# *Learn*
# POWERSHELL SCRIPTING
# IN A MONTH OF LUNCHES


## SAMPLE CHAPTERS

DON JONES AND JEFFERY HICKS

MANNING

# *Learn PowerShell Scripting in a Month of Lunches*

by Don Jones
Jeffery Hicks

**Chapter 8**

# brief contents

# 8 Always design first

Before you sit down and start coding up a function or a class, you need to do some thinking about its *design*. We almost constantly see toolmaking newcomers start charging into their code, and before long they've made some monstrosity that's harder to work with than it should be. In this chapter, we're going to lay out some of the core PowerShell tool design principles, to help you stay on the path of Toolmaking Righteousness. To be clear, all we're doing here is building on what we laid out in part 1 of this book. Now we're ready to provide some more concrete examples.

## 8.1 Tools do one thing

As we've mentioned before, the Prime Directive for a PowerShell tool is that *it does one thing*. You can see this in almost every single tool—that is, *command*—that ships with PowerShell. `Get-Service` gets services. It doesn't stop them. It doesn't read computer names from a text file. It doesn't modify services. It does *one thing*.

This concept is one we see newcomers violate the most. For example, we'll see folks build a command that has a `-ComputerName` parameter for accepting a remote machine name, as well as a `-FilePath` parameter so that they can alternately read computer names from a file. From PowerShell's perspective and ours, that's Dead Wrong, because it means the tool is doing two things instead of just one. A correct design to follow the paradigm would be to stick with the `-ComputerName` parameter and let it accept strings (computer names) from the pipeline. You could also feed it names from a file by using a `-ComputerName (Get-Content filename.txt)` parenthetical construct. Or define the `-Computername` parameter to accept input by value:

```
get-content filename.txt | get-serverstuff
```

The `Get-Content` command reads text files; you shouldn't duplicate that functionality in your command without a strong reason. Why reinvent the wheel?

Let's explore that antipattern for a moment. Here's an example of using a completely fake command (meaning, don't try this at home) in two different ways:

```
# Specify three computer names
Get-CompanyStuff -Computername ONE,TWO,THREE

# Specify a file containing computer names
Get-CompanyStuff -FilePath ./names.txt
```

That approach overcomplicates the tool, making it harder to write, harder to debug, harder to test, and harder to maintain. We'd go with this approach to provide the exact same effect in a simpler tool:

```
# Specify three computer names
Get-CompanyStuff -Computername ONE,TWO,THREE

# Specify a file containing computer names
Get-CompanyStuff -Computername (Get-Content ./names.txt)

# Or if you were smart in making the tool...
Get-Content ./names.txt | Get-CompanyStuff
```

Those patterns do a much better job of mimicking how PowerShell's own core commands work. But let's explore one more antipattern, which is "but I have the computer names in a specially formatted file that only I know how to read." Folks will convince themselves that this is okay:

```
# Specify three computer names
Get-CompanyStuff -Computername ONE,TWO,THREE

# Specify a file containing computer names
Get-CompanyStuff -FilePath ./names.dat
```

Recognize those? Yeah, it's the same file-reading pattern that we just said we don't like. "But `Get-Content` can't read my .DAT file," the argument goes, "so I'm not duplicating functionality." The argument misses the point: The "tools only do one thing" pattern has little or nothing to do with duplicating functionality; it has everything to do with simplicity. We'd use these patterns instead:

```
# Specify three computer names
Get-CompanyStuff -Computername ONE,TWO,THREE

# Specify a file containing computer names
Get-CompanyStuff -Computername (Get-SpecialDataFormat ./names.dat)

# Or again, if you were really smart...
Get-SpecialDataFormat ./names.dat | Get-CompanyStuff
```

The idea here is to take that "special data-format-reading stuff" and put it into its own standalone tool. Each tool then becomes simpler, easier to test by itself, easier to debug and maintain, and so on. Not to overplay the hammer analogy from chapter 7,

but if *we* were designing hammers, none of them would have the claw end for removing nails. That'd be a separate tool.

