The PowerShell Conference Book Volume 3

Foreword by Don Jones

Edited by

Mark Kraus, Michael Zanatta,
Adil Leghari, Phil Bossman,
Christian Coventry,
Joe Houghes,
Steven Judd, Bill Kindle,
and Arnaud Petitjean

the DEVOPS collective

The PowerShell Conference Book

Volume 3

Mark Kraus, Michael Zanatta, Joe Houghes, Christian Coventry, Steven Judd, Phil Bossman, Adil Leghari, Arnaud Petitjean and Bill Kindle

This book is for sale at http://leanpub.com/psconfbook3

This version was published on 2020-09-21

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.

© 2020 The DevOps Collective, Inc. All Rights Reserved

Tweet This Book!

Please help Mark Kraus, Michael Zanatta, Joe Houghes, Christian Coventry, Steven Judd, Phil Bossman, Adil Leghari, Arnaud Petitjean and Bill Kindle 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 Volume 3!

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

Contents

Foreword

by Don Jones

By supporting this book, either as a contributor or a purchaser, you've become part of a long tradition in the PowerShell community: a tradition of bootstrapping our own efforts and supporting each other in those efforts. Especially around conferences.

Once upon a time, Microsoft itself ran conferences around some of its major products: Exchange Server, the System Center family, SharePoint, their developer products, and more. But never for PowerShell. The PowerShell team was able to cajole NetPro, the software vendor behind "The Experts Conference," into running a 50-person "PowerShell Deep Dive." That lasted for two years until Quest Software acquired NetPro, and "The Experts Conference" was discontinued. So the PowerShell team turned to the one ally they knew wouldn't ever let them down, and wouldn't ever go away: the community itself.

PowerShell's community of users was already widespread by then. But they'd started pulling together one-day, low-cost "PowerShell Saturday" events, inspired by the SQL Server community's "SQL Saturdays" (notably, Microsoft never ran their own SQL Server conference either, instead partnering with a community organization to run PASS). When Microsoft said, "We really need a place to engage with our PowerShell power users," the community stepped up, and in 2013 PowerShell Summit North America was born. That spawned PowerShell Summit Europe for two years and eventually morphed into PowerShell + DevOps Global Summit. PowerShell Conference Asia and PowerShell Conference Europe were launched, demonstrating the global breadth and commitment of PowerShell's fan base. Importantly, all of these events were run as not-for-profits, providing a service to the community and basically attempting to break even.

PowerShell + DevOps Global Summit, run by the nonprofit DevOps Collective, Inc., started to take on more responsibilities. The organization runs a small, close-knit DevOps Camp event in the summers, and helps run several PowerShell Saturday and other regional events throughout the year. They provide financial support to other nonprofits that help bring young people from disadvantaged situations into the IT field. They continue to run the PowerShell.org[1] website, and do a lot on a daily basis to provide rallying points and networking opportunities for the PowerShell community. Your purchase of this book helps support that organization's efforts throughout the year, and you owe yourself a little pat on the back for helping out.

2020 was a bad year for conferences. The SARS-CoV-2 virus forced pretty much every conference to either "go virtual" or cancel outright, which is a shame. While I personally love the reach that a virtual event can have, there's simply no way to replace the kind of in-person contact and networking that live conferences have. PowerShell + DevOps Summit, in particular, was well-known for its friendly, easy-to-talk-to audience. It's like a giant user group that meets once a

[1] https://powershell.org

year, and its attendees routinely praise not only its content but its positive impact on their overall careers. Summit will return, make no mistake, and I hope you'll make every effort to attend in person when it does.

In the meantime, this conference-in-a-book is a fantastic way to capture some of the technology presentations that you probably would have seen at Summit. Most of the authors who contributed to this book aren't professional writers, which makes their efforts even more valuable. It's hard, sometimes, to "break out of your shell" and contribute to a work like this one. But when you do, as I think most of these authors will tell you, it's incredibly rewarding. It's doubly rewarding when your efforts are appreciated by others, so please: take a minute to reach out to these authors on social media and thank them for their contributions here. Perhaps you can contribute to next year's volume and learn what it feels like!

I've worked within the PowerShell community since its inception in 2006, and every year I'm more and more proud of it. An increasing number of people are noticing needs in the community—like the need for this book—and stepping up to help make it happen. *That* is what community is all about: rather than waiting for someone else to "get 'er done," you jump in and help make it happen yourself. This community has probably done more to help itself than any tech community I've ever been a part of, and it's amazing to see.

So enjoy this Third Edition of the *PowerShell Conference Book*. Tell your colleagues about it, and ask them to buy their own copy. Heck, ask the boss to pick up copies for the whole team. In doing so, you're not only adding to your own knowledge base and enhancing your career, you're supporting a wonderful organization and an amazing community of people that gives us all so much value.

Thank you for your support, and thank you for being a part of the community.

Don Jones

Co-Founder, The DevOps Collective, Inc.

Contributors

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

Jordan Borean

Role: Author

Jordan is a Software Engineer at Red Hat[2] working on the Windows integrations for Ansible. He originally worked on Java-based programs for a large company but felt the draw to open source software and has been an avid contributor since. Jordan mostly focuses on Python and PowerShell based languages, and he is committed to trying to bridge the Windows and Linux worlds and make it easier for them to work with each other. Some projects that he works on are pypsrp[3], smbprotocol[4], pypsexec[5], and more recently pyspnego[6]. When finding some free time, Jordan blogs on Blogging for Logging[7] that cover technologies like PowerShell, Ansible, Windows protocols, and anything else that takes his fancy. You can usually get in contact with him on the PowerShell Discord server[8], or various IRC Freenode channels like `#ansible`, `#Powershell`, `#packer-tool`, and others.

Phil Bossman

Role: Author, Editor

Phil is an accomplished Windows administrator and Citrix architect for a national building materials supplier. He 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. He is a contributing author to *The PowerShell Conference Book Volume 2*. Phil is a co-organizer of the Research Triangle PowerShell User Group[9] and an active member of the Citrix User Group Community (CUGC)[10]. You can follow him on Twitter, @Schlauge[11], and Github[12], or check out his blog[13].

[2] https://www.redhat.com/en
[3] https://github.com/jborean93/pypsrp
[4] https://github.com/jborean93/smbprotocol
[5] https://github.com/jborean93/pypsexec
[6] https://github.com/jborean93/pyspnego
[7] https://www.bloggingforlogging.com/
[8] https://aka.ms/psdiscord
[9] https://rtpsug.com
[10] https://www.mycugc.org
[11] https://twitter.com/Schlauge
[12] https://github.com/pbossman
[13] https://schlauge.com

Dave Carroll

Role: Author

Dave enjoys being part of the welcoming PowerShell community and writes about PowerShell and other IT topics on his blog[14]. Over the last twenty-plus years, he has focused on automation, with half of that time spent honing his PowerShell skills, from the first command to his open-source modules. He created and maintains PoshEvents[15] and PoshGroupPolicy[16], both useful tools to harvest data. You can find him on Twitter, @thedavecarroll[17], where he tweets and amplifies PowerShell discussion and information sharing.

Mateusz Czerniawski

Role: Author

Mateusz is a System Architect at Objectivity[18] who grew his skills in PowerShell on Hyper-V, Active Directory, and everything related. He is also a co-founder of the Polish PowerShell User Group[19] and a speaker. This year he was awarded his first MVP in Datacenter and Cloud Management. His greatest achievement so far is bringing more people to the PowerShell community every day. "But if I can help somebody as I pass along, then my living will not be in vain." You can reach him via his blog[20].

Tomasz Dabrowski

Role: Author

Tomasz is a Senior IT Systems Engineer at Objectivity[21], speaker, and co-founder of the Polish PowerShell User Group[22]. He is a father of three children, aged 5, 3, and 1, and is working full time from home. Tomasz works with PowerShell daily, focusing on Hyper-V, Active Directory, Azure, Office 365, and Veeam. He is super passionate about PowerShell automation at work and at home. You can reach him via his blog[23].

Luc Dekens

Role: Author

Luc Dekens is a Senior Systems Engineer, Microsoft MVP, VMWare vExpert, author, speaker, and subject matter expert for all things VMWare. He is super passionate about automating VMWare

[14] https://powershell.anovelidea.org
[15] https://github.com/thedavecarroll/PoShEvents
[16] https://github.com/thedavecarroll/PoShGroupPolicy
[17] https://twitter.com/thedavecarroll
[18] https://www.objectivity.co.uk
[19] https://www.meetup.com/Polish-PowerShell-Group-PPoSh
[20] https://www.mczerniawski.pl
[21] https://www.objectivity.co.uk
[22] https://www.meetup.com/Polish-PowerShell-Group-PPoSh
[23] https://dombrosblog.wordpress.com

products and is an active member contributing to the VMWare PowerCLI community. Luc is a co-author of the "VMware VSphere PowerCLI Reference: Automating VSphere Administration" Edition 1 and 2 books.

Matthew Dowst

Role: Author

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 Catapult Systems' Managed Automation Team. You can follow him on Twitter, @mdowst[24], and check out his blog site, Dowst.Dev[25], where he provides tons of PowerShell snippets and tips, and a weekly roundup of all things PowerShell.

Saggie Haim

Role: Author

Saggie is a Cloud Security Architect, tech trainer, and a local speaker. He has been involved with the PowerShell Conference Book since volume 2. He is a PowerShell and automation enthusiast, writing about it on his blog[26] You can check his GitHub profile[27] for more details.

Maciej Horbacz

Role: Author

Maciej is passionate about automating every task, process, or issue he runs into. At the beginning of his journey with PowerShell, he used it for managing dozen of users in Active Directory. Later on, he started to write more complex scripts and functions for Office 365 services like Sharepoint and Exchange Online. Today he is creating complex automation solutions and manages a fleet of 700+ devices using Microsoft Intune. From PowerShell through Microsoft Graph, Power Platform, Azure Automation to Azure DevOps pipelines, he publishes his ideas on his blog, UniverseCitiz3n[28]. He is also a co-organizer of the Polish PowerShell User Group[29].

Joe Houghes

Role: Editor

[24] https://twitter.com/MDowst
[25] https://www.dowst.dev/
[26] https://www.saggiehaim.net
[27] https://www.github.com/saggiehaim
[28] https://universecitiz3n.tech/
[29] https://meetup.com/pl-PL/Polish-PowerShell-Group-PPoSh

Husband, Father, Community Geek. Joe Houghes is a co-leader of @ATXPowerShell[30] and @AustinVMUG[31] user groups in Texas and a member of the @vBrownBag[32] crew. He is currently a Solutions Architect for Veeam, focused on automation & integration. Joe spends most of his time working within VMware environments when he is not active in planning or hosting community events. You can find Joe on Twitter, @jhoughes[33], or his blog[34].

Jeremy I Brown

Role: Author

Jeremy is a Microsoft Solution Architect for the North Carolina Judicial Branch. He has used PowerShell since v1. At work, he spends most of his time automating for Office 365 and Azure. In his free time, he enjoys creating serverless applications, co-managing the RTPSUG[35] user group, and volunteering for The DevOps Collective. Outside of technical work, his preferred sport is wakeboarding.

Don Jones

Role: Foreword Author

Don is PowerShell's "First Follower" and started with PowerShell in 2005. He co-authored the first published books on PowerShell and helped found PowerShell.org. Don's final PowerShell book, "Shell of an Idea," covers the untold history of the shell.

Steven Judd

Role: Author, Editor

Steven Judd is a 25+ year IT Pro and currently a Digital Security Analyst at Devon Energy Corporation[36] with an emphasis on DevOps and cloud-focused solutions and infrastructure. He has been using PowerShell since 2010 and co-developed a custom training program for PowerShell. He loves to help people learn and recognize the value of automation. He spends his free time learning more about PowerShell, digital security, and cloud technologies, along with creating and telling Dad jokes[37]. You can find him hanging out on the PowerShell Discord Server[38] bridge channel, taking care of his family, running marathons, playing the cello, plus a handful of other hobbies he can't seem to quit. Please follow him on Twitter, @stevenjudd[39], read his blog[40], and review, use, and improve his code on Github[41].

