Learn POWERSHELL SCRIPTING IN A MONTH OF LUNCHES

MONDAY
TUESDAY
WEDNESDAY
THURSDAY
1 Before you begin
2 Setting up your scripting environment
pretty cool
3 WWPD: what would PowerShell do?
4 Review: parameter binding and the PowerShell pipeline
7 The many forms of scripting (and which to use)
interesting
8 Scripts and security
9 Aways design first
10 Avoid bugs: start with a command
11 Building a basic
21 Professional-grade scripting
22 An introduction to source control with git
23 Pestering your script
24 Signing your script
25 Publishing your script
26
28
29 Wrapping up the .NET Framework
30 Storing data-not in Excel!
31 Never the end

SAMPLE CHAPTERS

DON JONES AND JEFFERY HICKS

MANNING

Learn PowerShell Scripting in a Month of Lunches

by Don Jones
Jeffery Hicks

Chapter 14

Copyright 2018 Manning Publications

brief contents

14 Simple help: making a comment

One of the things we all love about PowerShell is its help system. Like Linux's `man` pages, PowerShell's help files can provide a wealth of information, examples, instructions, and more. So we definitely want to provide help with the tools we create—and you should, too. You have two ways of doing so. First, you can write full PowerShell help that consists of external, XML-formatted Microsoft Assistance Markup Language (MAML) files, which can even include versions for different languages. This is an advanced topic that we won't cover in this book. In fact, with the advent of modules like PlatyPS, you won't ever have to learn MAML. For now, we're going to use the simpler, single-language, comment-based help that lives right inside your function.

14.1 Where to put your help

There are three legal places where PowerShell will look for your specially formatted comments, in order to turn them into help displays:

- Just before your function's opening `function` keyword, with no blank lines between the last comment line and the function. We don't like this spot, because we prefer…
- Just inside the function, after the opening `function` declaration and before your `[CmdletBinding()]` or `Param` parts. We love this spot, because it's easier to move your help with the function if you're copying and pasting your code someplace else. Your comments will also collapse into the function if you use an editor that has code-folding features. This is where you'll find that the majority of people stick their help.

- As the last thing in your function before the closing }. We're not fans of this spot, either, because having your comments at the top of the function helps better document the function for someone reading the code.

14.2 Getting started

As you'll see, there's nothing especially complicated about any of this. The best way to understand is to dive in and look at an example.