## 8.2 Tools are testable

Another thing to bear in mind is that—if you're trying to make tools like a real pro—you're going to want to create automated unit tests for your tools. We'll get into how that's done in chapter 20; but from a design perspective, you want to make sure you're designing tools that are, in fact, testable.

One way to do that is, again, to focus on tightly scoped tools that do just one thing. The fewer pieces of functionality a tool introduces, the fewer things and permutations you'll have to test. The fewer logic branches within your code, the easier it will be to thoroughly test your code using automated unit tests.

For example, suppose you decide to design a tool that will query a bunch of remote computers. Within that tool, you might decide to implement a check to make sure each computer is reachable, perhaps by pinging it. *That* might be a bad idea. First of all, your tool is now doing two things: querying whatever it is you're querying, but also pinging computers. That's two distinct sets of functionality. The pinging part, in particular, is likely to be code you'd use in many different tools, suggesting that it should be its own tool. Having the pinging built into the same querying tool will make testing harder, too, because you'll have to explicitly write tests to make sure that the pinging part works the way it's supposed to.

An alternate approach would be to write that Test-PCConnection functionality as a distinct tool. So, if your querying tool is something like Get-Whatever, you might concoct a pattern like this:

```
Get-Content computernames.txt | Test-PCConnection | Get-Whatever
```

The idea is that `Test-PCConnection` would filter out whatever computers weren't reachable, perhaps logging the failed ones in some fashion, so that Get-Whatever could focus on its one job of querying something. Both tools would then become easier to independently test, because each would have a tightly scoped set of functionality.

> **TIP** Really, having testable tools is a side effect of having tools that only do one thing. If you're being careful with your tool design and creating tightly scoped tools, you get all the benefits of more testable tools essentially for free.

You also want to avoid building functionality into your tools that will be difficult to test. For example, you might decide to implement some error logging in a tool. That's great—but if that logging is going to a SQL Server database, it will be trickier to test and make sure the logging is working as desired. Logging to a file might be easier, because a file would be easier to check. Easier still would be to write *a separate tool* that handles logging. You could then test that tool independently and *use it* in your other tools. This gets back to the idea of having each tool do one thing, and one thing only, as a good design pattern.

## 8.3 Tools are flexible

You want to design tools that can be used in a variety of scenarios. This often means wiring up parameters to accept pipeline input. For example, suppose you write a tool named Set-MachineStatus that changes some setting on a computer. You might specify a `-ComputerName` parameter to accept computer names. Will it accept one computer name, or many? Where will those computer names come from? The correct answers are, "Always assume there will be more than one, if you can," and "Don't worry about where they come from." From a design perspective, you want to enable a variety of approaches.

It can help to sit down and write some examples of using your command that you *intend to work*. These can become help-file examples later, but in the design stage they can help make sure you're designing to allow all of them. For example, you might want to be able to support these usage patterns:

```
Get-Content names.txt | Set-MachineStatus
Get-ADComputer -filter * | Select -Expand Name | Set-MachineStatus
Get-ADComputer -filter * | Set-MachineStatus
Set-MachineStatus -ComputerName (Get-Content names.txt)
```

That third example will require some careful design, because you're not going to be able to pipe an AD computer object to the same `-ComputerName` parameter that also accepts a `String` object from `Get-Content`! You may have identified a need for two parameter sets, perhaps one using `-ComputerName <string[]>` and another using `-InputObject <ADComputer>`, to accommodate both scenarios. Now, creating two parameter sets will make the coding, and the automated unit testing, a bit harder—so you'll need to decide whether the tradeoff is worth it. Will that third example be used so frequently that it justifies the extra coding and test development? Or will it be a rare enough scenario that you can exclude it and instead rely on the similar second example?

The point is that every design decision you make will have downstream impact on your tool's code, its unit tests, and so on. It's worth thinking about those decisions up front, which is why it's called the *design phase*!

