"PESTER IS AN IMPORTANT SKILL THAT EVERY POWERSHELL USER SHOULD MASTER"

| JEFFREY SNOVER, CREATOR OF POWERSHELL

THE PESTER BOOK

THE ALL IN ONE GUIDE TO UNDERSTANDING AND WRITING TESTS FOR POWERSHELL

BY: ADAM BERTRAM

The Pester Book

Adam Bertram

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

This version was published on 2021-09-02

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

Tweet This Book!

Please help Adam Bertram by spreading the word about this book on Twitter!

The suggested hashtag for this book is #PesterBook.

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

#PesterBook

Contents

About This Book

This is the 'Forever Edition' on LeanPub[1]. That means when the book is published as it's written and may see periodic updates. Although, at this time, this book is 100% complete. That is not to say it will never receive updates again. The updates will be sporadic, if at all.

If you purchased this book, thank you. Know that writing a book like this takes hundreds of hours of effort, during which I'm not making any other income. Your purchase price is important to keeping a roof over my family's heads and food on my table. I'm not rich–this income is important to us. Please treat your copy of the book as your own personal copy–it isn't to be uploaded anywhere, and you aren't meant to give copies to other people. I've made sure to provide a DRM-free file (excepting any DRM added by a bookseller other than LeanPub) so that you can use your copy any way that's convenient for you. I appreciate your respecting my rights and not making unauthorized copies of this work.

If you got this book for free from someplace, know that you are making it difficult for me to write books. When I can't make even a small amount of money from my books, I'm encouraged to stop writing them. If you find this book useful, I would greatly appreciate you purchasing a copy from LeanPub.com or another bookseller. When you do, you'll be letting me know that books like this are useful to you, and that you want people like me to continue creating them.

- Adam

---

[1] http://leanpub.com

Introduction

This book's code was developed and tested with Pester 4.7.3. Older (or newer) versions may have different capabilities and features.

In the world of software development, there are a few broad types of testing. The first is *integration testing*, which, very roughly speaking, is where you test a particular chunk of code by compiling your entire project and run the whole thing. A more granular approach is unit testing. Unit testing is where you test a particular chunk of code all by itself, feeding it various inputs and checking its outputs, in a standalone fashion. Unit testing tests *code* and while integration testing tests how that code works once deployed.

The general theory is something like this:

1. You perform *unit tests* as you code each unit of work, like a module or function. You try to make each chunk of code work as correctly as possible.
2. You then perform *integration tests* on the entire project. This can help catch problems that might only surface when different functions start to talk to each other. For example, if you forgot to use a last name with an apostrophe like "O'Shea" when testing the "Add a Customer" module, you might pass your unit tests, but the integration test might fail when testing "O'Shea."
3. If you made any fixes during integration testing, you run regression tests which is basically another round of unit testing to see if you introduced any new problems when making your fixes.
4. On some occasions, you may choose to write *acceptance tests* as well. These test the desired outcome of your script. For example, when creating an Active Directory user, the desired outcome isn't necessarily to create the actual user. It is just a step needed to get an employee, contractor, or whoever authenticated to the domain. An acceptance test for this feature might include consideration of how intuitive the process is, how quickly the app completes the task, and whether the UI matches the client's branding requirements.

Pester is the tool to use when performing any of these tests in the PowerShell world and, as you'll see in the coming chapters, to test infrastructure (no code needed).

Testing as Institutional Memory

Automated testing creates gates. It helps you not make the same mistake twice. Take that "O'Shea" example, where the presence of an apostrophe somehow breaks the code (a common occurrence in poorly-written database code). You might not catch that bug initially, but if you get into the habit

of coding a test for every bug you find, then you'll never forget to test for each one again. Instead of having to remember about all the weird little things, specific to your environment or situation, that you have to accommodate in your code, the tests *remember for you*. Should you leave the project, you leave a legacy of those memories, encoded as repeatable tests.

Testing Drives Better Modularization

Once you begin building tests, you'll soon find out many of your bad coding practices. You've stumbled upon another hidden benefit of automated testing. Automated tests are more manageable when your hunks of code are tightly scoped, and each does only one specific thing. Not incidentally, tightly scoped chunks of code are also easier to write, more comfortable to debug, easier to spread around a team, easier to re-use and easier to maintain.

Breaking your code into tightly scoped units of work makes *unit testing* more reliable.