Listing 14.1 Comment-based help

```
function Get-MachineInfo {
<#
.SYNOPSIS
Retrieves specific information about one or more computers, using WMI or
CIM.
.DESCRIPTION
This command uses either WMI or CIM to retrieve specific information about
one or more computers. You must run this command as a user who has
permission to remotely query CIM or WMI on the machines involved. You can
specify a starting protocol (CIM by default), and specify that, in the
event of a failure, the other protocol be used on a per-machine basis.
.PARAMETER ComputerName
One or more computer names. When using WMI, this can also be IP addresses.
IP addresses may not work for CIM.
.PARAMETER LogFailuresToPath
A path and filename to write failed computer names to. If omitted, no log
will be written.
.PARAMETER Protocol
Valid values: Wsman (uses CIM) or Dcom (uses WMI). Will be used for all
machines. "Wsman" is the default.
.PARAMETER ProtocolFallback
Specify this to automatically try the other protocol if a machine fails.
.EXAMPLE
Get-MachineInfo -ComputerName ONE,TWO,THREE
This example will query three machines.
.EXAMPLE
Get-ADUser -filter * | Select -Expand Name | Get-MachineInfo
This example will attempt to query all machines in AD.
#>
    [CmdletBinding()]
    Param(
        [Parameter(ValueFromPipeline=$True,
                   Mandatory=$True)]
        [Alias('CN','MachineName','Name')]
        [string[]]$ComputerName,

        [string]$LogFailuresToPath,

        [ValidateSet('Wsman','Dcom')]
        [string]$Protocol = "Wsman",

        [switch]$ProtocolFallback
    )
```

```
 BEGIN {}

 PROCESS {
    foreach ($computer in $computername) {

        if ($protocol -eq 'Dcom') {
            $option = New-CimSessionOption -Protocol Dcom
        } else {
            $option = New-CimSessionOption -Protocol Wsman
        }

        Write-Verbose "Connecting to $computer over $protocol"
        $session = New-CimSession -ComputerName $computer `
                                  -SessionOption $option

        Write-Verbose "Querying from $computer"
        $os_params = @{'ClassName'='Win32_OperatingSystem'
                       'CimSession'=$session}
        $os = Get-CimInstance @os_params

        $cs_params = @{'ClassName'='Win32_ComputerSystem'
                       'CimSession'=$session}
        $cs = Get-CimInstance @cs_params

        $sysdrive = $os.SystemDrive
        $drive_params = @{'ClassName'='Win32_LogicalDisk'
                          'Filter'="DeviceId='$sysdrive'"
                          'CimSession'=$session}
        $drive = Get-CimInstance @drive_params

        $proc_params = @{'ClassName'='Win32_Processor'
                         'CimSession'=$session}
        $proc = Get-CimInstance @proc_params |
                Select-Object -first 1

        Write-Verbose "Closing session to $computer"
        $session | Remove-CimSession

        Write-Verbose "Outputting for $computer"
        $obj = [pscustomobject]@{'ComputerName'=$computer
                   'OSVersion'=$os.version
                   'SPVersion'=$os.servicepackmajorversion
                   'OSBuild'=$os.buildnumber
                   'Manufacturer'=$cs.manufacturer
                   'Model'=$cs.model
                   'Procs'=$cs.numberofprocessors
                   'Cores'=$cs.numberoflogicalprocessors
                   'RAM'=($cs.totalphysicalmemory / 1GB)
                   'Arch'=$proc.addresswidth
                   'SysDriveFreeSpace'=$drive.freespace}
        Write-Output $obj

    } #foreach
} #PROCESS

END {}

} #function
```

The help here reflects what we believe is the bare minimum for inclusion in the race of Upright Human Beings. Some notes

- You don't have to use all-uppercase letters, but the period preceding each help keyword (`.SYNOPSIS`, `.DESCRIPTION`) must be in the first column.
- We used a block comment (`<#....#>`); you could also use line-by-line comments—that is, each line preceded by a `#` character. The block comment looks nicer and is considered a collapsible region in some scripting editors.
- `.SYNOPSIS` is meant to be a very short description of what your command does.
- `.DESCRIPTION` is a longer description, which can be full of details, instructions, and insights.
- `.PARAMETER` *is followed by the parameter name* and then a description of the parameter's use. You don't need to provide a listing for every single parameter.
- `.EXAMPLE` should be followed immediately *by the example itself*; PowerShell will add a PowerShell prompt in front of this line when the help is displayed. If your tool takes advantage of different providers such as the registry, you can certainly insert an appropriate prompt to illustrate your example. Subsequent text can explain the example.
- You can put blank comment lines between each of these settings to make it all easier to read in code.
- You normally don't need to worry about line length. PowerShell will wrap lines as necessary, depending on the console size of the current host. But if you want to manually break lines, a width of 80 characters is your best bet:

```
<#
.SYNOPSIS
Retrieves specific information about one or more computers, using WMI or
CIM.
.DESCRIPTION
This command uses either WMI or CIM to retrieve specific information about
one or more computers. You must run this command as a user who has
permission
to remotely query CIM or WMI on the machines involved. You can
specify a starting protocol (CIM by default), and specify that, in the
event of a failure, the other protocol be used on a per-machine basis.
.PARAMETER ComputerName
One or more computer names. When using WMI, this can also be IP addresses.
IP addresses may not work for CIM.
.PARAMETER LogFailuresToPath
A path and filename to write failed computer names to. If omitted, no log
will be written.
.PARAMETER Protocol
Valid values: Wsman (uses CIM) or Dcom (uses WMI). Will be used for all
machines. "Wsman" is the default.
.PARAMETER ProtocolFallback
Specify this to automatically try the other protocol if a machine fails.
.EXAMPLE
Get-MachineInfo -ComputerName ONE,TWO,THREE
This example will query three machines.
```

```
    .EXAMPLE
    Get-ADUser -filter * | Select -Expand Name | Get-MachineInfo
    This example will attempt to query all machines in AD.
    #>
```

As we wrote, these elements are the *bare minimum*. You can do more. A lot more.