## 8.4 Tools look native

Finally, be careful with tool and parameter names. We went over this in part 1, but it's worth repeating, because we see people get "creative" all the time. Tools should always adopt the standard PowerShell *verb-noun* pattern and should only use the most appropriate verb from the list returned by `Get-Verb`. Microsoft also publishes that list online (http://mng.bz/2vc8); the online list includes incorrect variations and explanations that you can use to check yourself. Don't beat yourself up *too* hard over fine distinctions between approved verbs, like the difference between Get and Read. If you check out that website, you'll realize that `Get-Content` should probably be `Read-Content`; it's likely a distinction Microsoft came up with *after* `Get-Content` was already in the wild.

We also recommend that you get in the habit of using a short prefix on your command's noun. For example, if you work for Globomantics, Inc., then you might design commands named Get-GloboSystemStatus rather than just Get-SystemStatus. The prefix helps prevent your command name from conflicting with those written by other people and it will make it easier to discover and identify commands and tools created for your organization.

> **NOTE** One reason we went on about native patterns in part 1 of this book is that they're so important. Don't ever forget that the existing commands, particularly the core ones authored by the PowerShell team at Microsoft, represent their vision for how PowerShell works. Break with that vision at your own peril!

Parameter names should also follow native PowerShell patterns. Whenever you need a parameter, take a look at a bunch of native PowerShell commands and see what parameter name they use for similar purposes. For example, if you needed to accept computer names, you'd use `-ComputerName` (notice it's singular!) and not some variation like "MachineName". If you need a filename, that's usually `-FilePath` or `-Path` on most native commands.

### The verb quandary

One area where you can get a bit wound up is in choosing the right verb for your command name. Honestly, Microsoft probably has too many verbs to choose from, and although we're sure someone in the company had a clear idea of the differences among them all, that hasn't always been well-communicated to the PowerShell public. For example, if you're writing a command that will retrieve information from a SQL Server database, is the command name Get-MyWhateverData, or is it Read-MyWhateverData? The company offers some guidance, stating, "The `Get` verb is used to retrieve a resource, such as a file. The `Read` verb is used to get information from a source, such as a file." This implies `Get` would be used to *get a file*, meaning an object representing the file itself, whereas `Read` would be used to retrieve *the contents of the file*. Except that `Get-Content` is a thing, so Microsoft didn't even take its own advice.

Our advice? Do what seems to be the most consistent with whatever's already in PowerShell. If you're truly stuck, post a question in the forums at Powershell.org to get a little feedback from experienced pros.

## 8.5 For example

Before we even start thinking about design decisions, we like to review the business requirements for a new tool. We try to translate those business requirements to usage examples so it's clearer to us how a tool might be used. If other stakeholders are involved—such as the people who might consume this tool, once it's finished—we get them to sign off on this functional specification so that we can go into the design

phase with clear, mutual expectations for the new tool. We also try to capture *problem statements* that this new tool is meant to solve, because those sometimes offer a clearer business perspective than a specification that someone else may have written.

We have a lot of different computers deployed in our company, which have different hardware vendors, different versions of Windows, different configurations, and so on. When users call the help desk, it's often difficult for the technicians to figure out what kind of computer they're dealing with. Users aren't always aware of details like model numbers, OS versions, installed RAM, and so on. We have a configuration management system the help desk can check, but it isn't always up to date or accurate. We'd like a tool that the help desk can use to quickly query a computer, if it's online, and get some key information about its OS and hardware configuration. In some cases, we have downtime and can query that information from multiple computers and double-check the accuracy of the configuration management system. The help desk can update that database if it needs updating.

### Be careful of context

When you start designing tools, it's fine to make business-level problem statements. That's a large part of what the design is for, after all! Statements like, "When users call the help desk, it's often difficult for the technicians to figure out what kind of computer they're dealing with," are fantastic.

