Get started with Pester (PowerShell unit testing framework)

Some time ago I stumbled upon the Pester framework that promised I would be able to test my scripts. This seemed to be super-useful for my day-to-day scripting, but unfortunately the learning curve was a bit steeper than I thought it would be. It took me reading a book on Test-Driven-Development (TDD) to understand what the ideas behind Pester are, and that they are actually pretty simple. Armed with this knowledge I was finally able to put Pester in use, and found it useful through the whole life cycle of my scripts. To help you with the first steps, I wrote a short series of articles that describe the basics of Pester.

**What is Pester?**

Pester is a unit testing framework for PowerShell. It provides a few simple-to-use keywords that let you create tests for your scripts. Pester implements a test drive to isolate your test files, and it can replace almost any command in PowerShell with your own implementation. This makes it great for both [black-box](http://en.wikipedia.org/wiki/Black-box_testing) and [white-box](http://en.wikipedia.org/wiki/White-box_testing) testing. Pester is best used with [TDD](http://en.wikipedia.org/wiki/Test-driven_development) approach to development.

For those of you who are not familiar with TDD, let me sum it up briefly: TDD is the opposite of the traditional approach to development. Traditionally you write your functional code first and then you write tests for it. You are progressing from implementation (how the code is written) to specification (how it should work). TDD is turning this around, so you progress from specification to implementation. First you define how the code should work and then you write the code. This is a pretty powerful concept.

In TDD you write the specification in the form of tests. Before you add any new feature you write a set of tests for it. You run your test suite and make sure all the new tests fail. This shows that there is a feature missing. You then develop the feature, testing it frequently. Once all your tests (including the new ones) pass you know the new feature is finished. You also know you did not break anything else while adding it.

Do this for every feature, patch or any other change and you are always one test suite run from seeing if everything works as requested. That is the biggest benefit of TDD.

There are three more you might like:

* + TDD forces you think before you start scripting. If you are unable to write the tests, chances are you don’t fully understand the problem you are trying to solve.
  + Switching between projects and patching bugs becomes easier. With tests in place you don’t have to remember how exactly your components interact. You can focus only on the change you are making and validate the rest by running your tests.
  + Bugs do not reappear. Once write test for a bug and squash it will never get to your production code again.