14.3 Going further with comment-based help

You can use an `.INPUTS` section to list .NET class types, one per line, that your command accepts as input from the pipeline. This is useful for helping others understand what kinds of input your command can deal with:

```
.INPUTS
System.String
```

Similarly, `.OUTPUTS` lists the type names that your script outputs. Because ours presently only outputs a generic `PSObject`, there's not much point in listing anything.

A `.NOTES` section can list additional information, which is only displayed when the full help is requested by the user:

```
.NOTES
version     : 1.0.0
last updated: 1 February, 2017
```

A `.LINK` heading, followed by a topic name or a URL, appears as a Related Topic in the help. Use one `.LINK` keyword for each related topic; don't put multiples under a single `.LINK`:

```
.LINK
https://powershell.org/forums/
.LINK
Get-CimInstance
.LINK
Get-WmiObject
```

There's more, too—read the `about_comment_based_help` topic in PowerShell for the full list. We'll include a few of them in upcoming chapters, as we add functionality that pertains to those help keywords, so be on the lookout.

14.4 Broken help

PowerShell's a little picky—okay, a lot picky—about help formatting and syntax. Get just one thing wrong, and none of the help will work, *and* you won't get an error message or explanation. So if you're not getting the help display you expect, go review your help keyword spelling, period locations, and other details *very carefully*.

14.5 Beyond comments

Comment-based help has more than a few limitations, but it's important to understand why it exists. Originally, PowerShell only supported external help, stored in

XML-based files written in a dialect called Microsoft Assistance Markup Language. MAML is incomprehensible—like, seriously unreadable to a human. But it offers advantages over comment-based help. Although it's harder to create, it

- Is separated from your code, so it can be updated independently. It's the basis of how PowerShell's `Update-Help` command works.
- Can be delivered in multiple languages, allowing PowerShell to offer localized help content to different audiences.
- Is parsed by PowerShell into an object hierarchy, providing additional features and functionality that can make help content useful across a wider range of situations.

So if MAML is so cool but so hard to make, what do you do? Back in the day, a bunch of different folks made tools that let you basically copy and paste content into a GUI that would then spit out MAML files for you. Easier, but super time-consuming. Nowadays, all the cool kids are using an open source project called PlatyPS. PlatyPS lets you write your help content in Markdown, which is a simple markup language. Markdown is the native markup language of GitHub, meaning your help files can be easily read and edited right on that website, if you're hosting a project there. PlatyPS can then take that Markdown and produce a valid MAML file. Other tools can consume Markdown and produce HTML, if you want to have web-based help for some reason. Markdown becomes the source format for your help (it's easy to read and edit with any text editor—you don't even need a dedicated Markdown editor, although VS Code has excellent Markdown plugins you can try), and you produce everything else from there.

If you've never written help for your code, PlatyPS can examine the code and create a framework, or *stub*, for your Markdown help files. The stub will include all of your parameters and so forth, with as much data as PlatyPS can figure out already filled in—like which parameters are mandatory, which ones accept pipeline input, and so on. PlatyPS can help you *maintain* your help files, too. Say you add a parameter, or change one, or whatever. It can look at your code, figure that out, and update your existing help files with stubs, which you can then fill in to fully document whatever's new and changed in your code.

We *love* PlatyPS and Markdown. Although they're bigger topics than we were ready to tackle for this book, we wanted to give you some directions for future exploration.

14.6 Your turn

Time to add some comment-based help to your function.

14.6.1 Start here

Here's where we left off after chapter 13. You can use this as a starting point, or use your own result from that chapter.