[30] https://twitter.com/ATXPowerShell
[31] https://twitter.com/AustinVMUG
[32] https://twitter.com/vbrownbag
[33] https://twitter.com/jhoughes
[34] https://www.fullstackgeek.net/
[35] https://rtpsug.com/
[36] https://www.linkedin.com/in/stevenjudd/
[37] https://www.youtube.com/watch?v=BZZM6i8AE1Y
[38] https://aka.ms/psdiscord
[39] https://twitter.com/stevenjudd/
[40] https://blog.stevenjudd.com/
[41] https://github.com/stevenjudd

Bill Kindle

Role: Editor

Husband to a wonderful woman that he doesn't deserve, and the father of two adorable children. Bill is a former career Systems Administrator turned Cyber Security Engineer currently working for Corsica Technologies[42]. Bill was an author for The PowerShell Conference Book Volume 2. His role focuses on automation engineering and supporting a Security Operations Center. Bill has a passion for helping others in IT and occasionally does presentations for @FortWayneVMUG[43]. You can find some of Bill's work over at AdamTheAutomator[44] and TechSnips LLC[45].

Josh King

Role: Author

Geek, Father, Walking Helpdesk. Josh King is a Microsoft MVP and TechOps Systems Administrator at Tribe, an IT services organization in New Zealand. Josh predominantly works within Windows and VMware environments and has a passion for all things PowerShell. He previously contributed to the PowerShell Conference Book Volume 2. You can find Josh on Twitter, @WindosNZ[46], or his blog[47].

Mark Kraus

Role: Editor in Chief

Mark has been involved with the PowerShell Conference Book since volume 1. Mark is an Engineering Manager at LinkedIn[48]. He is also a former Microsoft MVP[49], a DevOps and PowerShell Blogger[50], and a Twitch coding streamer[51]. He is the creator of PowerShell Live[52], a PowerShell contributor, and an open source maintainer/developer[53]. You can follow his escapades on Twitter, @markekraus[54].

Kevin Laux

Role: Author

[42] https://corsicatech.com
[43] https://twitter.com/FortWayneVMUG
[44] https://adamtheautomator.com
[45] https://techsnips.io
[46] https://twitter.com/WindosNZ
[47] https://toastit.dev/
[48] https://www.linkedin.com/in/markekraus/
[49] https://mvp.microsoft.com/en-us/PublicProfile/5002898?fullName=Mark%20%20Kraus
[50] https://get-powershellblog.blogspot.com
[51] https://www.twitch.tv/markekraus/
[52] https://www.twitch.tv/powershelllive
[53] https://github.com/markekraus/
[54] https://twitter.com/markekraus/

Kevin is a Platform Engineer who recently made a move into Management. He is passionate about PowerShell and has been leading training classes for his colleagues since the release of PowerShell v3. Kevin also serves as co-leader of the Research Triangle PowerShell User Group[55]. In addition to PowerShell, he is always tinkering with new technology in his home lab and trying to learn everything he can. You can follow him on Twitter, @rsrychro[56], and Github[57].

Frank Lesniak

Role: Author

Frank is a Senior Cloud & Infrastructure Architect at West Monroe[58], where he leads the technology automation and digital workplace consulting teams. While Frank spends much of his time working with PowerShell, he jokes that he isn't actually any good at it because he specializes in backward-compatibility – Frank writes code compatible back to PowerShell v1, when possible. Frank is active in the PowerShell community and is a co-organizer and frequent presenter for the Chicago PowerShell Users Group[59]. Frank is also a board member with DuPage Animal Friends[60], a nonprofit supporting life-saving and innovative initiatives at DuPage County Animal Services, a premier open admission animal shelter in the Chicago area. Frank is active on Twitter, @FrankLesniak[61], and can also be reached on LinkedIn[62].

Tommy Maynard

Role: Author

Tommy Maynard has been active in the PowerShell community for several years. It took some time initially, but he gave up Visual Basic Script, telling himself back at the beginning of this journey that his *next* automation project would be written with PowerShell instead. He never looked back. In June 2014, he began blogging at tommymaynard.com[63], and eventually at PowerShell.org[64], too. To this day, he uses the Twitter PowerShell Hashtag[65] to locate PowerShell-related content to consume. He shares his blog posts on Twitter, @thetommymaynard[66]. There appears to be no end in sight to his desire to automate, solve problems, and provide content for others that want to learn PowerShell. Tommy lives in Tucson, Arizona, with his wife Jenn and kids, Carson and Arianna. He is employed as a Senior Systems Administrator and is always on the hunt for new technologies to learn or anything he can automate.

[55] https://rtpsug.com
[56] https://twitter.com/rsrychro
[57] https://github.com/KevinLaux
[58] https://wmp.com
[59] https://www.meetup.com/chgopsug/
[60] http://dupageanimalfriends.org
[61] https://twitter.com/FrankLesniak
[62] https://linkedin.com/in/flesniak
[63] https://tommymaynard.com
[64] https://powershell.org
[65] https://twitter.com/search?q=%23PowerShell
[66] https://twitter.com/thetommymaynard

Chan Nyein Ko Ko

Role: Author

Chan Nyein Ko Ko (Chan) is a Cloud Engineer who loves learning new technologies and developing automation. The PowerShell 3.0 Jumpstart video series introduced by Jeffery Snover and Jason Helmick on Microsoft Virtual Academy (MVA) in 2014 inspired him to learn PowerShell. Since then, he has accomplished most of his daily tasks with PowerShell scripts. Today he works for Deloitte Cloud Engineering[67] in Sydney, Australia as part of the DevOps team. He migrates on-premise workloads to AWS and develops PowerShell scripts to automate AWS resources. You can reach out to him at LinkedIn[68] and the PowerShell Slack Channel[69].

Brandon Olin

Role: Author

Brandon is a Site Reliability Engineer at Stack Overflow[70] and a Microsoft MVP in Cloud and Datacenter Management[71]. He has an affection for PowerShell and DevOps methods and uses them to help drive the community forward. He is active in the PowerShell community, with many open-source projects available on the PowerShell Gallery[72]. Brandon is also the author of Building PowerShell Modules[73] and ChatOps the Easy Way[74] and is a contributing author to the PowerShell Conference Book 1[75]. You can check out his code on GitHub[76], his blog at devblackops.io[77], or reach him on Twitter, @devblackops[78].

Paul Ou Yang

Role: Author

Paul is a seasoned professional with vast experience in database administration, data visualization, and automation. He is a Microsoft Certified Master in SQL server and a Solutions Architect for Micron Technology, Inc[79]. When he is not playing with his three children, he blogs at paulouyang.blogspot.com[80].

[67] https://www2.deloitte.com/au/en/pages/strategy-operations/solutions/cloud-engineering.html
[68] https://www.linkedin.com/in/channyeinkoko/
[69] https://powershell.slack.com/team/U0115E539M5
[70] https://stackoverflow.com/
[71] https://mvp.microsoft.com/en-us/PublicProfile/5003334?fullName=Brandon%20%20Olin
[72] https://www.powershellgallery.com/profiles/devblackops/
[73] https://leanpub.com/building-powershell-modules
[74] https://leanpub.com/chatops-the-easy-way
[75] https://leanpub.com/powershell-conference-book
[76] https://github.com/devblackops
[77] https://devblackops.io/
[78] https://twitter.com/devblackops
[79] https://micron.com
[80] https://paulouyang.blogspot.com

Justin P. Sider

Role: Author

Mr. Sider has 15+ years of experience as owner of his own tech business and in leadership roles for various tech companies. As 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 Mr. Sider on Twitter, @jpsider[81], at the PowerShell Gallery[82], on GitHub[83] or read an occasional blog at Invoke-Automation[84].

James Petty

Role: DevOps Collective Liaison

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

Rob Pleau

Role: Author

Rob is a lover of all things PowerShell and Linux and has been writing PowerShell for over seven years. He writes occasional blog entries at https://ephos/github.io[85] about any number of PowerShell or Linux topics. Rob is also an occasional speaker at the PowerShell + DevOps Summit and the Boston PowerShell User Group. He is also a volunteer for the DevOps Collective OnRamp program.

Thomas Rayner

Role: Author

[81] https://twitter.com/jpsider
[82] https://www.powershellgallery.com/profiles/jpsider/
[83] https://github.com/jpsider
[84] https://invoke-automation.blog/
[85] https://ephos/github.io

Thomas Rayner is a Senior Security Service Engineer at Microsoft with 15+ years' experience in technology automation, software development, cloud & on-premises enterprise infrastructure, continuous integration & delivery, and helping make his teammates better. You can get in contact with Thomas by email at thmsrynr@outlook.com[86], on Twitter, @MrThomasRayner[87], or on LinkedIn[88].

Kevin Sapp

Role: Author

Kevin is a Cloud Platform Engineer at LinkedIn[89] with over a decade of experience in PowerShell automation. He founded CodeDuet[90], a blog for sharing tech industry news and how-to tutorials related to DevOps. Follow him on AdamtheAutomator[91] and GitHub[92].

Michael Zanatta

Role: Senior Editor

Michael has been involved with the PowerShell Conference book since Volume 2. Michael is a passionate PowerShell scripter, speaker, advocate, and streamer, working as a Senior Automation Consultant at Insync Technology[93]. You can follow him on Twitter, @PowerShellMich1[94], or LinkedIn[95]. Michael is a co-founder of the Brisbane Infrastrucure DevOps User Group[96] YouTube channel[97] and author of his livestream on Twitch[98].

[86] mailto:thmsrynr@outlook.com
[87] https://twitter.com/MrThomasRayner
[88] linkedin.com/in/thomasrayner
[89] https://www.linkedin.com/in/sappkevin/
[90] https://codeduet.com
[91] https://adamtheautomator.com/author/kevinsapp/
[92] https://github.com/CodeDuet
[93] https://insynctechnology.com.au
[94] https://twitter.com/PowerShellMich1
[95] https://www.linkedin.com/in/michael-zanatta-61670258/
[96] https://www.meetup.com/Brisbane-PowerShell-User-Group
[97] https://www.youtube.com/channel/UCQfLvFYohCCm_gTPEUfaAbw
[98] https://www.twitch.tv/PowerShellMichael

How to Use This Book

Envisage attending a PowerShell conference, where there are over thirty speakers each presenting a singular forty-five minute session. Normally, some sessions would be run concurrently. However, this conference has organized all presentations to be at different times. How much a would conference like this will cost? The tickets, airfares, accommodation, and meals would all add up to thousands. Thankfully, this conference costs a mere fraction of that as it has been compiled into a book.

This book is designed to be treated like a conference. It is not a textbook, nor is it a reference document that has been created to be used in conjunction with a course, but a book that is meant to be treated as a conference. Each chapter is written by a subject matter expert, with there being no dependency on other chapters. One could start with the first chapter, the last chapter, or somewhere in-between and not miss out on anything that is on offer.

This is a Leanpub "Agile-published" book. All of the work for this book has been completed. However, as issues are reported, supplementary updates may be released. Leanpub will send out an email when the book is updated. These revisions will be available at no extra charge. To provide feedback, use the "Join the Forum" or the "Email the Authors" link on the book's Leanpub web page[99]. Whether it's a code error, a typo, or a request for clarification, our editors will review your feedback, make changes, and re-publish the book. Unlike the traditional paper publishing process, your feedback can have an immediate effect.

About OnRamp

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

Through fundraising and corporate sponsorships, The DevOps Collective, Inc.[100] will be offering several full-ride scholarships to the OnRamp track at the PowerShell + DevOps Global Summit.

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

[99] https://leanpub.com/psconfbook3
[100] https://devopscollective.org/

More information about the OnRamp track[101] at the PowerShell + DevOps Global Summit and their scholarship program[102] can be found on the PowerShell.org[103] website.

See the DevOps Collective Scholarships cause[104] on Leanpub.com[105] for more books that support the OnRamp scholarship program.

Prerequisites

Prior experience with PowerShell is highly recommended. This book is written for intermediate to advanced audiences 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 line that is too long—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* unavoidable, splatting[106] will be used instead.

```
$params = @{
    ComputerName = $computer
    Classname = 'Win32_logicalDisk'
    Filter = "drivetype=3"
    property = 'DeviceID','Size','FreeSpace'
}

Get-CimInstance @params
```

[101] https://powershell.org/summit/summit-onramp/
[102] https://powershell.org/summit/summit-onramp/onramp-scholarship/
[103] https://powershell.org/
[104] https://leanpub.com/causes/devopscollective
[105] https://leanpub.com/
[106] https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_splatting?view=powershell-7

If you are reading this book on a Kindle, tablet or another 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 e-reader might format the page.