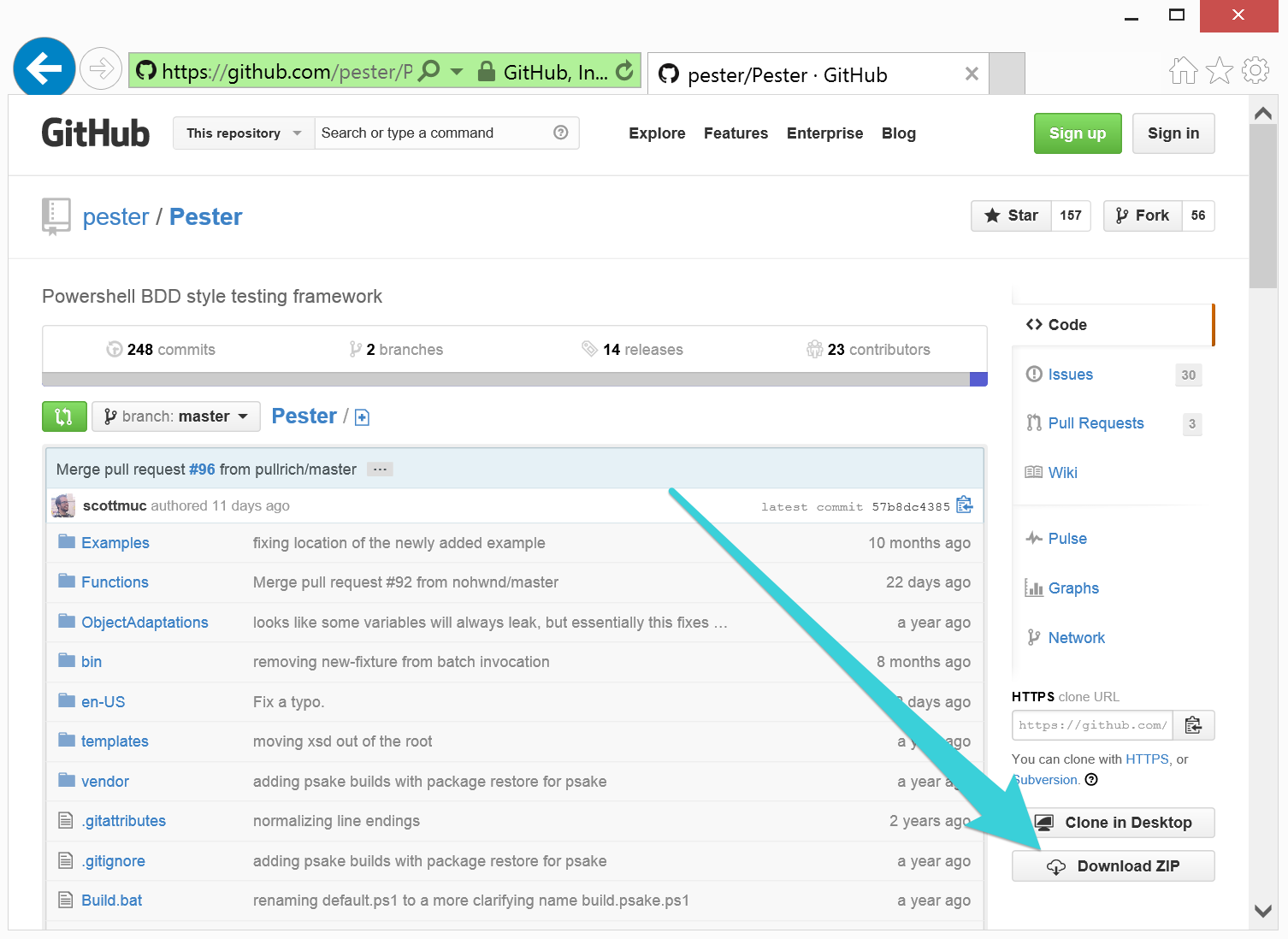
Pester brings these benefits to PowerShell and that is why I love using it.

**First steps**

**Downloading and installing Pester**

[*Pester*](https://github.com/pester/Pester) is a PowerShell module authored by [Scott Muc](http://scottmuc.com/) and improved by the community. It’s available for free on GitHub and installation is pretty easy. You just [download](https://github.com/pester/Pester) it, extract it to your Modules folder, and then import it to your PowerShell session.

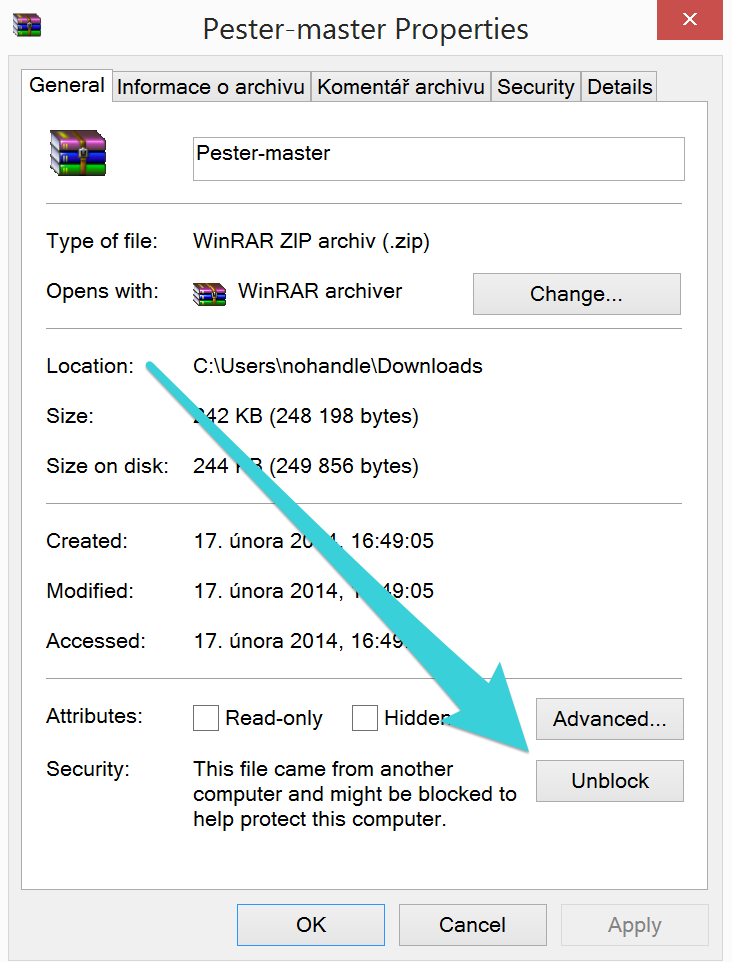
Let’s start by downloading it from the GitHub and extracting the archive into your Modules directory:

[](http://www.powershellmagazine.com/wp-content/uploads/2014/03/image0012.png)

If you used Internet Explorer to download the archive, you need to unblock the archive before extraction, otherwise PowerShell will complain when you import the module. If you are using PowerShell 3.0 or newer you can use the Unblock-File cmdlet to do that:

|  |  |
| --- | --- |
| 1 | Unblock-File -Path "$env:UserProfile\Downloads\Pester-master.zip" |

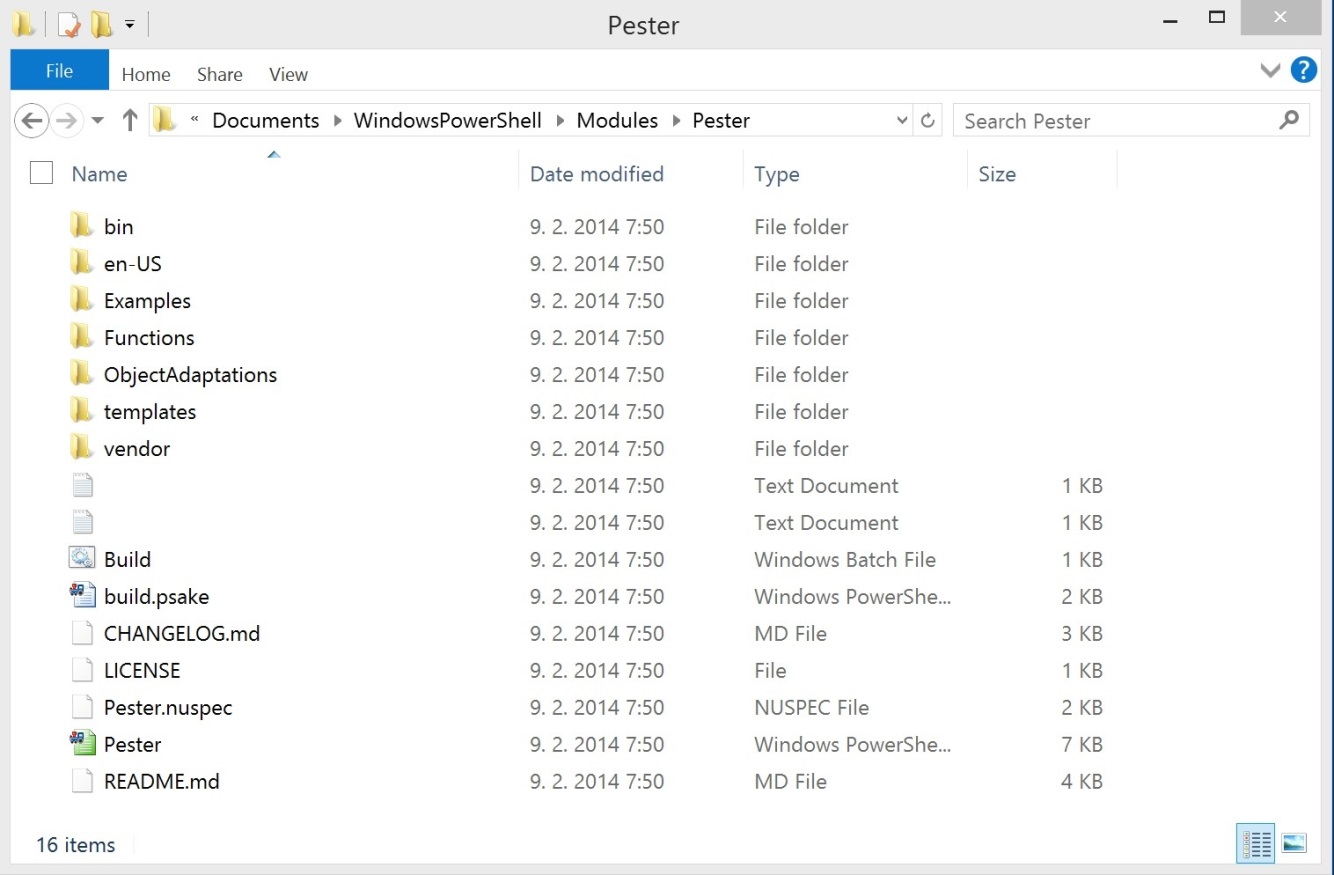
If you are using an older version of PowerShell you will have to unblock the file manually. Go to your Downloads folder and right-click *Pester-master.zip*. On the general tab click *Unblock* and then click OK to close the dialog.