Stating desired outcomes, such as when we wrote, "We'd like a tool that the help desk can use to quickly query a computer," is fine as well—it defines a business need. But it's *hugely important* that not every business statement be something you try to solve with a *single* tool or command. You may find that you need a suite of tools, which could be packaged as a module…but we're getting ahead of ourselves.

We've gone on at length about the need for tools to be as detached as possible from a particular context, yet our business statement has provided a very clear context: "We want technicians to query things." That context leads to certain assumptions, like, "The output needs to be human-readable," and maybe, "Our technicians aren't that experienced, so a GUI will be needed for them to operate this thing." This is good background information, but it doesn't mean you're going to solve it all with a single tool.

Our complete business statement kind of implies the creation of a tool to do the data retrieval, and perhaps a controller script to provide the help desk with an input/output interface. The *tool* doesn't need to worry about how the technician uses it or what the technician will see as a result; the *controller* can worry about those context-specific things and use the *tool* under the hood to get the data.

*Never* lose track of the tool/controller design pattern. Get used to reading business statements that will ultimately need tools *and* controllers, and understand which elements of a business solution will be best solved by each type of script.

Taking the last part of the previous sidebar to heart would lead us to some more detailed questions, asking for specifics about what the tool needs to query. Suppose the answer came back as follows:

- Computer host name
- Manufacturer
- Model
- OS version and build number
- Service pack version, if any
- Installed RAM
- Processor type
- Processor socket count
- Total core count
- Free space on system drive (usually C: but not always)

That's fine—we know we can get all that information somehow. We know we're going to write a tool, maybe called Get-MachineInfo, and it will probably have at least a `-ComputerName` parameter that accepts one or more computer names as strings. Thinking ahead, we might also start making notes for an `Update-OrgCMDatabase` command, which could consume the output of Get-MachineInfo and automatically update the organization's configuration management database. Nobody *asked* for that, but it's kind of implied in the business problem statements, and we can see them asking for it once we deliver the first tool—"Hey, because the tool gets all the data, is there any way we can have it just push that into the CM database?" We'll keep that in mind as we design the first tool—we want to ensure that the tool is outputting something that could be easily consumed by another command sometime in the future.

We'll assume that some computers won't respond to the query, and so we'll design a way to deal with that situation. We'll also assume that we have some old versions of Windows out there, so we'll make sure the tool is designed to work with as old a version of Windows as possible, as well as the latest and greatest.

Our design usage examples might be pretty simple:

```
Get-MachineInfo -ComputerName CLIENT
Get-MachineInfo -ComputerName CLIENTA,CLIENTB
Get-MachineInfo -ComputerName (Get-Content names.txt)
Get-MachineInfo -ComputerName (Get-ADComputer -id CLIENTA |
➥ Select -Expand name)
Get-Content names.txt | Get-MachineInfo
Get-ADComputer -id CLIENTA | Select -Expand name | Get-MachineInfo
```

The second chunk of examples will all require the same design elements, whereas the last chunk of examples will all be made possible by another set of design elements. No problem. The output of these should be pretty deterministic. That is, given a specific set of inputs, we should get the same output, which will make this a fairly straightforward design for which to write unit tests. Our command is only doing

one thing, and it has very few parameters, which gives us a good feeling about the design's tight scope.

### The beauty of usage examples in design

Stating usage examples as part of your tool design is a *wonderful* idea. For one thing, it helps you make sure you're not bleeding from *tool* design into *controller* design. If your usage examples start to take up 10 sheets of paper and look complicated, then you know you're probably not scoping your tool's functionality tightly enough, and you might be looking at several tools instead of just one.

Usage examples can also become part of your eventual help file. There's a school of thought that you should *start* tool design by *writing the help file*. The help file can then exist as a kind of functional specification, which you code to. Similarly, writing usage examples can help support *test-driven development* (TDD), in which you write automated tests *first*, to sort of specify how your tool should work, and *then* write the code.