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

Feedback

Have a question, comment, or feedback about this book? Please share it via the Leanpub forum dedicated to this book. Once you've purchased this book, log in to Leanpub, and click on the "Join the Forum" link in the Feedback section of this book's webpage[107].

[107] https://leanpub.com/psconfbook3

Acknowledgements

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

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

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

Special thanks to Don Jones for contributing to the Foreword and James Petty for assistance from The DevOps Collective.

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!

I Systems Management

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

1. Contain Yourself - Long-Running PowerShell Scripts in Containers

by Justin P. Sider

This chapter focuses on using a PowerShell container as a runtime environment for long-running PowerShell scripts. It covers the basics of running a Linux-based PowerShell container, setting up the container environment, and executing the long-running scripts.

Creating a consistent runtime environment is essential for executing long-running PowerShell scripts. A container is a lightweight, standalone, executable software package that includes everything needed to run an application: code, runtime, system tools, system libraries, and settings. Utilizing a container creates a portable runtime environment for long-running scripts that can run on Windows and Linux-based hosts.

Container environments isolate long-running scripts, allowing multiple containers to run on the same host. This process isolation allows for running different versions of the script or testing different PowerShell modules and dependencies.

This chapter assumes you are already at a moderate skill level with PowerShell and have a basic knowledge of Docker containers[1]. The below commands require a Windows 10 operating system and a working internet connection.

Getting Started

Installing Docker Desktop

To get started, install the Windows version of Docker Desktop Community Edition[2]. For this chapter, please ensure that Docker Desktop is running in the legacy Hyper-V mode. Full command line documentation for Docker Desktop is available on the Docker website[3].

Downloading a Container Image

To run a container, you must first download the container image to the host machine.

From the host machine, open PowerShell and enter the following command to download the PowerShell container image from Docker Hub[4] (no account is required):

[1] https://www.docker.com/resources/what-container
[2] https://hub.docker.com/editions/community/docker-ce-desktop-windows
[3] https://docs.docker.com/engine/reference/commandline/cli/
[4] https://hub.docker.com/

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

Output:

```
Using default tag: latest
latest: Pulling from powershell
23884877105a: Pull complete
bc38caa0f5b9: Pull complete
2910811b6c42: Pull complete
36505266dcc6: Pull complete
f43b8f1114bc: Pull complete
Digest: sha256:9a141117590e8c7cad2abf42abee5c7a21e466679525c87d7a2455ad7d37e514
Status: Downloaded newer image for mcr.microsoft.com/powershell:latest
mcr.microsoft.com/powershell:latest
```

If you have already pulled the PowerShell container previously, some of your notifications may have "Already exists" as their status. Enter the following command to take a look at the container images on the host machine:

```
docker images --format "table {{.Repository}}\t{{.ID}}"
```

Output:

```
REPOSITORY                      IMAGE ID
mcr.microsoft.com/powershell    f544cbdcb00a
```

Technically, it is not required to pull an image before running a container. Docker will automatically pull an image when you execute the "docker run" command. However, it is a good practice to download an image before starting it with the run command.

Running a PowerShell Container in Interactive Mode

It is now time to run a container to execute commands manually. Enter the following command in the PowerShell console:

```
docker run -it mcr.microsoft.com/powershell
```

You will see that the prompt changes, and the console displays some information.

Output:

```
PowerShell 7.0.2
Copyright (c) Microsoft Corporation. All rights reserved.

https://aka.ms/powershell
Type 'help' to get help.

PS />
```

At this console, you can run PowerShell commands as if it were running the host machine. For example, to list the running processes, execute the following command:

```
Get-Process | Select-Object ProcessName
```

Output:

```
ProcessName
-----------
pwsh
```

Notice that the output only returns a single item. Containers provide a minimal environment to execute the software they need, nothing more.

Environment Variables

Environment variables are generally system-wide settings that can be used by applications running on an operating system. These system-wide settings are critical when executing long-running scripts in standard operating systems or containers. Environment variables can be added to the system at startup or altered during runtime. List the default environment variables inside the container:

```
Get-ChildItem -Path Env: | Sort-Object -Property Name
```

Output:

```
Name                           Value
----                           -----
DOTNET_SYSTEM_GLOBALIZATION_I… false
HOME                           /root
HOSTNAME                       a56453c89ef3
LANG                           en_US.UTF-8
LC_ALL                         en_US.UTF-8
PATH                           /opt/microsoft/powershell/7:...
POWERSHELL_DISTRIBUTION_CHANN… PSDocker-Ubuntu-18.04
PSModuleAnalysisCachePath      /var/cache/microsoft/powersh...
PSModulePath                   /root/.local/share/powershel...
TERM                           xterm
```

Now exit the container. To do this, type the word “exit” in the prompt, then press the “Enter” key. The container will exit and return to the initial PowerShell prompt on the host machine.

Add an Environment Variable

To add an environment variable to a container at startup, use the "–env" parameter as a key-value pair.

```
docker run -it --env GolfCourse=PebbleBeach mcr.microsoft.com/powershell
```

Once the container starts, list the environment variables:

```
Get-ChildItem -Path Env: | Sort-Object -Property Name
```

Output:

```
Name                           Value
----                           -----
DOTNET_SYSTEM_GLOBALIZATION_I… false
GolfCourse                     PebbleBeach
HOME                           /root
HOSTNAME                       a56453c89ef3
LANG                           en_US.UTF-8
LC_ALL                         en_US.UTF-8
PATH                           /opt/microsoft/powershell/7:...
POWERSHELL_DISTRIBUTION_CHANN… PSDocker-Ubuntu-18.04
PSModuleAnalysisCachePath      /var/cache/microsoft/powersh...
PSModulePath                   /root/.local/share/powershel...
TERM                           xterm
```

List the value of the `GolfCourse` variable:

```
$env:GolfCourse
```

Output:

```
PebbleBeach
```

Exit the container just like before by typing "exit" and pressing the "Enter" key.

Digging Deeper into the Docker CLI

Naming a Container

Start a new container. This time, use the "–name" parameter to give it a friendly name.

```
docker run -it --name mypwsh mcr.microsoft.com/powershell
```

While the container is still running, open a second PowerShell console window. To display the running containers, execute the following command:

```
docker ps --format 'table {{.Image}}\t{{.Status}}\t{{.Names}}'
```

Output:

```
IMAGE                            STATUS              NAMES
mcr.microsoft.com/powershell     Up 2 seconds        mypwsh
```

Friendly names are easier to remember when multiple containers are running.

Obtaining the IP Address of a Running Container

Obtain the IP Address of the container by executing the following commands:

```
$format = '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}'
docker inspect --format=$format mypwsh
```

Output:

```
172.17.0.2
```

It is possible based on your Docker configuration that your IP Address could be different.

Return to the PowerShell console with the container prompt and exit the container. Remember, to do this, type the word “exit” in the prompt, then press the “Enter” key.

Cleaning up old containers

Sadly, Docker Desktop does not do a great job of cleaning up after itself. To view all containers run the following command:

```
docker ps -a --format 'table {{.Image}}\t{{.Status}}\t{{.Names}}'
```

To delete all old containers run the following command:

```
docker rm $(docker ps -a -q)
```

Output:

```
dfc7099d33cf
```

The output of the "docker rm" command will differ based on the container id and the number of containers that you exited.

Setting up a Local Test Environment

For the rest of this chapter, the environment will replicate a Client-Server lab environment that includes a vCenter Simulator container and a separate PowerShell container to execute the long-running script. You can find documentation for the vCenter Simulator on GitHub[5]. Do not modify vCenter Simulator before starting the container.

Download the vCenter Simulator container image

Run the following command to download the vCenter Simulator image, created by William Lam, from Docker Hub:

```
docker pull lamw/govcsim
```

Start the vCenter Simulator image and use "vcenter" as the container name:

```
docker run --rm --name vcenter -d -p 443:443 lamw/govcsim
```

Verify that the container is running:

```
docker ps -a --format 'table {{.Image}}\t{{.Status}}\t{{.Names}}\t{{.Ports}}'
```

Output:

```
IMAGE               STATUS              NAMES       PORTS
lamw/govcsim        Up 2 minutes        vcenter     0.0.0.0:443->443/tcp
```

Get the IP Address of the vCenter Simulator container.

```
$format = '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}'
docker inspect --format=$format vcenter
```

Output:

[5] https://github.com/vmware/govmomi/blob/master/vcsim/README.md

```
172.17.0.2
```

Interacting with the vCenter Simulator Container

In the same PowerShell Console window, start a new PowerShell Container:

```
docker run -it --name mypwsh mcr.microsoft.com/powershell
```

Once it starts, install the VMware.PowerCLI Module:

```
Install-Module -Name VMware.PowerCLI -Scope AllUsers -Force
```

Next, set the PowerCLI Configuration, which will allow PowerCLI to connect to a server with an invalid certificate.

```
$Parameters = @{
    InvalidCertificateAction='Ignore'
    Scope='AllUsers'
    ParticipateInCeip=$false
    Confirm=$false
}
Set-PowerCLIConfiguration @Parameters
```

Connect to the vCenter Simulator

Connect to the vCenter Simulator with PowerCLI:

```
Connect-VIServer -Server 172.17.0.2 -User u -Password p -Port 443
```

Output:

```
Name                           Port  User
----                           ----  ----
172.17.0.2                     443   u
```

With the vCenter Simulator container, any username and password combination will allow you to connect. These fields cannot be left blank.

Play Around with VIObjects

List the ESXi hosts:

```
Get-VMHost | Select-Object Name,ConnectionState,PowerState
```

Output:

```
Name       ConnectionState PowerState
----       --------------- ----------
DC0_H0           Connected  PoweredOn
DC0_C0_H0        Connected  PoweredOn
DC0_C0_H1        Connected  PoweredOn
DC0_C0_H2        Connected  PoweredOn
```

List the virtual machines:

```
Get-VM
```

Output:

```
Name                 PowerState Num CPUs MemoryGB
----                 ---------- -------- --------
DC0_H0_VM0           PoweredOn  1        0.031
DC0_H0_VM1           PoweredOn  1        0.031
DC0_C0_RP0_VM0       PoweredOn  1        0.031
DC0_C0_RP0_VM1       PoweredOn  1        0.031
```

Power a virtual machine off:

```
Get-VM -Name DC0_H0_VM0 | Stop-VM -Confirm:$false
```

Output:

```
Name                 PowerState Num CPUs MemoryGB
----                 ---------- -------- --------
DC0_H0_VM0           PoweredOff 1        0.031
```

Place an ESXi host in maintenance mode:

```
Get-VMHost DC0_C0_H1 | Set-VMHost -State Maintenance |
   Select-Object Name,ConnectionState,PowerState
```

Output:

```
Name                 ConnectionState PowerState
----                 --------------- ----------
DC0_C0_H1            Maintenance     PoweredOn
```

Exit the PowerShell container. To do this, type the word “exit” in the prompt, then press the “Enter” key. This will stop the PowerShell container. However, the vCenter Simulator is still running.

Stopping Non-Interactive Containers

To stop a running container when it is not in the interactive mode, use the following command:

```
docker kill vcenter
```

Output:

```
vcenter
```

Verify that the container is not running:

```
docker ps -a --format 'table {{.Image}}\t{{.Status}}\t{{.Names}}\t{{.Ports}}'
```

Output:

```
IMAGE               STATUS              NAMES               PORTS
```

The kill command can use the “Name” or “Container ID” to kill the running container process.

Check Point

At this point, it is clear that running PowerShell in a container is very similar to executing commands on a host machine. This chapter has covered how to download, run, stop, list, and clean up old containers. The next sections will focus on building a custom PowerShell container to execute a long-running script. Before moving forward, please kill all running containers and clean them up.