[](http://www.powershellmagazine.com/wp-content/uploads/2014/03/image0032.png)

Open your Modules directory and create a new folder called Pester. You can use this script to open the correct folder effortlessly:

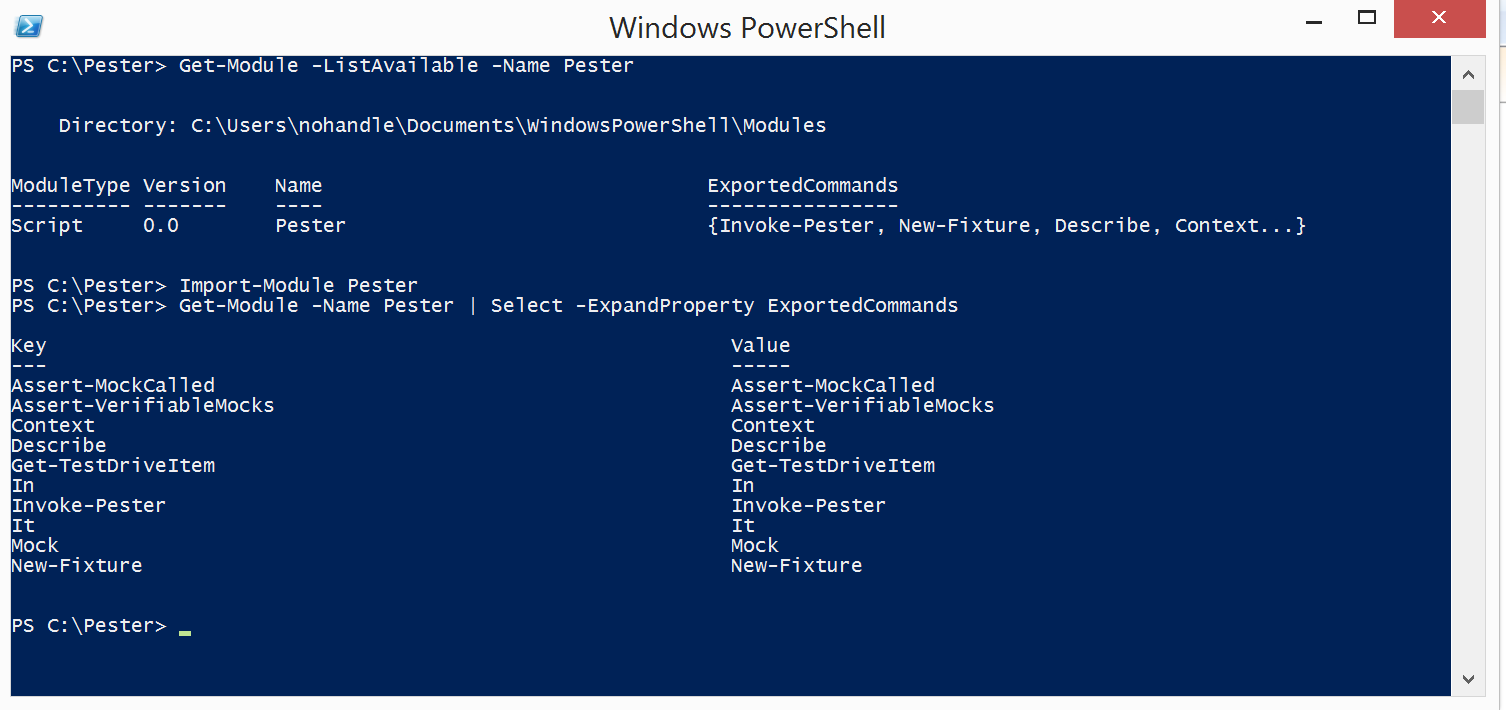
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | function Get-UserModulePath {        $Path = $env:PSModulePath -split ";" -match $env:USERNAME        if (-not (Test-Path -Path $Path))      {          New-Item -Path $Path -ItemType Container | Out-Null      }          $Path  }   Invoke-Item (Get-UserModulePath) |