Listing 14.2 Set-TMServiceLogon

```
function Set-TMServiceLogon {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$True,
                   ValueFromPipelineByPropertyName=$True)]
        [string]$ServiceName,

        [Parameter(Mandatory=$True,
                   ValueFromPipeline=$True,
                   ValueFromPipelineByPropertyName=$True)]
        [string[]]$ComputerName,

        [Parameter(ValueFromPipelineByPropertyName=$True)]
        [string]$NewPassword,

        [Parameter(ValueFromPipelineByPropertyName=$True)]
        [string]$NewUser,

        [string]$ErrorLogFilePath
    )

BEGIN{}

PROCESS{
    ForEach ($computer in $ComputerName) {

        Write-Verbose "Connect to $computer on WS-MAN"
        $option = New-CimSessionOption -Protocol Wsman
        $session = New-CimSession -SessionOption $option `
                                  -ComputerName $Computer

        If ($PSBoundParameters.ContainsKey('NewUser')) {
            $args = @{'StartName'=$NewUser
                      'StartPassword'=$NewPassword}
        } Else {
            $args = @{'StartPassword'=$NewPassword}
            Write-Warning "Not setting a new user name"
        }

        Write-Verbose "Setting $servicename on $computer"
        $params = @{'CimSession'=$session
                    'MethodName'='Change'
                    'Query'="SELECT * FROM Win32_Service " +
                            "WHERE Name = '$ServiceName'"
                    'Arguments'=$args}
        $ret = Invoke-CimMethod @params

        switch ($ret.ReturnValue) {
            0  { $status = "Success" }
            22 { $status = "Invalid Account" }
            Default { $status = "Failed: $($ret.ReturnValue)" }
        }

        $props = @{'ComputerName'=$computer
                   'Status'=$status}
        $obj = New-Object -TypeName PSObject -Property $props
        Write-Output $obj
        Write-Verbose "Closing connection to $computer"
        $session | Remove-CimSession

    } #foreach
} #PROCESS

END{}

} #function
```

14.6.2 Your task

Add, at a minimum, the following to your tool:

- Synopsis
- Description
- Parameter descriptions
- Two examples, including descriptions

Import your module, and test your help (`Help Set-TMServiceLogon -ShowWindow`, for example) to make sure it works.

14.6.3 Our take

Here's the help we came up with. As always, you'll find this in the code downloads at www.manning.com/books/learn-powershell-scripting-in-a-month-of-lunches, under this chapter's folder.

Listing 14.3 Our solution

```
function Set-TMServiceLogon {
<#
.SYNOPSIS
Sets service login name and password.
.DESCRIPTION
This command uses either CIM (default) or WMI to
set the service password, and optionally the logon
user name, for a service, which can be running on
one or more remote machines. You must run this command
as a user who has permission to perform this task,
remotely, on the computers involved.
.PARAMETER ServiceName
The name of the service. Query the Win32_Service class
to verify that you know the correct name.
.PARAMETER ComputerName
One or more computer names. Using IP addresses will
fail with CIM; they will work with WMI. CIM is always
attempted first.
.PARAMETER NewPassword
A plain-text string of the new password.
.PARAMETER NewUser
Optional; the new logon user name, in DOMAIN\USER
format.
.PARAMETER ErrorLogFilePath
```

```
If provided, this is a path and filename of a text
file where failed computer names will be logged.
#>
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$True,
                   ValueFromPipelineByPropertyName=$True)]
        [string]$ServiceName,

        [Parameter(Mandatory=$True,
                   ValueFromPipeline=$True,
                   ValueFromPipelineByPropertyName=$True)]
        [string[]]$ComputerName,

        [Parameter(ValueFromPipelineByPropertyName=$True)]
        [string]$NewPassword,

        [Parameter(ValueFromPipelineByPropertyName=$True)]
        [string]$NewUser,

        [string]$ErrorLogFilePath
    )

BEGIN{}

PROCESS{
    ForEach ($computer in $ComputerName) {

        Write-Verbose "Connect to $computer on WS-MAN"
        $option = New-CimSessionOption -Protocol Wsman
        $session = New-CimSession -SessionOption $option `
                                  -ComputerName $Computer

        If ($PSBoundParameters.ContainsKey('NewUser')) {
            $args = @{'StartName'=$NewUser
                      'StartPassword'=$NewPassword}
        } Else {
            $args = @{'StartPassword'=$NewPassword}
            Write-Warning "Not setting a new user name"
        }

        Write-Verbose "Setting $servicename on $computer"
        $params = @{'CimSession'=$session
                    'MethodName'='Change'
                    'Query'="SELECT * FROM Win32_Service " +
                            "WHERE Name = '$ServiceName'"
                    'Arguments'=$args}
        $ret = Invoke-CimMethod @params

        switch ($ret.ReturnValue) {
            0  { $status = "Success" }
            22 { $status = "Invalid Account" }
            Default { $status = "Failed: $($ret.ReturnValue)" }
        }

        $props = @{'ComputerName'=$computer
                   'Status'=$status}
        $obj = New-Object -TypeName PSObject -Property $props
        Write-Output $obj
        Write-Verbose "Closing connection to $computer"
        $session | Remove-CimSession

    } #foreach
} #PROCESS

END{}

} #function
```