Start the vCenter Simulator

For the rest of the chapter, you will interact with the vCenter Simulator container. Start the vCenter Simulator container with the following command:

```
docker run --rm --name vcenter -d -p 443:443 lamw/govcsim
```

Verify that the container is running:

```
docker ps -a --format 'table {{.Image}}\t{{.Status}}\t{{.Names}}\t{{.Ports}}'
```

Output:

```
IMAGE               STATUS              NAMES       PORTS
lamw/govcsim        Up 2 minutes        vcenter     0.0.0.0:443->443/tcp
```

Get the IP Address of the vCenter Simulator container.

```
$format = '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}'
docker inspect --format=$format vcenter
```

Output:

```
172.17.0.2
```

Building a Custom PowerShell Container Image

The process of building a custom container image is straightforward. It is best to create a new directory that will include any build artifacts that you require. A `Dockerfile` is also required. This file provides the information Docker Desktop needs to build the container image.

Create a directory

Open a new PowerShell console window and create a new directory.

```
New-Item -Path 'C:\psconf3' -ItemType Directory
```

Output:

```
    Directory: C:\


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        6/17/2020  12:50 PM                psconf3
```

Set the console location to this new directory.

```
Set-Location -Path 'C:\psconf3\'
```

Create a Long-Running Script File

Create a new script file where you will save the long-running script code.

```
New-Item -Path . -Name 'myscript.ps1' -Type File
```

Output:

```
    Directory: C:\psconf3


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        6/17/2020  12:54 PM              0 myscript.ps1
```

Open this file in a text editor. Included below is a long-running script that will update the status of an ESXi host to “maintenance” and then reset its status back to “connected.” Add the following code to the “myscript.ps1” file and save the file.

```
# My Long-Running Script

# Connect to the vCenter Simulator
$Parameters = @{
    Server = "$env:vCenter"
    User = "$env:vcUser"
    Password = "$env:vcPass"
    Port = '443'
}
# Update the PowerCLI Profile
$PowerCLIProfile = @{
    InvalidCertificateAction = 'Ignore'
    ParticipateInCeip = $false
    Scope = 'AllUsers'
    Confirm = $false
}

'Setting PowerCLI Profile Configuration'
Set-PowerCLIConfiguration @PowerCLIProfile

"Connecting to vCenter: $env:vCenter"
Connect-VIServer @Parameters
'Starting Host Maintenance Loop.'
```

```
Do {
    # Find any VMHosts in Maintenance mode
    $VMHosts = Get-VMHost | Where-Object {$_.ConnectionState -eq 'Maintenance'}

    # Remove the Host from maintenance mode
    Foreach ($VMHost in $VMHosts){
        'Setting host back to Connected:'
        Set-VMHost -VMHost $VMhost -State Connected
        $VMHost.Name
    }

    # List the Current state of all VMHosts
    Get-VMHost | Select-Object Name,ConnectionState,PowerState


    'Waiting 10 Seconds'
    Start-Sleep -Seconds 10
    # Find any VMhosts with no VM's and place them in Maintenance mode
    $VMHosts = Get-VMHost
    Foreach ($VMHost in $VMHosts) {
        $VMs = Get-VMHost -Name $VMHost | Get-VM | Measure-Object
        if ($VMs.Count -eq 0){
            'Setting VMHost to Maintenance:'
            $VMHost.Name
            Set-VMHost -VMHost $VMhost -State Maintenance
        }
    }
    # List the Current state of all VMHosts
    Get-VMHost | Select-Object Name,ConnectionState,PowerState

    'Waiting 30 Seconds.'
    Start-Sleep -Seconds 30
} while ($true)
```

This Script can be found in the PSConfBookExtras Repository on Github[6].

Create a DockerFile

This Dockerfile will be used to create a custom container image.

[6] https://github.com/devops-collective-inc/PSConfBookExtras/tree/main/volume-03/justin-sider-contain-yourself/LongRunningScript.ps1

```
New-Item -Path . -Name DockerFile -Type File
```

Output:

```
    Directory: C:\psconf3


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----       6/17/2020   6:50 PM              0 DockerFile
```

Open the DockerFile in a text editor. Add the following code and save the file:

```
# Indicates the base image.
FROM mcr.microsoft.com/powershell

# Install the VMware.PowerCLI Module
RUN pwsh -Command "Install-Module VMware.PowerCLI -Force -Confirm:0;"

# Add the Long-Running Script to the Image in the /tmp directory
COPY myscript.ps1 /tmp
```

Build a Custom Image

Run the below “docker build” command to build a custom image. Notice the “-t” option, which tags the image with a friendly name.

```
docker build . -t myimage
```

Output:

```
Step 1/3:FROM mcr.microsoft.com/powershell
 ---> f544cbdcb00a
Step 2/3:RUN pwsh -Command "Install-Module VMware.PowerCLI -Force -Confirm:0;"
 ---> Running in 5c6f6b1d38fc
 ---> d66405ec96b4
Step 3/3 : COPY myscript.ps1 /tmp
 ---> 445fd73d4c23
Successfully built 445fd73d4c23
Successfully tagged myimage:latest
```

This chapter truncates the output of the build command.

You must be in the directory where the DockerFile exists or specify the full path to the directory with the DockerFile.

After the build is complete, view container images on the host machine:

```
docker images --format 'table {{.Repository}}\t{{.ID}}'
```

Output:

```
REPOSITORY                      IMAGE ID
myimage                         445fd73d4c23
mcr.microsoft.com/powershell    f544cbdcb00a
lamw/govcsim                    470f3123955e
```

The successfully built container image "myimage/445fd73d4c23" will now show up in the image list. Your Image ID may be different than the above ID.

Running the Custom Container

Now run the new custom image with environment variables.

```
docker run -it --env vCenter=172.17.0.2 --env vcUser=u --env vcPass=p myimage
```

Output:

```
PowerShell 7.0.2
Copyright (c) Microsoft Corporation. All rights reserved.

https://aka.ms/powershell
Type 'help' to get help.

PS />
```

Run the following command to validate that the environment variables made it to the running container:

```
Get-ChildItem -Path Env:* | Sort-Object -Property Name
```

Output:

```
Name                           Value
----                           -----
DOTNET_SYSTEM_GLOBALIZATION_I… false
HOME                           /root
HOSTNAME                       a56453c89ef3
LANG                           en_US.UTF-8
LC_ALL                         en_US.UTF-8
PATH                           /opt/microsoft/powershell/7:...
POWERSHELL_DISTRIBUTION_CHANN… PSDocker-Ubuntu-18.04
PSModuleAnalysisCachePath      /var/cache/microsoft/powersh...
PSModulePath                   /root/.local/share/powershel...
TERM                           xterm
vCenter                        172.17.0.2
vcPass                         p
vcUser                         u
```

Run the Long-Running Script inside the Custom container

Set the location of the prompt to the "/tmp" directory:

```
Set-Location /tmp
```

Execute the "myscript.ps1" long-running script file:

```
./myscript.ps1
```

The output of this script will show a single ESXi host going from a connected state to a maintenance state and back to a connected state, over and over. The script will not end unless it encounters a fatal error. While this script may not be useful in your environment, you can understand the usefulness and simplicity of running a long-running script in a PowerShell container.

Press "Ctrl+C" to end the script. Exit the PowerShell container by typing "exit" and pressing the "Enter" key.

Removing a Custom Image

View container images on host machine:

```
docker images --format "table {{.Repository}}\t{{.ID}}"
```

Output:

```
REPOSITORY                      IMAGE ID
myimage                         445fd73d4c23
mcr.microsoft.com/powershell    6850488c74c6
lamw/govcsim                    470f3123955e
```

Enter the following command in the PowerShell prompt to delete the custom "myimage" container image:

You will need to enter the Image ID for your custom image. Remember to remove the stopped container before trying to remove it. If there is a stopped container from the custom image, you will get an error if you try to remove the custom image.

```
docker rmi 445fd73d4c23 -f
```

Output:

```
Untagged: myimage:latest
Deleted: sha256:2c43c46fe9a...666198af300740d02e
Deleted: sha256:cb5c8e9d519...5c98c570ad341e4765
Deleted: sha256:eff86406b33...720360fe7817911359
Deleted: sha256:d66405ec96b...0cb372e9b7ffbf8d03
Deleted: sha256:ddf197b71b5...ce5476af8fbcff6c91
```

You can use the "Image ID" or the "Repository" name to remove a container image.

Create a Container to AutoRun the Long-Running Script

To have the container automatically start the long-running script, the Dockerfile needs to be modified. The default command for the PowerShell containers is to run "pwsh" rather than a script. Adding a command parameter with a script name will automatically execute the script at the startup of the container.

Overwrite the existing Dockerfile with this code.

```
# Indicates the base image.
FROM mcr.microsoft.com/powershell

# Install the VMware.PowerCLI Module
RUN pwsh -Command "Install-Module VMware.PowerCLI -Force -Confirm:0;"

# Add the Long-Running Script to the Image in the /tmp directory
COPY myscript.ps1 /tmp

# Start the Long-Running Script at Runtime
CMD ["pwsh", "-Command", "/tmp/myscript.ps1"]
```

The CMD line specifies a program and arguments to execute at the container startup.

Build the AutoRun Image

Execute the build command to create the image:

```
docker build . -t auto
```

Output:

```
Step 1/4:FROM mcr.microsoft.com/powershell
 ---> f544cbdcb00a
Step 2/4:RUN pwsh -Command "Install-Module VMware.PowerCLI -Force -Confirm:0;"
 ---> Running in 5c6f6b1d38fc
 ---> d66405ec96b4
Step 3/4:COPY myscript.ps1 /tmp
 ---> cb5c8e9d519c
Step 4/4:CMD ["pwsh", "-Command", "/tmp/myscript.ps1"]
 ---> Running in 39216a454f3c
Removing intermediate container 39216a454f3c
 ---> 2c43c46fe9a9
Successfully built 2c43c46fe9a9
Successfully tagged myimage:latest
```

This chapter truncates the output of the build command. After the build completes, run the new custom image with environment variables.

```
docker run -it --env vCenter=172.17.0.2 --env vcUser=u --env vcPass=p auto
```

You will notice that the container immediately starts to run the script. It does not open to a PowerShell prompt. Press “Ctrl+C” to exit the script and container. That will return to the host machine PowerShell prompt.

Additional Considerations

This chapter has covered the basics of running PowerShell containers with Docker Desktop on a Windows 10 operating system, concluding with executing a long-running script inside of a container. While this can prove to be a useful tool, consider your requirements before converting all scripts to run in this new environment. Keep in mind this is just another tool to add to your already complex toolbox. Consider researching more about containers, deployment methods, and management tools.

The Pro's

Containers provide an isolated, clean environment for scripts to run. The runtime environment is portable to any host capable of running containers, meaning this solution is platform-independent. The containers can be shared within a team utilizing a shared repository, allowing all users to run identical scripts in duplicate runtime environments.

The Con's

While the setup for containers is relatively simple, it can become overwhelming for novice users to walk into a containerized environment. Using containers potentially creates an additional layer of abstraction that is not needed. Security can play a crucial role in determining the deployment of containers, the passing of secrets, and the storage location of protected data. Logging is complex when running scripts in containers if they need to be stored long term.

Summary

Executing long-running scripts in PowerShell containers is a great way to isolate those processes in a clean and consistent runtime environment. There are many other use cases to use containers as a runtime environment, such as testing PowerShell code, multiple module versions, external service versions, building PowerShell applications, and more. Let this chapter open the door to the beginning of your container journey.

II 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.

2. Save Your Standards from Becoming Rarely Used Checklists; Codify Them

by Josh King

Whether you are talking about manual processes or automation, standards are critical: They're used to ensure everyone is on the same page and help guarantee that common tasks and implementations are consistent.

Imagine the chaos that would ensue if five different technicians deployed a domain controller each, into an existing Active Directory domain without any defined standards. Each server would have a unique flavor. Different firewall ports are open if the firewall is even enabled. Only three of them have your usual suite of support tools stored locally. There is a mix of both Server Core, and Desktop Experience installs. You eventually find out that one technician even deployed a Windows Server 2012 R2 image.

If only you had a standard that dictated precisely how domain controllers were to be deployed and configured in your environment.

The Sad Fate of Most Standards

As crucial as standards are, you inevitably find that they become glorified checklists. If you're lucky, the wider team knows that these standards exist; if you're even luckier, the team follows them.