Tests as Functional Specifications

There's a whole software development theory called *Test Driven Development*, or TDD, which posits, more or less, that you write your tests *first*. The tests serve as a kind of code-based functional specification for what the hunk of code is supposed to do, and developers can run unit tests as they go, to make sure they're getting the code right. I'm oversimplifying a lot here, but we'll dig in a bit as I go through Pester's paces. Because tests focus on *inputs* and *outputs*, they can indeed serve as a kind of "in this situation, this is what should happen" functional specification that can drive development.

Tests in Automated Build Pipelines

In an ideal world of Continuous Delivery and/or Continuous Integration, automated testing plays an important role. Imagine checking in a PowerShell module to a source code repository. That triggers the spin-up of a virtual machine which is loaded with your code and your Pester tests. Your tests run against your code, and if the tests all pass, the repo publishes your module to a gallery, where anyone can download and use it. Does that guarantee your code is bug-free? Well, no. But, if you *do* find a bug, you code up a new test to catch it, and you'll never have that particular bug again. Over time, you get closer and closer to the rainbow's end of perfect code.

Where We're Not Going

Before I dive in, it's essential to set the scope for this book. We're going to focus on Pester itselfâ€"the tool. As a necessary part of that, we're going to be scratching the surface of unit, integration, and

infrastructure testing methodologies. But, and here's an important point, this isn't a book *about* those methodologies. So while we're going to try and make some key recommendations, this book isn't going to make you a unit/infrastructure/integration methodology expert. There are tons of books and other resources on methodologies (you'll encounter acronyms like TDD, DRY, DAMP, and others) and techniques, and I encourage you to read up on the subject if you're interested. We're going to focus as tightly as possible on Pester itself, mechanically, without getting deep into how it fits into one approach or another.

Resources

You will find all external resources for this book in the pesterbookcode GitHub repository[2].

[2] https://github.com/adbertram/pesterbookcode

Executing a Basic Test

So you're ready to take the next step in PowerShell development by build Pester tests? That's great! To give you a peek at what you're in for in this book, let's go through what it takes to build a simple Pester test.

If the concepts covered in this chapter don't make sense now, don't worry, you'll get the most in-depth information on what you're doing here in the chapters to come.

Installing Pester

Pester is preinstalled on Windows 10 and later, and on Windows Server 2016 or later. Pester currently works back to PowerShell v2. If you have the PowerShell Package Manager[3] installed, you can use the `Install-Module` command to install Pester. If not, head to the project's GitHub page[4] to download it as a ZIP file. When you extract the ZIP file, Pester will need to be in your *C:Program FilesWindowsPowerShellModules* folder or any other folder in your'PSModulePath' environment variable. You can look at this environment variable by running `$env:PSModulePath`.

Also, when you download Pester, your browser might mark the downloaded ZIP file as potentially unsafe. If so, use the `Unblock-File` command to unblock the ZIP file before extracting it. You can also right-click the file, and select *Properties* from Windows Explorer to find an *Unblock* button.

Creating a Test

In your PowerShell console with Pester installed, you can quickly create a simple project. Pester provides a command called `New-Fixture` that scaffolds out a single PowerShell script and test file to work with. However, this command is now considered legacy and will be removed from future versions Pester. Don't worry, though, creating a simple test is straightforward.

To build a Pester test, you need, at a minimum, two files: a PS1 script which contains code to test and a test script file. Create a folder called *Pester101* somewhere, a PS1 script called *Pester101.ps1* and an associated test script file called *Pester101.Tests.ps1*. Below is some code to get that done for you.

[3] http://powershellgallery.com/
[4] https://github.com/pester/Pester

```
PS> New-Item -Path 'C:\Pester101' -ItemType Directory
PS> New-Item -Path 'C:\Pester101\Install-Pester.ps1' -ItemType File
PS> New-Item -Path 'C:\Pester101\Install-Pester.Tests.ps1' -ItemType File
```

Inside of *Install-Pester.ps1*, create a single function (you'll learn Pester *loves* functions), called `Install-Pester` and leave it blank. The function name can be anything.

```
1  function Install-Pester { param() }
```

Once you have the *Install-Pester.ps1* script created, now add the following code to the *Install-Pester.Tests.ps1* file at the top. The first three lines code aren't related to Pester, per se; it's merely a way to dot sourcing[5] the function inside of *Install-Pester.ps1* to make available inside of this test script. The *describe* block below that is where the magic happens.

```
1  $here = Split-Path -Parent $MyInvocation.MyCommand.Path
2  $sut = (Split-Path -Leaf $MyInvocation.MyCommand.Path).Replace(".Tests.", ".")
3  . "$here\$sut"
4
5  describe 'Install-Pester' {
6      it 'does something useful' {
7          $true | Should -Be $false
8      }
9  }
```

Congratulations! You've officially created your first Pester test!