Extract the archive to the Pester folder. When you are done you should have all these files in your Pester directory:

[](http://www.powershellmagazine.com/wp-content/uploads/2014/03/image005.jpg)

Start a new PowerShell session and import the Pester module using the commands below:

|  |  |
| --- | --- |
| 1  2 | Get-Module -ListAvailable -Name Pester Import-Module Pester Get-Module -Name Pester | Select -ExpandProperty ExportedCommands |

[](http://www.powershellmagazine.com/wp-content/uploads/2014/03/image007.png)

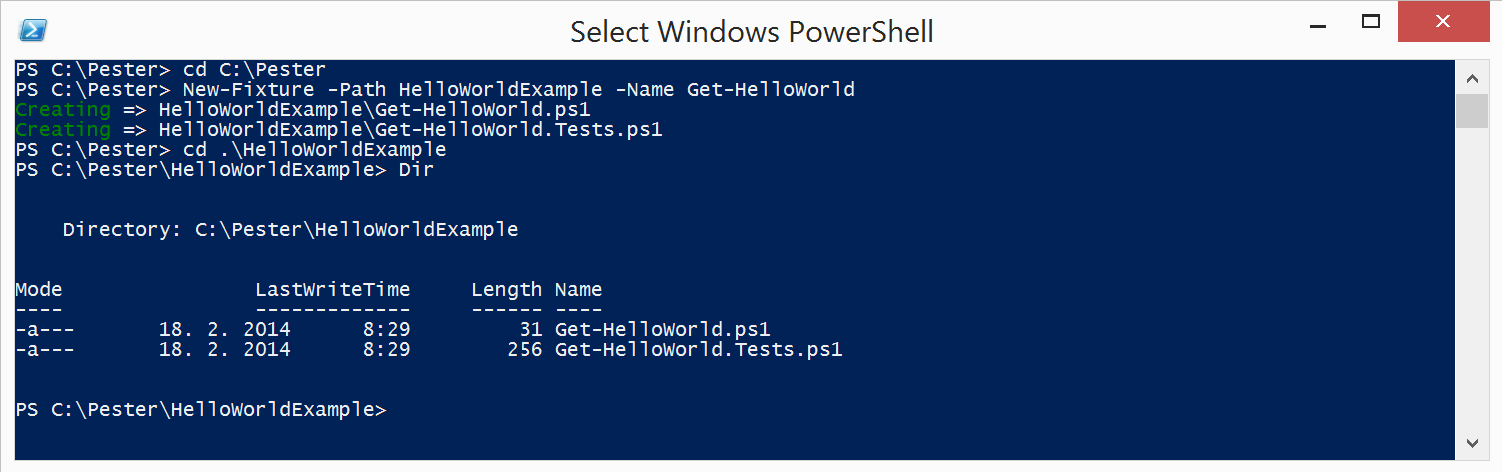
Next you will need a folder to play with. On my computer I created *‘C:\Pester’* for this purpose and you should do the same. If you choose another working directory, make sure you change the paths in the examples accordingly.

