airs**, and are just *folks*.

Tale of Two Definitions

By this stage, I hope you're looking at the PowerShell community in a different light, and maybe even smiling, because—just by reading this book, following folks, and most importantly contributing in some large or small way— you understand that it's something bigger and more special than one user group or one message board. Here are two definitions that might contribute to your understanding.

[188] AWS Tools for PowerShell
[189] VMware PowerCLI

A Community of Practice

> A community of practice is a group of people who share a concern or a passion for something they do, and learn how to do it better as they interact regularly.
>
> This definition reflects the fundamentally social nature of human learning. It's very broad. It applies to a street gang, whose members learn how to survive in a hostile world, [...] a group of engineers who learn how to design better devices or a group of civil servants who seek to improve service to citizens.[190]

Communities Aren't Just Teams

> A community of practice is held together by the "learning value" members find in their interactions. They may perform tasks together, but these tasks don't define the community. It's the ongoing learning that sustains their mutual commitment. Members may come from different organizations or perspectives, but it's their engagement as individual learners that's the most salient aspect of their participation. The trust members develop is based on their ability to learn together: to care about the domain, to respect each other as practitioners, to expose their questions and challenges, and to provide responses that reflect practical experience.[191]

(These definitions may seem a bit dense, but Etienne & Beverly Wenger-Trayner created the concept, and have a whole gig associated with it.[192])

I was fortunate enough to participate in a Community of Practice in my local area, sponsored by Community Engagement Fellows.[193] From the very first day, when communities of practices were discussed, I was amazed to realize at how well the actions of the broader PowerShell community already reflect this relatively rare and powerfully collaborative practice.

Where To Go From Here

The point of this chapter is simply to help you understand how special this community is. And the point of this book is to broaden access, to break down financial barriers, and to build a more inclusive and diverse community in which we can all benefit. It's a charge to you, reader, to go from *not knowing that you know*, to *knowing that you know*. This is powerful and deliberate, just like your code, just like your source repositories.

Share, encourage, and contribute. This can be why we work.

[190] What's a Community of Practice
[191] How are Communities of Practice different from Teams or Taskforces
[192] Wenger Trayner
[193] Community Engagement Fellows