Running the Test

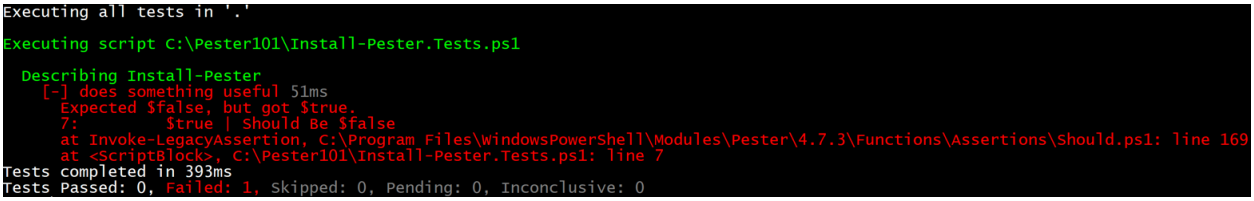
Now that you have a basic test scaffold built out let's run the test as-is and see what happens. At this point, you haven't put any code into the `Install-Pester` function in the *Install-Pester.ps1* file. The function does nothing.

If you're not already, change to the *C:Pester101* folder then run `Invoke-Pester`.

```
PS> cd C:\Pester101
PS> Invoke-Pester
```

When `Invoke-Pester` runs, it will automatically look for any files ending with *Tests.ps1* in the same folder you're in or any subfolders. If it finds one, it runs it.

[5] https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_scripts?view=powershell-6#script-scope-and-dot-sourcing



Failing Pester Test

Notice the `[-]` and red text. This indicator means that the test has failed. Below that indicator, it will tell you why. In this instance, it has failed because the boolean value `$false` was supposed to be `$true` which, as you probably know, is not valid.

Writing A Passing Test

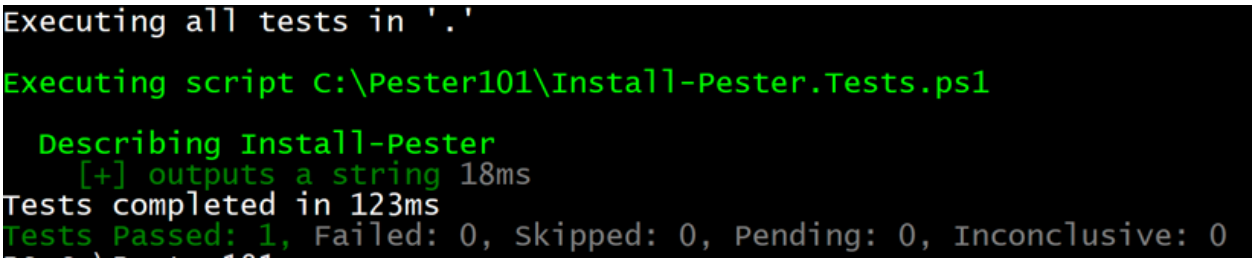
So you've failed but don't sweat, I'll help you pass this test. Open up the *C:Pester101Install-Pester.ps1* file in your favorite editor and insert a single line `Write-Output "Working!"` as shown below.

```
function Install-Pester {
    param()
    Write-Output "Working!"
}
```

Save the file and now open the *C:Pester101Install-Pester.Tests.ps1* file. Replace the entire *describe* section with the below code.

```
describe 'Install-Pester' {
    it 'outputs a string' {
        Install-Pester | Should -Be 'Working!'
    }
}
```

Now, run `Invoke-Pester` again while in the *C:Pester101* folder and see what happens.



Passing Pester Test

Notice now that your test has passed indicated by the `[+]` and green text.

How this Worked

You've now successfully gone through all there is to create and run tests in Pester, but how did that work anyway?