**Creating and failing our first test**

If you got this far I am sure you can’t wait to start testing. For the first example we’ll start with something extremely simple and create a function called *Get-HelloWorld*. The function will output a ‘Hello world!’ string when invoked and nothing more.

But before we start writing any code we should have a test in place. Creating a new test in Pester is easy. It contains a utility function *New-Fixture* to create the basic “scaffolding” for a test. Optionally it can create a separate folder to keep our working folder well-organized.

|  |  |
| --- | --- |
| 1  2  3  4 | cd C:\Pester New-Fixture -Path HelloWorldExample -Name Get-HelloWorld  cd .\HelloWorldExample  Dir |

[](http://www.powershellmagazine.com/wp-content/uploads/2014/03/image009.png)

The “scaffolding” created by the *New-Fixture* function consists of two files, and auto-generated code that links them. The first file *Get-HelloWorld.ps1* is the file where the production code is placed. The second file, *Get-HelloWorld.****Tests****.ps1*, is where the tests are placed. First take a look at the content of the *Get-HelloWorld.ps1* file.

|  |  |
| --- | --- |
| 1  2  3 | function Get-HelloWorld {    } |

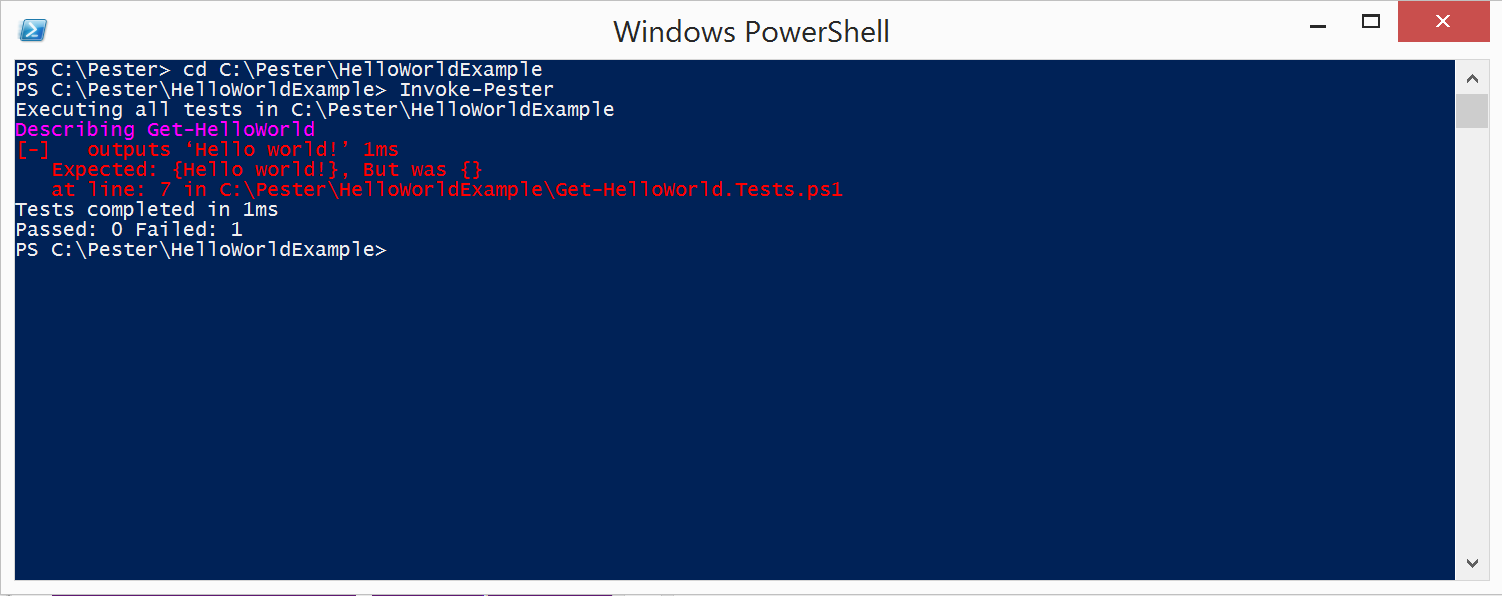
Nothing surprising except an empty function definition.