Writing usage examples first can also help you avoid bad design decisions. If you're struggling to write all the examples you know you need, and you still keep coming up with an overly long or overly complicated list, then you know you're on the wrong track entirely. It might be worth sitting down with a colleague to try and refactor the whole project to keep it simpler.

We'd take that set of examples back to the team and ask what they think. Almost invariably, doing so will generate questions.

> *How will we know if a machine fails? Will the tool keep going? Will it log that information anyplace?*

Okay—we need to evolve the design a bit. We know that we need to keep going in the event of a failure and give the user the option to log failures to, perhaps, a text file:

```
Get-MachineInfo -ComputerName ONE,TWO,BUCKLE,SHOE
➥ -LogFailuresToPath errorlog.txt
```

Provided the team is happy with a text file as the error log, we're good including that in the design. If they wanted something more complicated—the option to log to a database or to an event log—then we'd design a separate logging tool to do all of that. For the sake of argument, though, let's say they're okay with the text file.

What about older computers? We know some machines use WMI and others will only take CIM. We thought about that, but we didn't make it explicitly clear in the design. And, to be fair, we could handle that situation entirely within the tool—but it could make the tool's performance slower if it had to repeatedly try WMI and then CIM for each computer. It might be better to design an option so that if the technician *knew* one or the other would work, they could just say so. We could still fall back automatically if we weren't told otherwise:

```
Get-MachineInfo -ComputerName PC1,PC2 -Protocol WMI -ProtocolFallback
```

We'll plan to default `-Protocol` to CIM and allow either WMI or CIM to be specified. By adding `-ProtocolFallback`, we'll always try the specified protocol first, but we'll *try* the other one on a per-computer basis if the first attempt fails. If `-ProtocolFallback` isn't specified, we'll *only* try the specified protocol, which will save time when the tool runs. There's no need at this stage to figure out *how* we'll do all that; right now, we're just designing the thing.

Let's say that the team is satisfied with these additions and that we have our desired usage examples locked down. We can now get into the coding. But before we do, why don't you take a stab at your own design exercise?

#### Designing sets of commands

The forgoing discussion is great when you're writing a command to do something self-contained, like retrieving management information from multiple computers. There's a slightly different discussion, however, when you start writing sets of commands to help manage a large system.

For example, suppose you want to write a set of commands to help manage a customer information-tracking application. What commands might you need to write?

Start by inventorying the *nouns* in the system. What are the things that the system works with? Users? Customers? Orders? Items in an order? Addresses? Write down that list somewhere.

Next, look at each noun and decide what the system can *do* with it. For users, what tasks does the system offer? Creating new ones? Removing them? Modifying existing ones? Listing them all? Those give you your verbs—*New*, *Remove*, *Set*, and *Get*, in this case, yielding commands like New-SystemUser, Remove-SystemUser, Set-SystemUser, and Get-SystemUser (assuming *System* is a useful prefix for your organization).

This little inventory exercise helps make sure you're not missing any key functionality. Having the command list doesn't automatically mean you're going to *write* all of those commands, but it does give you a checklist to prioritize and work against.

## 8.6 Your turn

If you're working with a group, this will make a great discussion exercise. You won't need a computer, just a whiteboard or a pen and paper. The idea is to read through the business requirements and come up with some usage examples that meet the requirements. We'll provide *all* the business requirements in a single statement, so that you don't have to "go back to the team" and gather more information.

### 8.6.1 Start here

Your team has come to you and asked you to design a PowerShell tool that will help them automate a repetitive, boring task. They're all skilled in *using* PowerShell, so they just need a command or set of commands that will help automate this task.

You've been lazy about changing service logon passwords. Many have been switched over to Managed Service Accounts, so you don't need to, but you have a lot of services—many of which run on multiple computers in a cluster—that haven't had a password change in years. The native `Set-Service` command doesn't do it. You'd like a tool that will let you change the logon user account as well as the password, for a single service, on one or more machines at once. If any machine fails, you need to know about it so you can handle it manually. Displaying onscreen and/or logging to a text file is fine.