1. You created two files; *Install-Pester.ps1* and *Install-Pester.Tests.ps1* in the *C:Pester101* directory.

   The *Install-Pester.ps1* script represented the code to test. The *Install-Pester.Tests.ps1* script held the tests to run against the code in *Install-Pester.ps1.*
2. When `Invoke-Pester` was run while inside of the *C:Pester101* directory, it looked for all scripts ending in *.Tests.ps1.*
3. Once `Invoke-Pester` found all of the scripts, it then ran each one (in this case, a single one).
4. When the *Install-Pester.Tests.ps1* test script was run, it dot sourced the *Install-Pester.ps1* script so that Pester could run the `Install-Pester` function inside.
5. Pester then executed that function inside of the test (doing nothing at all the first time) and then compared the boolean value `$true` with the boolean value `$false`. Since the test was expecting `$true` to be `$false`, it failed.
6. On the second round, Pester dot sourced the `Install-Pester` function again, ran it and this time captured the output of `Working!`. It then compared that output to the expected output of `Working!`. Since they were both the same, the test passed.

Summary

We just covered what a basic Pester test looks like. Of course, you're going to get a lot more involved in this book covering real-world scenarios and in-depth tutorials. Strap in and get ready for a full soup to nuts tour of testing PowerShell scripts using the Pester testing framework!

Tagging

Pester has a concept called *tags*. *Tags* apply to `describe` blocks and serve a variety of purposes. When designing tests, using *tags* can be a beneficial way to call certain groups of tests at once, or to exclude specific tests, or to change the test in other ways.

Introduction to Tagging

Perhaps you have lots of `describe` blocks scattered across a single test script. Since they are already in a single test script, there's already some level of organization. However, *tags* allow for a different level of categorization than just grouping tests inside of a single script.

Without *tags*, when tests are in a single test script, you've only got two options for calling them. You can either run them all by simply executing the `Invoke-Pester` command and passing it the script path (`Invoke-Pester -Path 'C:\Tests.ps1'`), or you could run a single test at a time in that script using the `TestName` parameter on the `Invoke-Pester` command (`Invoke-Pester -Path 'C:\Tests.ps1' -TestName 'foo'`). You have no way to run a subset of tests or to exclude sets of tests.

For example, you have four `describe` blocks in a single test script that represent four functions. Each function is built around an object called *Thing* and is saved as *Thing.Tests.ps1*.

You're working with a single object with multiple functions that perform various actions on that object.

```
describe 'New-Thing' {

}

describe 'Set-Thing' {

}

describe 'Get-ThingAttribute' {

}

describe 'Get-Thing' {

}
```

You've already got these scripts in a single test file which lets us invoke all the tests at once. But, in some situations, you'd rather not do that. Instead, you only want to invoke the tests that may modify things on the system, or those that are less intrusive and just read things. All kinds of reasons exist to group tests together.

For example, you can assign a *tag* to each `describe` block representing the action of each of those functions. Below, you've decided to use the *tag* `Modifications` to indicate a test for a function that changes things and `ReadOnly` for a test for a function that reads things.

```
describe 'New-Thing' -Tag 'Modification' {

}

describe 'Set-Thing' -Tag 'Modification' {

}

describe 'Get-ThingAttribute' -Tag 'ReadOnly' {

}

describe 'Get-Thing' -Tag 'ReadOnly' {

}
```

Once you have the `describe` blocks tagged, you can now use the `Tags` parameter on the `Invoke-Pester` command to run only those tests for each type of function as shown below.

```
PS> Invoke-Pester -Path C:\Thing.Tests.ps1 -Tags 'Modification'
PS> Invoke-Pester -Path C:\Thing.Tests.ps1 -Tags 'ReadOnly'
```

The reverse action also applies to tagged `describe` blocks without modifying the tags themselves in any way. The only difference is how you call the `Invoke-Pester` command. Instead of using the `Tags` parameter, you'll use the `ExcludeTag` parameter to prevent specific tests from running.

```
PS> Invoke-Pester -Path C:\Thing.Tests.ps1 -ExcludeTag 'Modification'
```

The `Tags` parameter supports multiple *tags* as well so perhaps you have a different set of tags you'd like to apply to these functions too like `Database`, `Server`, whatever. Simply add another tag delimited by a comma, as shown below.

```
describe 'Get-Thing' -Tag 'ReadOnly','Database' {

}
```

Tagging Strategies

Because tagging can serve a multitude of purposes, let's cover some examples of how others are using *tags*. I hope this section will give you some good examples of how you can use *tags* in your tests and help you come up with some tagging strategies of your own.