The content of the *Get-HelloWorld.Tests.ps1* file is way more interesting, a default test is waiting there:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | $here = Split-Path -Parent $MyInvocation.MyCommand.Path $sut = (Split-Path -Leaf $MyInvocation.MyCommand.Path).Replace(".Tests.", ".")  . "$here\$sut"    Describe “Get-HelloWorld" {      It "does something useful" {          $true | Should Be $false      }  } |

Before explaining how everything works, let’s do some testing first. First you need to change the test to reflect what the *Get-HelloWorld* function should do. Take the whole content of the tests file and replace it with this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | $here = Split-Path -Parent $MyInvocation.MyCommand.Path $sut = (Split-Path -Leaf $MyInvocation.MyCommand.Path).Replace(".Tests.", ".")  . "$here\$sut"    Describe "Get-HelloWorld" {      It "outputs 'Hello world!'" {          Get-HelloWorld | Should Be 'Hello world!'      }  } |

[](http://www.powershellmagazine.com/wp-content/uploads/2014/03/image011.png)

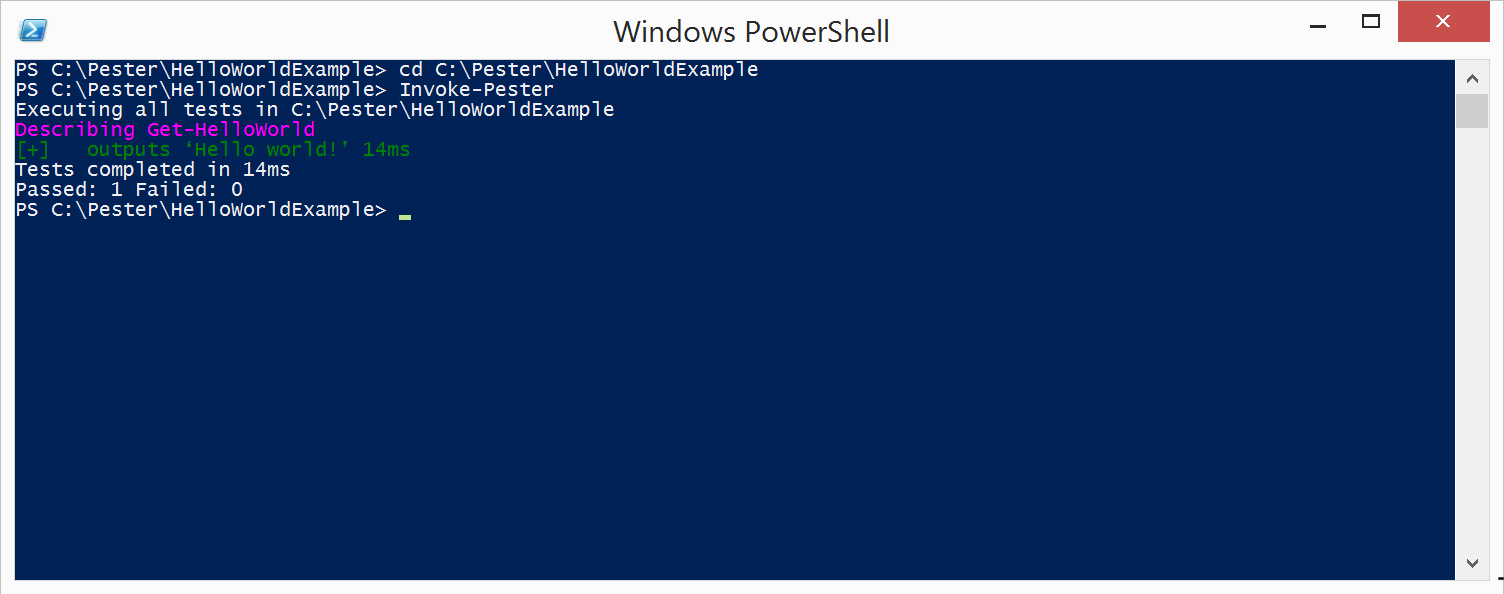
Now it is time to run the test and see if it passes. To do so, you will use another Pester function called *Invoke-Pester*. By default,*Invoke-Pester* runs all the tests in all the files in the current directory and its sub-