This needs to run on a variety of Windows Server versions, so either WMI or CIM will work, but usually it's one or the other, not both. In most cases, the tech running this won't know if it has to be CIM or WMI, so the tool will need to handle it. CIM is probably more common right now, but you know you've got old WMI-only machines, too.

You don't usually need to script this, so the password can be provided in clear text on the command line as a parameter. You'd like the command to output something no matter what happens—such as the name of each computer and whether it succeeded, the service it was changing, and the logon account the service is now using (whether that was changed or not). You'll usually want that output either onscreen, in a simple HTML report, or in a CSV file you can load into Microsoft Excel.

### 8.6.2 Your task

Your job is to design the tool that will meet the team's business requirements. You are *not* writing any code at this point. When creating a new tool, you have to consider who will use the tool, how they might use it, and their expectations. And the user might be you! The end result of your design will be a list of command usage examples (like those we've shown you), which should illustrate how each of the team's business needs will be solved by the tool. It's fine to include existing PowerShell commands in your examples, if those commands play a role in meeting the requirements.

**TRY IT NOW** Stop reading here, and complete the task before resuming.

### 8.6.3 Our take

We'll design the command name as Set-TMServiceLogon. The *TM* stands for *Toolmaking*, because we don't have a specific company or organizational name to use. We'll design the following use cases:

```
Set-TMServiceLogon -ServiceName LOBApp
                   -NewPassword "P@ssw0rd"
                   -ComputerName SERVER1,SERVER2
                   -ErrorLogFilePath failed.txt
                   -Verbose
```

Our intent is that `-Verbose` will generate onscreen warnings about failures, and `-ErrorLogFilePath` will write failed computer names to a file. Notice that, to make this

specification easier to read, we've put each parameter on its own line. The command won't *execute* exactly like that, but that's fine—clarity is the idea at this point:

```
Set-TMServiceLogon -ServiceName OurService
                   -NewPassword "P@ssw0rd"
                   -NewUser "COMPANY\User"
                   -ComputerName SERVER1,SERVER2
```

This example illustrates that `-ErrorLogFilePath` and `-Verbose` are optional, as is `-NewUser`; if a new user isn't specified, we'll leave that property alone. We also want to illustrate some of our flexible execution options:

```
Get-Content servers.txt |
➥ Set-TMServiceLogon -ServiceName TheService -NewPassword "P@ssw0rd"
```

This illustrates our ability to accept computer names from the pipeline. Finally

```
Import-CSV tochange.csv | Set-TMServiceLogon | ConvertTo-HTML
```

We're illustrating two things here. First is that we can accept an imported CSV file, assuming it has columns named ServiceName, NewPassword, ComputerName, and, optionally, NewUser. Our output is also consumable by standard PowerShell commands like `ConvertTo-HTML`, which also implies that `Format-` commands and `Export-` commands will also work.

### Big designs don't mean big coding

We usually create initial designs that are all-encompassing. That doesn't mean we immediately sit down and start implementing the entire design. In software, there's a difference between *vision* and *execution*.

We're just talking about PowerShell commands, so there's perhaps no need to go all philosophical on you, but this is an important point. You may have no desire right this minute to implement error logging in your command. Fine. That doesn't mean you can't *plan for it* to someday exist. Planning—in other words, having a *vision* for your code—means you can take that into account as you write the code you *do* need right away.

"You know, I have no plans to log failed computers right now, but I know I will someday. I'll go ahead and implement a code structure that'll be easier to add logging to in the future." Your execution today, in other words, doesn't have to be the entire vision. You can create your vision now and then execute it in increments as you have time and need.

POWERSHELL

# *Learn* POWERSHELL SCRIPTING IN A MONTH OF LUNCHES

Don Jones and Jeffery Hicks


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]