Unfortunately, teams only forget the standards if those standards are not ingrained into your workplace culture. You may come across a new domain controller that doesn't match the defined standard. When asked, the technician that deployed it reports that they didn't even know that the standard existed.

Other than ongoing *A Clockwork Orange*-style reconditioning sessions, how can you ensure that your team follows the standards?

The answer—much like automation in general—is to remove the human element.

If you can codify your standards, then you can audit that the team is following them. Once you have made that leap, you can then take your codified standards and use them to help automate compliance.

Introducing the Requirements Module

Requirements is a PowerShell Module that is available on the PowerShell Gallery[1]. Its purpose is to be a framework for defining and enforcing system configurations.

Hearing that description, you might rightly wonder if Microsoft has renamed Desired State Configuration[2] (DSC). It hasn't, but there is some overlap between the Requirements module and DSC. The main difference between the two is that DSC is built for massive operations, applying many different configurations across numerous systems at the same time. On the other hand, Requirements applies a single configuration against one system at a time.

If you have already invested in learning DSC, you don't have to throw away what you know to use Requirements. In fact, you can use DSC resources with Requirements, although this means that your Requirements configuration will now depend on DSC's configuration manager.

The Requirements module is open source, and you can contribute to the project by visiting the codebase on GitHub[3].

Requirements Primer

At a fundamental level, you use the Requirements module by defining a collection of, well, "requirements."

You can define these requirements as a hashtable containing three elements:

- **Describe**: The human-readable summary of a given requirement;
- **Test**: Code that determines if the system meets the requirement;
- **Set**: Code that puts a system into a state which satisfies the requirement.

It is possible to leave out the set component if you only want to audit if a system meets your standards. If you happen to know what idempotence is, you can also leave out the test component, but this chapter won't be covering that concept.

The following example illustrates a sample standard represented with the Requirements module:

```
$Requirements = @(
  @{
    Describe = 'Notepad is running'
    Test = {
      $null -ne (Get-Process -Name 'notepad' -ErrorAction SilentlyContinue)
    }
    Set = {
      Start-Process -FilePath 'notepad.exe'
    }
  },
```

[1] https://www.powershellgallery.com/packages/Requirements
[2] https://docs.microsoft.com/en-us/powershell/scripting/dsc/overview/overview
[3] https://github.com/microsoft/Requirements

```
  @{
    Describe = 'The Requirements module is installed'
    Test = {
      $null -ne (Get-Module -Name 'Requirements' -ListAvailable)
    }
    Set = {
      Install-Module -Name 'Requirements' -Force
    }
  }
)
```

This example tests to see if Notepad is running and starts the process if not. It then tests the installation status of the Requirements module itself, which would pass, since it is running the test. For completeness, if the second test fails, the code within the Set element will install the missing module.

If you only want to audit this set of requirements, you would use the `Test-Requirement` function. To make the output from this execution easier to digest, you can pipe the output to `Format-Table`.

```
$Requirements | Test-Requirement | Format-Table State, Result, Requirement
```

```
State Result Requirement
----- ------ -----------
Start        Notepad is running
 Stop False  Notepad is running
Start        The Requirements module is installed
 Stop True   The Requirements module is installed
```

This example output shows that Notepad is not running, indicated by the **False** result. As expected, `Test-Requirement` detected the Requirements module, and there is a corresponding **True** result.

Because this is only testing the requirements, `Test-Requirement` makes no changes to bring the system into compliance.

To enforce these requirements, you will instead use the `Invoke-Requirement` function. You can also use an included formatting function, `Format-Checklist`, to get dynamic output as the requirements are tested and enacted.

```
$Requirements | Invoke-Requirement | Format-Checklist
```

```
√ 12:11:21 Notepad is running
√ 12:11:22 The Requirements module is installed
```

In this output, the √ character indicates that the test has passed. If you run this yourself, you see that the output is also colored. While a given test runs, its line is a neutral color with no icon and is then updated based on the success or failure of each test.

When invoking your set of requirements, `Invoke-Requirement` executes them in order. It tests the first requirement, and if the test fails, the Set code runs followed by the tests being run again to validate that the change was effective. If the test passes in the first instance or the change was successfully validated, then the next requirement in the set begins processing.

`Invoke-Requirement` terminates if validation fails, meaning it stops all output if one item remains out of compliance and won't process additional requirements. Make sure that you order your requirements properly to ensure that any prerequisites for subsequent items are completed and verified before they are needed.

The Lab

To follow along with the demonstration in this chapter, you need a fresh install of Windows Server 2019 Standard. You can run this in a virtual machine (VM) on Hyper-V on your workstation, or in your cloud provider of choice.

If needed, you can download the ISO for this server from the Microsoft Evaluation Center[4].

Aside from setting your administrator password, you also also need to install the Requirements module on your new server.

```
Install-Module -Name Requirements
```

As this is a fresh install, PowerShell prompts you to install the NuGet provider. You're safe to agree to it.

If running Windows Server in a VM, you may want to take a snapshot/checkpoint before carrying out the demonstration. Having a snapshot/checkpoint allows you to quickly roll back changes that are made without needing to undo changes manually or reinstalling the entire operating system.

Demo: From Standards to Requirements

This demonstration takes you from zero to hero—or at least competent—with regards to using the Requirements module. Take note that there is a lot more to this module than what one chapter can cover. The GitHub Repository[5] is an excellent resource if you want to learn more.

[4] https://www.microsoft.com/en-us/evalcenter/evaluate-windows-server-2019
[5] https://github.com/microsoft/Requirements

The Standard

Your company is deploying several instances of Windows Server 2019. There are some essential configuration elements you need to keep consistent across all of these instances, so you define the following standard:

- The default computer name must not be used;
- The time zone must be set to UTC;
- CD/DVD drive must be on drive letter B;
- The C drive must be labeled as "OS";
- The NTFSSecurity PowerShell module[6] should be installed.

Imagine that you have typed this up on the proper template, and management has signed off on it. You have circulated the standard through all proper channels to the technicians that should follow it.

To your shock, you come across several new installs that don't follow—or only partially follow—the standard.

Testing the Standard

Armed with your defined standard, you can start working towards codifying and testing it as a set of requirements. The best way to get started is to focus on each element of your standard individually, addressing only one at a time before moving onto the next.

This demo walks you through building out your requirements, but remember that you group these in a collection, which is the first thing that is specified.

```
$Requirements = @(
  # Include your individual requirements here.
)
```

Remember that you need to put a comma between elements in your `$Requirements` collection. This demo includes them after the closing curly brace, and you should include them if you are replicating these examples directly.

Test: The Default Computer Name Should Not Be Used

Generally, if you wanted to see the name of the computer you are using, you might use an environment variable, `$env:COMPUTERNAME`. While this returns the computer's current computer name, you need to think forward to enforcing the standard. When you change a computer's name, that environment variable doesn't update until after a reboot, but you need to check that your change is effective before that.

Luckily, you can find a computer's name—changed or not—in the registry.

[6] https://www.powershellgallery.com/packages/NTFSSecurity

```
@{
  Describe = "The default computer name should not be used"
  Test = {
    $Reg = "HKLM:\SYSTEM\CurrentControlSet\Control\ComputerName\ComputerName"
    (Get-ItemProperty -Path $Reg -Name "ComputerName") -notmatch 'WIN-[\w]+'
  }
},
```

Notice that the Describe element is copied directly from your standard, as it's already written for human consumption.

The Test element starts by defining where in the registry to find the computer name. Next, it receives the current computer name and checks that it does not match a pattern.

This pattern is a Regular Expression (RegEx), and it looks for a string that starts with "WIN-" followed by one or more numbers or letters. This happens to match the automatically generated names used when installing Windows Server.

Test: The Time Zone Should Be Set to UTC

Time zones are a more straight forward setting than the computer name.

You can run `Get-TimeZone` before and after changing a system's time zone setting and get the correct information right away.

```
@{
  Describe = "The time zone should be set to UTC"
  Test = { (Get-TimeZone).Id -eq 'UTC' }
},
```

This time you are checking to see if the current time zone's ID is equal to 'UTC'. This logic is similar to an if statement, and results in either a True or False being returned.

Test: CD/DVD Drive Should Be on Drive Letter B

Depending on how you deploy your virtual servers, you may find that they don't have any optical drives. With that in mind, this test needs to succeed if there is no drive to be assigned the desired drive letter.

```
@{
  Describe = "CD/DVD drive should be on drive letter B"
  Test = {
    $Drive = Get-CimInstance -ClassName Win32_Volume -Filter 'DriveType = 5'
    $null -eq $Drive -or $Drive.DriveLetter -eq 'B:'
  }
},
```

Here, you retrieve the optical disk by selecting anything volumes with a drive type of 5 (Compact Disk). When testing this, you first check to see if the drive exists by comparing it against null. If there is no optical disk, the test passes.

Assuming there is a disk, you then check to see if its drive letter is equal to "B:" and the test fails if it isn't.

Test: The C Drive Should Be Labeled as "OS"

You've already run tests on one type of disk, and the test for the C drive is very similar. This time you get all disks assigned the drive letter C and then check whether its label is equal to "OS."

```
@{
  Describe = "The C drive should be labeled as 'OS'"
  Test = {
    $C = Get-CimInstance -ClassName Win32_Volume -Filter "DriveLetter = 'C:'"
    $C.Label -eq 'OS'
  }
},
```

Test: The NTFSSecurity PowerShell Module Should Be Installed

Your final test is regarding a PowerShell module that you want to ensure is available on all of your servers.

```
@{
  Describe = "The NTFSSecurity PowerShell module should be installed"
  Test = {
    $null -ne (Get-Module -Name NTFSSecurity -ListAvailable)
  }
}
```

To perform this test, you run `Get-Module` and specify the desired module by name. You must include the `-ListAvailable` switch so that you get the needed output even if PowerShell has not loaded the module into the current session.

You then check that output against `$null` using the "not equals" operator, meaning that you got **something** from `Get-Module`.

Running The Test

With all of your requirements written, add them all to your `$Requirements` collection and then pass them through `Test-Requirement`.

```
$Requirements = @(
  # .. snip ..
)

$Requirements | Test-Requirement | Format-Table State, Result, Requirement
```

On a fresh Windows Server 2019 install, the output won't show much alignment with the standard.

```
State Result Requirement
----- ------ -----------
Start        The default computer name should not be used
 Stop False  The default computer name should not be used
Start        The time zone should be set to UTC
 Stop False  The time zone should be set to UTC
Start        CD/DVD drive should be on drive letter B
 Stop False  CD/DVD drive should be on drive letter B
Start        The C drive should be labeled as 'OS'
 Stop False  The C drive should be labeled as 'OS'
Start        The NTFSSecurity PowerShell module should be installed
 Stop False  The NTFSSecurity PowerShell module should be installed
```

Every test did not pass, meaning that each of your requirements is not yet satisfied.

Enforcing the Standard

Testing your standards is only half the battle. You could call your automation journey quits here and try to use the failing tests to get technicians to revisit their work. The better option is to build on your current requirements so that they can bring systems into compliance automatically.

This section follows the same format as the previous one on testing, including the same test elements that you saw previously.

Set: The Default Computer Name Should Not Be Used

The main problem when changing away from the default computer name assigned to your server is deciding what the new name should be.

If you have a particular naming scheme already, you should find some way of representing it in code. For this example, you're changing the server name to LAB- followed by eight characters from a randomly generated globally unique identifier (GUID).

A potential new computer name is *LAB-FD5C436F*.

```
@{
  Describe = "The default computer name should not be used"
  Test = {
    $Reg = "HKLM:\SYSTEM\CurrentControlSet\Control\ComputerName\ComputerName"
    (Get-ItemProperty -Path $Reg -Name "ComputerName") -notmatch 'WIN-[\w]+'
  }
  Set = {
    $NewName = "LAB-$(((New-Guid).Guid.split('-')[1,2] -join '').ToUpper())"
    Rename-Computer -NewName $NewName -Force
  }
},
```

This new name is applied using the `Rename-Computer` cmdlet, and you supply the `-Force` switch so that PowerShell does not prompt you to confirm this change.