To use the *tags* in a systematic and structured way, I recommend first defining all of the *tags* you will be using. Setting tags upfront is especially crucial if you're working on a team. Store available tags in a text file, spreadsheet, database, or some other place and stick to only using those tags. There's no built-in way to manage *tags* in Pester so you'll have to build your own simple storage and retrieval system.

General Strategies

Although some roles will have specific examples of tagging strategies, many of them have a standard set of *tags* that can be applied across any technology position. These *tags* can be used for just about any particular role.

Some examples of general tagging strategies are:

- By Status:
  - *Disabled*
- By Action:
  - Read
  - Modify
  - Remove/Create
- By Operating System:
  - Windows 7
  - Windows Server 2019
  - Ubuntu Linux
  - MacOS
- By Time:
  - NighlyBuild
  - AdHoc

For DevOps

Perhaps you're on a software development team or in charge of automating builds and software releases. You have a continuous integration/continuous deployment (CI/CD) pipeline set up, and you're using Pester tests to perform checks against test, QA, or production environments when the software is deployed. In a situation like this, you can define tags by a particular stack.

Maybe the software you're deploying is a web application that allows users to log into an application with a backend database of some kind. Some useful tags might be:

- By Stack:
  - *Database*
  - LoadBalancer
  - WebServer
- By Feature:
  - NewUser
  - FeatureX
  - FeatureY
- By Release:
  - v51
  - v4
- By Deployment Stage:
  - Dev
  - UA
  - QA
  - Prod

For Sysadmins

Maybe you're a system administrator and are using Pester to ensure your infrastructure is at a consistent state, no changes are being made without your consent. In this case, I'd argue for the point of a configuration management tool, but Pester tests are better than nothing!

Some example tags might be:

- By Rights:
  - RequiresAdminPrivileges
  - RequiresDomainAdmin
- By Server Type:
  - *ActiveDirectory*
  - *FileServer*
- By Server Component:
  - *CPU*
  - *Memory*
- By Cloud:
  - Azure
  - AWS
  - Google Cloud

For DBAs

For database administrators, Pester tests can provide significant value too. Since a Pester test is just a PowerShell script, you can write a test to check for anything you can read with PowerShell. For DBA, that could be:

- By Database Vendor:
  - MicrosoftSQLServer
  - MySQL
  - PostGres
- By Database Metrics:
  - PagingSpeed
  - ReadSpeed
  - Size

The options are limitless when it comes to coming up with *tags* to use. Remember to first come up with a pre-defined set of tests you and your team decide to use ahead of time and then assign the tags to your `describe` blocks and see what works for you.

Summary

Tagging is a simple but time-saving element of Pester. If you can assign *tags* in an organized fashion and track when specific tagged tests should be run, this tactic will save you and your team a ton of time over the long run. Tagging becomes more critical with the more tests you continue to create.

But Wait... There's More

This sample includes just a couple of chapters from the book, but in the book you'll also find:

Part 1: Pester Concepts

In this part of our book, we're going to focus on the concepts. We'll be covering each component of Pester from a basic, 101-level perspective. Each chapter is broken down by an important area of Pester. This part will introduce you to Pester and all that comes with it to prepare you for the next part where then take that knowledge and begin applying it to practical examples.

Part 2: Using Pester

In Part II, we're going to get down to business. By now, you've graduated from the 101-level information and learned what Pester is and how to use it in a basic manner. It's now time to get down to applying that knowledge to real-world use cases.

This part will be mostly hands-on demonstrations of how to use Pester in various scenarios. We'll cover how to use Pester to solve real problems, go over best practices and show you how to go from a Pester newbie to a Pester rockstar!

Part 3: Hands-On Design and Testing

In Part III, you're an old hand at building Pester tests. At this point, you should be well aware of how to build Pester tests in real-world scenarios and know many tricks of the trade. However, knowing how to *build* Pester tests and how to *design* them along with the code being tested are two separate things.

In this part, you're going to take a step back and cover how to build testable code to more easily work with Pester and finally how to design your Pester tests to be easy to manage.

Part 4: Pester Cookbook

If you need more examples of how to use Pester to test all kinds of scenarios, this is part of the book is for you. In this part, you'll find different testing scenarios Pester applies to.

In this part, you'll find various "recipes" that'll help you cook up the best Pester tests!