Adding comment-based help doesn't have to be a tedious chore. Use the snippets feature of your scripting editor to create a template. In the PowerShell ISE, if you press Ctrl-J to access the built-in snippets, the one for Cmdlet (Advanced Function) will have most of what you need.

And before we sign off, here's a quick pro tip: Comment-based help is tolerant of extra whitespace. So instead of this

```
.SYNOPSIS
Sets service login name and password.
.DESCRIPTION
This command uses either CIM (default) or WMI to
set the service password, and optionally the logon
user name, for a service, which can be running on
one or more remote machines. You must run this command
as a user who has permission to perform this task,
remotely, on the computers involved.
.PARAMETER ServiceName
The name of the service. Query the Win32_Service class
to verify that you know the correct name.
```

you could do this:

```
.SYNOPSIS
Sets service login name and password.

.DESCRIPTION
This command uses either CIM (default) or WMI to
set the service password, and optionally the logon
user name, for a service, which can be running on
one or more remote machines. You must run this command
as a user who has permission to perform this task,
remotely, on the computers involved.

.PARAMETER ServiceName
The name of the service. Query the Win32_Service class
to verify that you know the correct name.
```

Those extra blank lines go a *long* way toward making your code more readable, and they don't affect the help-file displays that PowerShell creates from your comments.

POWERSHELL

Learn POWERSHELL SCRIPTING IN A MONTH OF LUNCHES

Don Jones and Jeffery Hicks

Free eBook
SEE INSERT

Automate it! With Microsoft's PowerShell language, you can write scripts to control nearly every aspect of Windows. Just master a few straightforward scripting skills, and you'll be able to eliminate repetitive manual tasks, create custom reusable tools, and build effective pipelines and workflows. Once you start scripting in PowerShell, you'll be amazed at how many opportunities you'll find to save time and effort.

Learn PowerShell Scripting in a Month of Lunches teaches you how to expand your command-line PowerShell skills into effective scripts and tools. In 27 bite-size lessons, you'll discover instantly useful techniques for writing efficient code, finding and squashing bugs, organizing your scripts into libraries, and much more. Advanced scripters will even learn to access the .NET Framework, store data long term, and create nice user interfaces.

WHAT'S INSIDE

- Designing functions and scripts
- Effective pipeline usage
- Dealing with errors and bugs
- Professional-grade scripting practices

Written for devs and IT pros comfortable with PowerShell and Windows.

Don Jones and *Jeffery Hicks* are internationally recognized PowerShell teachers, consultants, and authors.

"A very clear and concise depiction of the best parts of PowerShell."

—Justin Coulston
Intellectual Technology

"A great resource for those who want to create scripts for task automation."

—Bruno Sonnino
Revolution Software

"Teaches you how to become an informed expert in PowerShell scripting."

—Shankar Swamy
Stealth Mode Start-up

"Real-world examples, best practices, and tips from two of the most respected PowerShell MVPs."

—Roman Levchenko
Microsoft MVP

"It makes you stop and think, not just 'read and nod.'"

—Reka Horvath, Wirecard CEE

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit www.manning.com/books/learn-powershell-scripting-in-a-month-of-lunches

ISBN-13: 978-1-61729-509-6
ISBN-10: 1-61729-509-4
54499
9 781617 295096

MANNING $44.99 / Can $59.99 [INCLUDING eBOOK]