Set: The Time Zone Should Be Set to UTC

Changing the time zone is instantaneous. If you watch the desktop clock on your server when `Invoke-Requirement` enforces these requirements, you may even catch the time change.

```
@{
  Describe = "The time zone should be set to UTC"
  Test = { (Get-TimeZone).Id -eq 'UTC' }
  Set = { Set-TimeZone -Id 'UTC' }
},
```

To enact the change, you call `Set-TimeZone` and specify the desired time zone.

Set: CD/DVD Drive Should Be on Drive Letter B

When testing for optical drives, remember that you stored any available drives in a variable. This variable is only available in the Test element and doesn't flow through to the Set element. Not having this variable accessible means that you need to find your optical drive again rather than being able to reuse the existing variable.

```
@{
  Describe = "CD/DVD drive should be on drive letter B"
  Test = {
    $Drive = Get-CimInstance -ClassName Win32_Volume -Filter 'DriveType = 5'
    $null -eq $Drive -or $Drive.DriveLetter -eq 'B:'
  }
  Set = {
    $Drive = Get-CimInstance -ClassName Win32_Volume -Filter 'DriveType = 5'
    Set-CimInstance -InputObject $Drive -Property @{DriveLetter='B:'}
  }
},
```

The syntax for changing the drive letter involves passing your `$Drive` variable to `Set-CimInstance`, along with a hashtable containing any properties you want to change. For this requirement, we're only changing the DriveLetter property, so that is the only property specified in the hashtable.

Set: The C Drive Should Be Labeled as "OS"

Like the previous Set element, you need to find your C drive again to make changes to it. Here you use `Set-CimInstance` again, but specify Label in the property hashtable, as that's the only thing that needs to be changed.

```
@{
  Describe = "The C drive should be labeled as 'OS'"
  Test = {
    $C = Get-CimInstance -ClassName Win32_Volume -Filter "DriveLetter = 'C:'"
    $C.Label -eq 'OS'
  }
  Set = {
    $C = Get-CimInstance -ClassName Win32_Volume -Filter "DriveLetter = 'C:'"
    Set-CimInstance -InputObject $C -Property @{Label='OS'}
  }
},
```

Set: The NTFSSecurity PowerShell Module Should Be Installed

Finally, you need to make sure that technicians have installed the desired PowerShell module on the server.

To do this, you call `Install-Module` like you would when installing modules on your own computer. You use the `-Force` switch is to install the module without needing manual input.

```
@{
  Describe = "The NTFSSecurity PowerShell module should be installed"
  Test = {
    $null -ne (Get-Module -Name NTFSSecurity -ListAvailable)
  }
  Set = {
    Install-Module -Name 'NTFSSecurity' -Force
  }
}
```

Applying Requirements

With your definitions for the Requirements module now fully formed, it is time to use them to bring your server into compliance with your standard. Your final `$Requirements` collection looks like this:

```
$Requirements = @(
@{
  Describe = "The default computer name should not be used"
  Test = {
    $Reg = "HKLM:\SYSTEM\CurrentControlSet\Control\ComputerName\ComputerName"
    (Get-ItemProperty -Path $Reg -Name "ComputerName") -notmatch 'WIN-[\w]+'
  }
  Set = {
    $NewName = "LAB-$(((New-Guid).Guid.split('-')[1,2] -join '').ToUpper())"
    Rename-Computer -NewName $NewName -Force
  }
},
@{
  Describe = "The time zone should be set to UTC"
  Test = { (Get-TimeZone).Id -eq 'UTC' }
  Set = { Set-TimeZone -Id 'UTC' }
},
@{
  Describe = "CD/DVD drive should be on drive letter B"
  Test = {
    $Drive = Get-CimInstance -ClassName Win32_Volume -Filter 'DriveType = 5'
    $null -eq $Drive -or $Drive.DriveLetter -eq 'B:'
  }
  Set = {
    $Drive = Get-CimInstance -ClassName Win32_Volume -Filter 'DriveType = 5'
    Set-CimInstance -InputObject $Drive -Property @{DriveLetter='B:'}
  }
},
@{
  Describe = "The C drive should be labeled as 'OS'"
  Test = {
    $C = Get-CimInstance -ClassName Win32_Volume -Filter "DriveLetter = 'C:'"
    $C.Label -eq 'OS'
  }
  Set = {
    $C = Get-CimInstance -ClassName Win32_Volume -Filter "DriveLetter = 'C:'"
    Set-CimInstance -InputObject $C -Property @{Label='OS'}
  }
},
@{
  Describe = "The NTFSSecurity PowerShell module should be installed"
  Test = {
    $null -ne (Get-Module -Name NTFSSecurity -ListAvailable)
  }
  Set = {
    Install-Module -Name 'NTFSSecurity' -Force
```

```
  }
}
)
```

Now you can enforce your standard by passing this collection to `Invoke-Requirement` and formatting the output using `Format-Checklist`.

Before executing this, you may want to change your `$WarningPreference`, as you get a warning about needing to restart the server for the new name to take effect. Typically this is fine, but a warning disrupts the checklist output.

```
$WarningPreference = 'SilentlyContinue'
$Requirements | Invoke-Requirement | Format-Checklist
```

```
√ 10:23:55 The default computer name should not be used
√ 10:23:57 The time zone should be set to UTC
√ 10:24:00 CD/DVD drive should be on drive letter B
√ 10:24:02 The C drive should be labeled as 'OS'
√ 10:24:21 The NTFSSecurity PowerShell module should be installed
```

Congratulations, your server is now entirely completely in compliance with your defined standard!

Wrap Up

The demonstration in this chapter involved running your requirements directly on the target server. If you want to scale this out a little, you could consider running it from your Remote Monitoring & Management tool, or your automation orchestrator of choice. You could even build this into your image so that it executes on the first login.

While Requirements is a potent tool, at larger scales, you should consider implementing PowerShell DSC instead.

If you're hooked and want a challenge to test your knowledge of the Requirements module, try the following. Create a set of requirements that set up your workstation with the software you need and any personalizations you always make. The result gives you a repeatable tool you can use to speed up migrating to a new computer.

Now, go and save your essential standards from being forgotten forever!

III DevOps

This section highlights DevOps patterns and practices, including CI/CD and Infrastructure as Code.

3. Git for Admins That Don’t Get Git

by Jeremy I Brown

My Story

I work in an operations group that doesn’t write much code. Most of my colleagues are the “old dogs” still clicking away in their GUIs. These “old dogs” spend a lot of time working serially in wizards to accomplish the same thing that most people within the PowerShell community know can be done more efficiently with PowerShell. Within this GUI-centric working environment, I have spent the past couple years trying to foster a culture that embraces more Infrastructure-as-Code and less work in the GUI. One of my biggest hurdles was getting the existing organizational code captured in a single place, where everyone in my group could read and learn from each other. Currently, each administrator had unique pieces of code, their unique workflows, and their unique storage locations. This fragmented environment posed a big part of the problem. No one can learn from each other when everyone stays siloed in their unique processes and workflows. I knew getting all the code for these different service stacks into a single place would be a considerable benefit to everyone overall, but the question was how. I believe many other administrators fall into this same category of wanting to centralize code. This solution described in this chapter has worked wonders for our group. It helped those that don’t write much code follow a “Best Practices” model. It has also sparked conversations and training opportunities with other users that do write more code. This sharing and collaboration provide an excellent opportunity to learn git workflows without a heavy burden.

Intro

This chapter will walk you through how to automate moving code into a Git repository. This process will allow users that are not familiar with Git workflows to copy code into a shared folder. This shared folder will automatically sync the new and changed files to a remote Git repository.

The chapter covers the following aspects:

- How to configure a service account on a local file server.
- How to enable the service account as a Git account.
- How to create an SSH key.
- How to deploy the SSH key to GitHub.

- How to push code from the shared folder to a remote Git repository.
- How to create a scheduled task to automate the push.

In this chapter, there is no expectation of any Git experience; however, reading this chapter can be used as a stepping stone.

> The following content should be considered a Proof-of-Concept. Some of these steps are not suitable for syncing production code. The information in this chapter is a starting point and will give you information, so you can build a solution that works for your organization.

Step 1: Create Accounts

Local Service Account

You will need a service account to run the scheduled task on the local development server. For this example, the service account will be named `svc.gitSync`. This account does not need any special permissions on the development server. A basic user account is sufficient. The service account requires the ability to read and write to the shared folder. When selecting the shared folder's location, avoid a folder within a user profile, and make sure normal users have the appropriate access to this folder.

The following PowerShell code creates this service account.

```
$Splat = @{
    Name = 'svc.gitSync'
    Password = (ConvertTo-SecureString -String 'P@ssw0rd' -AsPlainText -Force)
    Description = 'Service Acct syncing C:\GitSync to GitHub'
}
New-LocalUser @Splat
```

> The code above contains a hash table `$Splat`. Splatting saves the parameters for a command into a single hashtable. You specify the name of the service account, the initial password, and the description. After you create the `$Splat`, you expand all those parameters using '@' and the variable name. You can add as much to the Splat as you wish. For example, you can omit the FullName property, or you could choose to provide a friendlier display name, if you wish, by adding that parameter to the Splat.

Email and GitHub Accounts

Next, create an email account for use by the `svc.GitSync` account to connect to GitHub. Any email account will work. I used Gmail for this demo and created `svc.gitSync@gmail.com`. After you set up the email account, use that email address to create a GitHub account. This part will be different depending on the situation.

> If you have an internal Git service, like GitHub Enterprise, GitLab, or Bitbucket, you should contact the service owner for assistance. The service owner will guide you to the best identity to use for connecting to your organization's source control service.

Step 2: Install Git

> If you already have Git installed on your local development machine. You can skip to Step 3.

Git is a set of binaries that anyone will use to interact with versioning and source control. Download the Git binaries from the Git website[1]. After download, install the software as you would any other. Another option is to use a package manager, such as Chocolatey.

If you have Chocolatey installed, run the following to install Git.

```
choco install git
```

When installing Git, make sure to choose "Git from the command line and also from 3rd-party software" when you reach the "Adjusting your PATH environment" part of the install wizard. This choice is important because you won't be able to leverage PowerShell in the scheduled task if you do not choose it.

Step 3: Configure Git and SSH

Configure Git

At this point, it is now possible to create a folder, initialize it as a Git repository, and begin using Git functions; however, there is no ability to push data to a remote location. To complete post-installation tasks and allow an account to commit to a remote repository, identity information and SSH keys need to be specified and created. The SSH key will be in the user's local profile

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

directory. If you choose to move the key to another location, you will have to do some extra work to tell Git where the SSH key is.

Do not store the key in a public location on the computer.

After the initial Git installation, there will be a small, generic list of data configured. You can view those settings by issuing `git config --list`. This command shows you the currently configured settings. Notice that `user.name` and `user.email` are not listed. To ensure that the service account will submit code as it's identity, issue the following commands:

```
git config user.email --global "svc.gitSync@gmail.com"
git config user.name --global "svc.gitSync"
```

> If you have previously configured Git for another user and set these values using the `--global` option, they are populated as a different identity on the machine.

These commands will set the name and email address for this service account globally. All repositories created on this server will default to using the specified username and email. If a single repository needs to use a different identity, issue the above commands without `--global`.

If you get an error stating "fatal: not in a git directory," you will need to wait to override the global username and email. This error stems from the fact that you are not setting a global property, but only a property in a specific repository. That requires there be an initialized Git repository, and you must be in that directory.

Configure SSH

The identity data created in Step 3 must exist to create an SSH key. For the demonstration, your best option is to use the `--global` option mentioned above.

To begin creating an SSH key, execute the following command on the development server:

```
ssh-keygen -t rsa -b 4096 -C "svc.gitSync@gmail.com"
```

Once executed, there will be a set of prompts that require a response. The first prompt is where to store your public/private key pair. Keep the public key file in a location where only the service account has access. For this example, the default location is sufficient. The next prompt is your password prompt. You need to make sure you **DO NOT** set a password.

> This is where you must assume some risk. If you set a password, then when the service account executes `git push`, the password will be required to be entered interactively. This demo uses a non-interactive scheduled task, and the service account will use an SSH key instead of a password, so a password is not needed. It is possible to include the username and password to `git push`; however, that is outside the scope of this chapter.

Step 4: Create a GitHub Repo

Once the GitHub account is created and signed in, click "Create Repository" and make a new repository.