directories, and that is exactly what we need:

|  |  |
| --- | --- |
| 1  2 | cd C:\Pester\HelloWorldExample Invoke-Pester |

The output is red, and red means that the test failed. Don’t worry, remember what I said about TDD, you should always start with a failing test. Before we move on to implementing the function we should confirm the test failed for the right reason. In this case we are testing for specific output but the Get-HelloWorld returns nothing. Personally I would expect a message saying that there was no output but output was expected. And fortunately that is what the error message says:*“Expected: {Hello world!}, But was {}”*. The test fails correctly and we can finally implement the funciton. Open the *Get-HelloWorld.ps1* file and replace the empty definition with this:

|  |  |
| --- | --- |
| 1  2  3 | function Get-HelloWorld {     'Hello world!'  } |

[](http://www.powershellmagazine.com/wp-content/uploads/2014/03/image013.png)

Make sure you saved the file and run the test again:

|  |  |
| --- | --- |
| 1  2 | cd C:\Pester\HelloWorldExample Invoke-Pester |

The tests passed this time and that means the function is working! Congratulations, you just created, failed and passed your first Pester test!

**So what happened?**

Let’s start over and explain what actually happened step-by-step.

First you took the default test and replaced it with another one. Two things were different in these tests. The description of the *It* block and the code in the *It* block.

In Pester the *It* block represents one test. The description of the *It* block sums up what is tested, and the code inside the script block determines whether the test passes or fails. If you’d keep the script block empty the test would always pass. Such test is useless so you need a way to make the test pass or fail depending on a given condition. To do that Pester implements a set of keywords called *assertions*.

The assertion used in the default and the new test is *Should Be*. This assertion takes input from the pipeline and compares it with the expected value. You provide that value after the *Should Be* keywords. In the default test the *$true* is compared to *$false* and such test always fails.

In our test the output of the *Get-HelloWorld* function is compared to the expected value *‘Hello world!’*. So the result of the tests depends on the output of the *Get-HelloWorld* function. When we invoked the test for the first time *Get-HelloWorld* produced no output and hence the assertion failed with: *“Expected: {Hello world!}, But was {}”* message. We then changed the function implementation to produce the correct output and the test passed.

**Rest of the tests file**

In the tests file there are few more things worth noticing. As you can see the *It* block is placed inside a *Describe* block. The *Describe* block represents a group of tests and helps you keep your test file well-organized. It may seem useless now but it will prove more useful as your tests suite will grow. You can change the description of the *Describe* to whatever you like, but do not remove the *Describe* block entirely. Every tests file has to have at least one.

The *Describe* keyword, has more to it than grouping your tests. You use it to separate scopes for [*TestDrive*](https://github.com/pester/Pester/wiki/TestDrive), [*Mock*](https://github.com/pester/Pester/wiki/Mock) and you can even use its description to run a set of tests. All these capabilities will be covered later. Now it is enough to remember that every test file has to have at least one *Describe* block and that the *It* blocks are placed inside the *Describe* block.

The last thing in the tests file we did not cover are the top three lines of it. These lines link your tests file to your production code file. That is why the *Get-HelloWorld* function can be used in your tests file even though it is defined elsewhere.

The first line sets the *$here* variable to the path where the tests file is placed. The second line takes the name of the tests file, and by removing the “.Tests.” from the name it gets the name of the code file. The third line joins these two pieces of information together to form a full path to the code file and dot-sources it. Dot-sourcing the code file acts as if you copied the whole content of it and pasted it to the test file. As a result the *Get-HelloWorld* function can be used inside the tests file.

**Summary**

This covers the basics of testing with Pester. In the next article we will take a detailed look on more Pester keywords and all the assertions available. Using *TestDrive* and Mock will follow. If you found this article interesting, you missed something, or you disagree, please share your opinion in the comment section.