For the demo, leave the repository as a Public repository.

You shouldn't put production code in a public repository. This demo is a Proof-of-Concept and not considered production-ready. Please only copy data that is safe to make publicly accessible.

On the local development server, log in as the service account. This initial configuration requires an interactive session. There are a few prompts on the first `git push` that require the service account to respond interactively. First, create a local folder that will sync to the GitHub repository previously created. For this demo, name it `GitCronSync`. Create the directory with this command:

```
New-Item -Type Directory -Path 'C:\git\GitCronSync'
```

This folder will be the destination for all users to save their scripts. Next, retrieve the SSH public key. If you used the default location, then the SSH key will be at:

```
"C:\Users\<UserName>\.ssh\id_rsa.pub"
```

Open the `id_rsa.pub` file with a text editor and copy the data. The SSH key should look something like this:

```
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAACAQDpENPa4n90/sXVZ9TkL9r6GhFHLcVvESoctn2COY2
NbSU3D8JT5g4sCvMyTtD1/Z7/noJT0QCv1RLyCmbYa2/kZfPXZoIyeYK1nb0+A5d1ZSW5QmFLheo2HD
p5y/n10/m/A7I3PODQLyNby3XifE6EDKVByKI6W0e3Koc7Q0EQ6Q7WikB4kMhIFI4tBrgQbmqBj569R
ROPFlOR2v2Qfu0BYVg3Od5Fcimf3xPhzeEuGPKTfj7F9WyoD5kSV1cijUoWQlIoRfLi49GAcupMYr98
QvogUPznJxTcyIWmtWVUZOtkaSKCtSawD48BqTJNOXtzPda79ht/LrdAbhx0rthLev/uBlaiNTcAsfq
bmcrSxJOon3XVRR1MG32bylgaAsZFcDCWs0sTcjIWX+kHTVvYrDJRuoTC4USe+G+qUIIoQ/qYsCLjZb
dUii/02i6bSBGjNAByq2iO/OdHrGOCzlcJTgBWv4laK/LQRdPIwWRSqN7gNHx0Aeujpqr3OA0ovvako
taHJa9j9lqKIzTw3biZ0v5kmIIhwhcVcDzOxuaHaqTts+7IJsUNDvp4raJRgzvZAkHDemTdp6IRmYrD
Je1NX0Fu6i8mhuJFORY7e3Yz1SKJThJDUMjs2A22WxKmtalOQD6sj4cwq94ZroP8hWrMF818gfCev/O
8I88K5ynUw== svc.gitSync@gmail.comsvc.gitSync@gmail.com
```

This key is the public side of the service account's identity. The service account will authenticate to GitHub using the private key, and this will allow the service account to push to the repository. With the SSH public key in the clipboard, add the copied public key to the GitHub user profile that you set up previously. Navigate to the settings of the GitHub user. At the time of this chapter's creation, the steps are:

1. In the top right corner, click your user.
2. Click Settings.
3. Select SSH and GPG Keys.
4. Select "New SSH Key."
5. Paste the public key you copied from the text editor, above, into the "key" field and provide a title for this SSH key.
6. Click "Add SSH Key."

Once the SSH key is attached to the user, the service account will be able to push data from the local repository on the development server. There should be a code block example in GitHub, located in the empty repository you just created. Copy and paste it into a PowerShell console after traversing into the shared folder on the local development server. These commands will turn the shared folder on your local development server into a Git repository. That code should look something like this:

```
echo "# GitCronSync" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin git@github.com:svcGitSync/GitCronSync.git
git push -u origin master
```

Upon refreshing the GitHub webpage, a README file should exist.

Step 5: Create the Git Sync Script

There isn't much to the script that will automate the push to the remote repository. The code listed below is not complicated. This script can be as verbose or simple as you choose to make it. Overall, the script does the following:

1. Move into the shared folder that users are copying files in to.
2. Construct a commit message to apply later.
3. Execute `git add .` to select all the current changed files.
4. Execute `git commit -m "$CommitMessage"` to add the required commit message.
5. Execute `git push` to sync all the changes to the GitHub repository, configured above.

A few points to note:

First, this script's commit message grabs the current date and time. You can add as much to commit messages as you want. Perhaps use `git status` to capture the output and use that as the commit message. That would give each commit a list of changed files in the commit message. Let your imagination run wild. Second, capture any file added, removed, or altered in the folder. Any file that has no change will not be selected.

```
Push-Location 'C:\git\GitCronSync'
$CommitMsg = "Commit files added or updated before {0}" -f (Get-Date -Format g)
git add .
git commit -m "$CommitMsg"
git push
```

Don't forget, if this is a new server, you may need to run `Set-ExecutionPolicy Unrestricted`.

Step 6: Create the Scheduled Task

The last part required is to automate the syncing of the code added to the shared folder. To achieve this, use a scheduled task. Create the scheduled task on the development server where the shared folder exists. When creating a scheduled task via PowerShell, the cmdlets are different from most others. You build a scheduled task using different cmdlets to create specific, required objects for the `Register-ScheduledTask` cmdlet. Once those objects exist, use them to create the scheduled task. The code should look something like this:

```
$TaskActionSplat = @{
    Execute = 'PowerShell.exe'
    Argument = '-file C:\scripts\GitCronSync.ps1 -WindowStyle Hidden'
}
$TaskAction = New-ScheduledTaskAction @TaskActionSplat

$TaskTriggerSplat = @{
    At = '1am'
    RepetitionInterval = New-TimeSpan -Minutes 60
    RepetitionDuration = New-TimeSpan -Minutes 60
}
$TaskTrigger =  New-ScheduledTaskTrigger -Once @TaskTriggerSplat

$TaskSettingsSplat = @{
    RestartCount = 3
    RestartInterval = New-TimeSpan -Minutes 5
    ExecutionTimeLimit = New-TimeSpan -Minutes 60
```

```
}
$TaskSettings = New-ScheduledTaskSettingsSet @TaskSettingsSplat

$RegisterTaskSplat = @{
    Action = $TaskAction
    Trigger = $TaskTrigger
    Settings = $TaskSettings
    TaskName = 'GitCronSync'
    Description = 'Commits files from C:\GitCronSync to GitHub repo'
    User = 'GitCronSync'
    Password = 'P@ssw0rd'
}
Register-ScheduledTask @RegisterTaskSplat
```

Looking above, notice you created three different scheduled task objects; a TaskAction, a TaskTrigger, and a TaskSettings object. After you create all the individual parts of the scheduled task, use those objects as parameters to create the scheduled task. This code will configure a scheduled task to run once an hour. The task will stop executing if it runs longer than one hour, and it will attempt to restart three times every 5 minutes if there is a failure to start. The other options in the Splat configure the username and password the task needs to execute as, the name you'll see in Task Scheduler, and a description.

It is not required to pass the username and password via the code.
If omitted from the splat, go to the created scheduled task and edit via the GUI. If done in code, redact the password from any saved scripts.

Conclusion

As with most things, anything worth doing takes some effort. Following the example in this chapter, you have created a service account, a remote repository in GitHub, a folder to sync to the remote GitHub repository, and all the required steps to glue them together. From here, all another user has to do is navigate to the shared folder and drop a script or config file into the appropriate directory. Once per hour, the scheduled task will fire and sync to GitHub. Again, you should not put production code in a public GitHub repository; however, once you have this setup, it shouldn't be too difficult to adapt this to your production environment.
Enjoy.

IV PowerShell Language Features

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

4. Exploring Experimental Features in PowerShell 7

by Dave Carroll

PowerShell 7 marks the first long-term servicing release to support experimental features, though PowerShell Core 6.1 introduced them to the community. In this chapter, you learn about PowerShell experimental features along with their basic command usage. Additionally, you learn how to include them in your code and module manifest.

As a bonus, this chapter includes a few ideas on how to collect metrics, which can be valuable in determining the success and adoption rate of experimental features.

The examples in this chapter use PowerShell 7.0.3.

Experimental Features Definition

New features or behavior that are not production-ready are considered experimental. For example, the new `ConciseView` for `$ErrorView` in PowerShell 7 was initially implemented as an experimental feature.

Users can choose to opt-in for an experimental feature on an individual basis. Administrators can choose to opt-in at the system level.

Using the built-in support for experimental features for your module, you can provide users with alternate commands, parameters, or behaviors as required.

Please note that user configuration will take precedence over system configuration.

In PowerShell preview releases, all experimental features contained in the release are enabled by default.

Experimental Feature Commands

The following commands allow you to discover, enable, and disable experimental features.

Requirements to Enable or Disable Experimental Features for All Users

For Windows operating systems, you need to start the PowerShell session using the `Run As Administrator` mode. For Linux-based operating systems, you need to start the PowerShell session using `sudo`.

Get-ExperimentalFeature

The command `Get-ExperimentalFeature` displays a list of experimental features discovered on the system at the beginning of the PowerShell session. These features can come from the PowerShell engine itself or modules. The output of this command is an object with the properties *Name*, *Enabled*, *Source*, and *Description*. The *Source* property shows where the experimental feature is defined.

Experimental feature discovery targets the paths in `$env:PSModulePath`.

Experimental features can be specific to an Operating System as you can see in the following table.

Table of Experimental Features with Source and Operating System

Name	Source	Operating System
PSCommandNotFoundSuggestion	PSEngine	All
PSImplicitRemotingBatching	PSEngine	All
PSNullConditionalOperators	PSEngine	All
PSUnixFileStat	PSEngine	Linux
Microsoft.PowerShell.Utility.PSManageBreakpointsInRunspace	Module	All
PSDesiredStateConfiguration.InvokeDscResource	Module	All

The `Get-ExperimentalFeature` command only returns those experimental features that are available to all operating systems or those specific to the current operating system hosting the PowerShell session.

Enable-ExperimentalFeature

The `Enable-ExperimentalFeature` command turns on one or more experimental features for the current user or all users.

Enabling a feature adds it to an array in the `ExperimentalFeatures` key in the PowerShell configuration file, `$PSHOME\powershell.config.json`. If you do not specify a `Scope`, it defaults to `CurrentUser`.

PowerShell User Configuration File Location

For Windows, the default location for the user configuration file is `$HOME\Documents\PowerShell`.

For Linux, the default location is `$HOME\.config\powershell`.

You can turn on all experimental features in one line, as shown below.

Enable All Experimental Features for All Users

```
Get-ExperimentalFeature | Enable-ExperimentalFeature -Scope AllUsers
```

Restart Sessions

Take note of the warning message that serves as a reminder to restart the PowerShell session. You must close all console sessions, of the same version and platform before the change will take effect. This includes any terminals of the same version in any open editor, such as Visual Studio Code or Visual Studio.

Contents of `$PSHOME\powershell.config.json` After Enabling Experimental Features for All Users on Windows

```
{
    "WindowsPowerShellCompatibilityModuleDenyList": [
        "PSScheduledJob",
        "BestPractices",
        "UpdateServices"
        ],
    "Microsoft.PowerShell:ExecutionPolicy":"RemoteSigned",
    "ExperimentalFeatures": [
        "PSCommandNotFoundSuggestion",
        "PSImplicitRemotingBatching",
        "PSNullConditionalOperators",
        "Microsoft.PowerShell.Utility.PSManageBreakpointsInRunspace",
        "PSDesiredStateConfiguration.InvokeDscResource"
        ]
}
```

Disable-ExperimentalFeature

The `Disable-ExperimentalFeature` command turns off one or more experimental features.

Disable Experimental Feature `PSNullConditionalOperators` for All Users

```
Disable-ExperimentalFeature -Name PSNullConditionalOperators -Scope AllUsers
```

As with enabling, when you disable one or more features, you must close all PowerShell sessions and start a new session.

Disabling the feature removes its entry from the enabled feature list in the configuration file for the given scope. The `ExperimentalFeatures` key remains even if you disable all experimental features.

Adding Experimental Features to Your Module

You can add multiple experimental commands, parameters, or behaviors to your module. First, name each experimental feature and prepare a description for it. Next, add a new entry for each in the `PrivateData.PSData` section in the module manifest.

After updating the module manifest, you can then add experimental behavior using one or both of the following methods.

- Add the `Experimental` attribute and the appropriate `Experiment` action where needed.
- Include experimental feature behavior directly in your code.

Please note that, when you add experimental features to a module, you must include a `PowerShellVersion` entry in the module manifest, and it must be 6.1 or higher.

Experimental Feature Name and Description

Experimental feature names must be in the format of *ModuleName.FeatureName*.

Assume that you want to include an experimental feature for the module `PSTemperature` to support the Rankine temperature scale. A valid name could be `PSTemperature.SupportRankine`.

The name could include very short descriptive text that identifies where you include the experimental feature in your module. `DemoModule.FunctionShowHelloWorld` could indicate that the function `Show-HelloWorld` is experimental or that it has an experimental counterpart.

The name can include multiple periods if your module name includes periods. For instance, the name

```
Microsoft.PowerShell.Utility.PSManageBreakpointsInRunspace
```

is valid as `Microsoft.PowerShell.Utility` is the module's name.

Please note that experimental features included with the PowerShell engine, as shown by the *Source* `PSEngine`, adhere to the naming convention of ***PS****DescriptiveText*.

In addition to the name, you must write a short description of the experimental feature that you will include in the module manifest. This description should provide the user with information about the experimental behavior and, potentially, how to access it once enabled.

Module Manifest

In the module manifest file, typically named `ModuleName.psd1`, you must add a new entry to the section `PrivateData.PSData`. The entry is an array of hashtables, with each hashtable being a specific experimental feature. The hashtable must include Name and Description keys.

Example of PrivateData.PSData ExperimentalFeatures Entry

```
PrivateData = @{
    PSData = @{
        ExperimentalFeatures = @(
            @{
                Name = 'PSTemperature.SupportRankine'
                Description = 'Support Rankine Temperature Scale'
            }
        )
    }
}
```

You can include more than one experimental feature in your module. Each one must have a unique name.

Example of PrivateData.PSData ExperimentalFeatures Entry with Multiple Features

```
PrivateData = @{
    PSData = @{
        ExperimentalFeatures = @(
            @{
                Name = 'DemoModule.ExperimentalFunction'
                Description = 'Demo of Experimental Function'
            },
            @{
                Name = 'DemoModule.ExpParameter'
                Description = 'Demo of Experimental Parameter'
            }
        )
    }
}
```

Once you have added the experimental feature(s) to the module manifest file, you should test its validity by using the `Test-ModuleManifest` command.

```
Test-ModuleManifest -Path 'PSTemperature.psd1'
```

If an experimental feature name does not meet the correct format, `Test-ModuleManifest` will fail.

`Test-ModuleManifest` Error Due to Invalid Experimental Feature Name

```
Test-ModuleManifest:
  One or more invalid experimental feature names found: PSDemoExpFeature.
A module experimental feature name should follow this convention:
  'ModuleName.FeatureName'.
```

Experimental Attribute and Experiment Action

You must include the `Experimental` attribute and an appropriate `Experiment` action to a function or cmdlet declaration or a parameter declaration.

The `Experimental` Attribute is a decorator much like the `CmdletBinding` Attribute and the `Parameter` Attribute.
Decorators, which are the embodiment of the *decorator design pattern*, add behaviors to an object.

For instance, including `[CmdletBinding()]` at the beginning of a function or a `[Parameter()]` attribute for any parameter, it becomes an *advanced function* which enables it to inherit *common parameters* and take advantage of the pipeline.

Each of these decorators has parameters that change how the script, function, or parameter behaves.

Add the following code to your function, cmdlet, or parameter to enable the experimental behavior.

```
[Experimental(NameOfExperimentalFeature, ExperimentAction)]
```

In a previous section, you learned about the naming scheme for `NameOfExperimentalFeature`.

The `ExperimentAction` is an `enum` with values of `Hide` or `Show`.

- `Hide` disables specific behavior with the experimental feature state disabled—*this is the default state.*
- `Show` enables specific behavior with the experimental feature state enabled.

For functions, you can use either the text or reference the `enum` directly with *[ExperimentAction]::Hide* or *[ExperimentAction]::Show*. For cmdlets, written in *C#*, you must use *ExperimentAction.Hide* or *ExperimentAction.Show*.

When used correctly, they can enforce mutual exclusivity between different versions of a command or parameter.

Example of Mutual Exclusivity for `ExperimentAction`

```
function Get-LoremIpsum {
    [CmdletBinding()]
    param(
        [Experimental('DemoModule.ExpParameter', [ExperimentAction]::Hide)]
        [switch]$Display,

        [Experimental('DemoModule.ExpParameter', [ExperimentAction]::Show)]
        [switch]$Show
    )

    if ($Display) {
        $Lorem = @()
        $Lorem += 'Lorem ipsum dolor sit amet, consectetur adipiscing elit.'
        $Lorem += 'Sed varius mi erat, in laoreet nibh eleifend eget.'
        $Lorem += 'Phasellus odio diam, tincidunt rhoncus massa in, feugiat'
        $Lorem += 'iaculis mauris. Nulla ornare enim et semper tincidunt.'
        $Lorem += 'Maecenas ac tempor quam, in scelerisque lorem.'
        $Lorem -join ' ' | Write-Output
    }

    if ($Show) {
        'The quick brown fox jumps over the lazy dog.' | Write-Output
    }
}
```

When `DemoModule.ExpParameter` is not enabled, the function `Get-LoremIpsum` will have the `Display` switch parameter *(along with the common parameters)*. The `Show` switch parameter replaces the `Display` parameter when the experimental feature is enabled.

```
# disabled
Get-LoremIpsum -Display

# enabled
Get-LoremIpsum -Show
```

Similarly, you can include two functions that are mutually exclusive based on the state of the experimental feature.

```
# This function is exported when the experimental feature is disabled.
# This is the default.
function Show-HelloWorld {
    [Experimental('DemoModule.ExperimentalFunction', [ExperimentAction]::Hide)]
    [CmdletBinding()]
    param()
    'PowerShell 7 is shipping soon!' | Write-Host -ForegroundColor Green
}

# This function is exported when the experimental feature is enabled.
function Show-HelloWorld {
    [Experimental('DemoModule.ExperimentalFunction', [ExperimentAction]::Show)]
    [CmdletBinding()]
    param()
    'PowerShell 7 is here!' | Write-Host -ForegroundColor Yellow
}
```

The command names need not be the same. However, be sure to include all exportable commands, regardless of their experimental feature state in the module manifest `FunctionsToExport` or `CmdletsToExport` entries.

Include Experimental Feature Behavior Inside a Code Block

You can include additional behavior not tied directly to a function, cmdlet, or parameter. Once you have named your experimental feature and updated the module manifest, use the static `IsEnabled()` method of the `System.Management.Automation.ExperimentalFeature` class.

Example of Including Experimental Behavior within a Code Block

```
if ([ExperimentalFeature]::IsEnabled('DemoModule.ExperimentalBehavior')) {
    # code specific to experimental feature
}
```

When writing Pester tests for your experimental code, you can check the state of an experimental feature similarly.

Collecting Metrics

Whether your module includes experimental features or not, a good practice would be to understand users' experience with it. Metrics on the frequency of use, environment configuration, and generated exceptions or errors are prime candidates for collection.

The collection process, sometimes called telemetry, can look very different, given the target audience and its size. Target audiences can include colleagues in your department, employees within your company, employees or contractors in other companies, the public at large, or any combination of these.

For experimental features, the collected metrics should provide the developer, or development team, with enough information to decide on the next step for the feature.

Questions that metrics should help answer:

- Should the experimental feature be *graduated* to stable and be included as a regular feature in an upcoming release?
- What errors generated will be addressed?
- Given a low adoption rate of the experimental feature, should it be removed entirely from the module?
- Can the experimental feature be redesigned to better align with user expectations?

Conduct a Survey

For small-to-midsize target audiences, a simple survey may provide sufficient data. The survey could be in the form of directly asking the handful of users if they have used the experimental feature and, if so, what did they like or dislike about it. Alternatively, the survey could be an email to the users, or it could be implemented via a hosted survey service.

The data you collect in your surveys would need to be collated and presented in a report. The more data points included in your report, the longer creating the report will take. Though you may collect more metrics than you use, the report should focus only on the data points that will help make the module better.

Send Metrics to a Local Repository

For midsize target audiences, a simple survey would generate considerable overhead consuming valuable time you could spend on writing code. You can shift this overhead to the development of an automated collection of metrics to a local repository, whether a text file on a network share or tables in a relational database.

Once you have collected the metrics, you can then generate the metrics report using this data.

Send Metrics to a Cloud Service

Gathering telemetry requires adequate infrastructure and appropriate access. Unless your team has time and resources to handle configuration and access requests to send telemetry data to a local repository, you should consider using a cloud-hosted solution, ideal for most mid-to-large sized target audiences.

In the software development realm, Application Performance Management/Monitoring (APM) solutions include telemetry. Microsoft offers Azure Application Insights, while Amazon Web Services (AWS) offers several third-party APM solutions in their AWS Marketplace. Many other vendors offer APM solutions, as well.

Cost and ease of use would be two factors that you must take into account when selecting an APM solution. Ultimately, it will be up to your company, or you, to determine which solution is the best for your particular situation.

In highly secure environments, collecting to a local repository may be a better choice than sending telemetry data to an externally hosted solution.

Additional Information

Learn more about Experimental Features by reviewing the contextual help, `Get-Help about_-Experimental_Features` or each command's help, `Get-Help Get-ExperimentalFeature`, locally or online using the `-Online` switch. *Be sure to `Update-Help` to get the latest help links.*

To learn how the PowerShell Team addressed the original proposal for experimental features, visit the finalized Request for Comments (RFC) for Experimental Features RFC0029[1].

Sample Modules

This chapter includes examples from two sample modules, one a script module and the other a binary module. You can find them in this book's Extras GitHub repo[2]. Once there, navigate to Volume 3 and look for this chapter's name on the list.

Summary

Experimental feature support in PowerShell 7 provides users the choice to opt-in to alternate, not-widely-tested behaviors. It includes three commands that allow you to discover and enable or disable experimental features. PowerShell, itself, delivers several experimental features via the engine and two others with included modules.

As a PowerShell developer, you can include experimental features with your modules, as well. After updating your module's manifest with the name and description of your new experimental feature, you can include the `Experimental` attribute and appropriate `Experiment` action, as often as needed. You can also include the experimental code within an `if` statement that validates whether the feature is enabled before exposing the new behavior.

Lastly, you learned about collecting metrics that can help you in determining whether to graduate an experimental feature or abandon it.

[1] https://github.com/PowerShell/PowerShell-RFC/blob/master/5-Final/RFC0029-Support-Experimental-Features.md
[2] https://github.com/devops-collective-inc/PSConfBookExtras

Afterword

by Michael Zanatta

To the Authors and Editors, I would like to say thank you for spending the countless hours that it took to create this book.

To you, the reader who purchased this book and helped support a good cause: thank you, and I hope that you enjoyed it.

I was involved with the PowerShell Conference Book Volume 2, as an author and an un-official editor, two things which I never thought that I would be doing. Fast-forward to Conference Book Volume 3; I decided to take up the editing hat again as Senior Editor. As the saying goes, "Many hands make light work." Having additional editors on hand was key to the success of this book.

Recently, Mark Kraus approached me and asked if I would be interested in continuing as Editor-in-Chief, with Mark stepping down from the Project. After some discussion with my wife, I accepted. So what does this mean? For you, the reader? Nothing. The book will still focus on delivering the same great content.

So What didn't go well?

LeanPub margin issues continue to cause problems with producing a "printable" manuscript. There is a known intermittent bug within the compiler which ignores margins for long.pieces.of.text, code blocks, and links.

So when will the next book be out?

Short Answer: Next Year. We all need a break, the authors, the editors, and you the readers will need some time to read and reread this book!

If you are interested in becoming a future editor or author, you can follow me on @PowerShellMich1[3] for updates.

One Final Thought:

We are living in crazy times right now. I like to remind myself of Carl Sagan's "Pale Blue Dot" photo to provide a perspective of the reality of our existence. After all, we are all in-this-together.

> "On a Mote of Dust Suspended in a Sunbeam."

[3] https://twitter.com/PowerShellMich1

Till then, next time,

Michael Zanatta
Senior Editor,
The PowerShell Conference Book.