Mastering PowerShell Testing with Pester

Mark Wragg

DevOpsGroup

System Frontier

eSynergySolutions

TECHSNIPS

About Me

```
Describing Speaker
  [-] Error occurred in Describe block
    Expected: 'The foremost expert in Pester'
    But was:  'Mark Wragg'
```

Twitter : @markwragg
Website : http://wragg.io
GitHub : http://github.com/markwragg

DevOps Engineer, XE.com

2018

About Pester

- A PowerShell Module for testing PowerShell code
- Pester is written in PowerShell
- Pester is used to test PowerShell Core
- Pester is used to test.. Pester
- Install it via:

```
Install-Module Pester
```

- On Windows 10 its pre-installed, but you can update it via:

```
Install-Module Pester -Force -SkipPublisherCheck
```

Testing Strategies

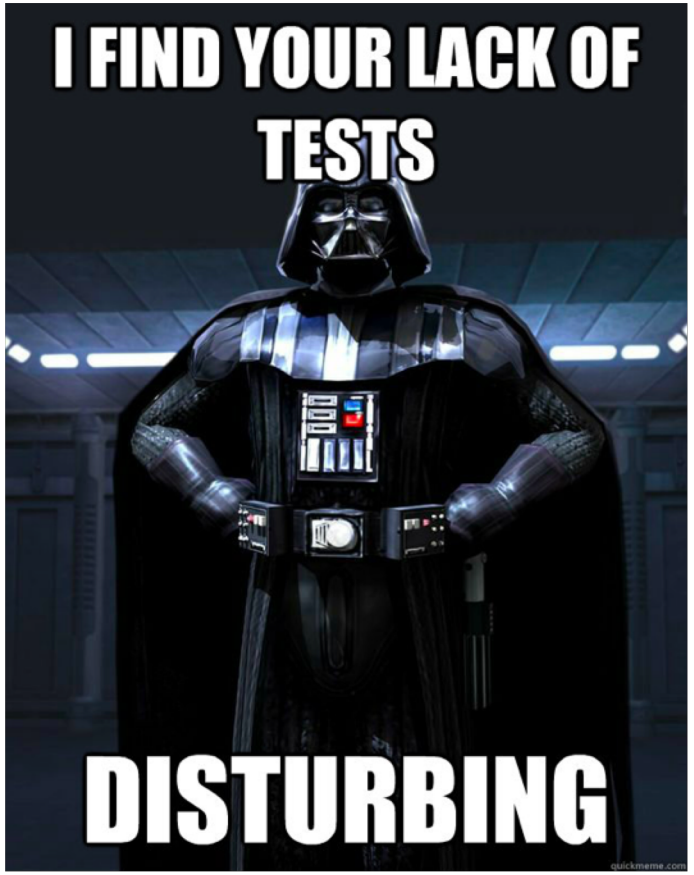
Pester is a versatile tool that can be used for a variety of testing strategies.

- Operational Validation:
  - System / infrastructure is configured / working as expected
- Integration Testing:
  - The end result of your code is correct / your code works with other modules.
- Unit Testing:
  - A unit of code – does one thing

Why should you write tests?

Testing is the single most important thing you will ever do.

- You are already testing your code, Pester allows that testing to be more detailed and thorough.
- Testing is an essential prerequisite to continuous integration / continuous delivery (CICD). You need tests that are run automatically when your code changes.
- Tests act as a form of documentation (but shouldn't be exclusively so).



You **are** a developer.

Beware of imposter syndrome.

Developer

[dih-**vel**-*uh*-per]

noun

An organism capable of converting caffeine in to code.

2018

PowerShell Code Maturity

Pester needs to be able to read your code and control when parts execute for it to be testable.

- Console commands (one liners)
- Basic scripts (one or more commands saved as a .ps1)
- Scripts that include functions but that are executed within the same script.

Not testable

- Scripts that define only functions
- Module with functions all declared as a single .psm1 file
- Module with functions organised in individual files
- Module with manifest, published in the PowerShell Gallery
- Module automatically tested and published via a CICD Pipeline

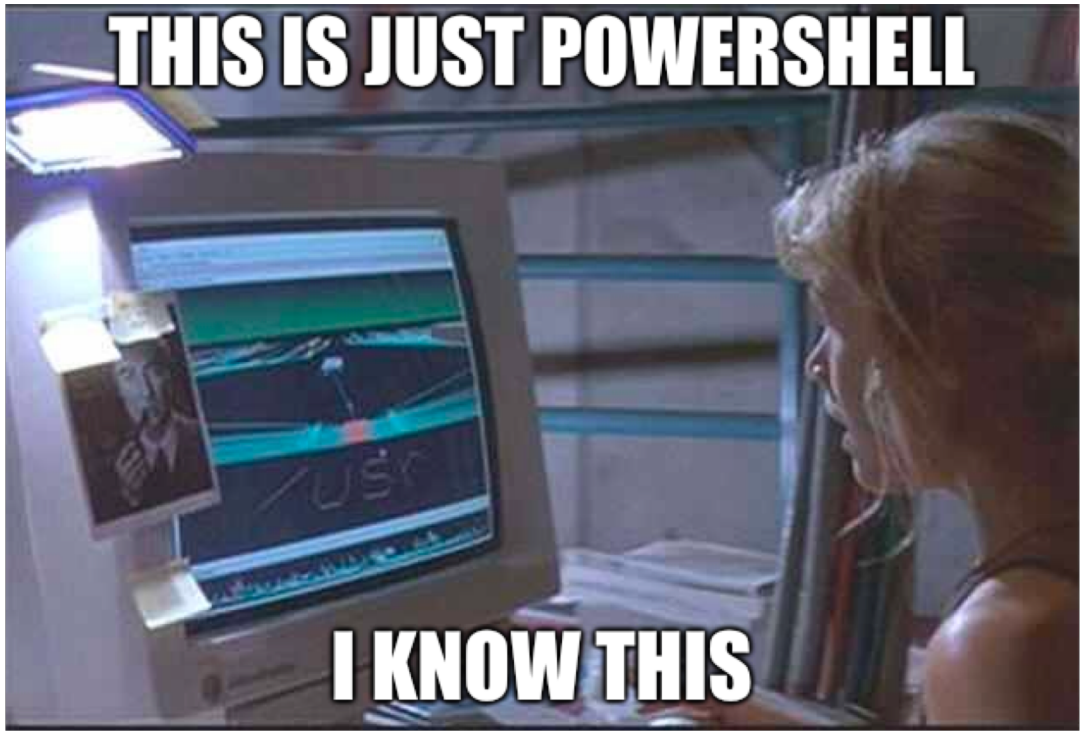
Testable

2018

Getting Started

A Pester script is just a PowerShell script with some additional keywords and structure.

- `Describe`
- `Context`
- `It`
- `Should`
- `Mock`



A simple example


```powershell
function Add-One ([int]$Number) {
    $Number + 1
}

Describe 'Add-One tests' {
    It 'Should add one to a number' {
        Add-One 1 | Should -Be 2
    }
    It 'Should add one to a negative number' {
        Add-One -1 | Should -Be 0
    }
}
```

2018

A more realistic example

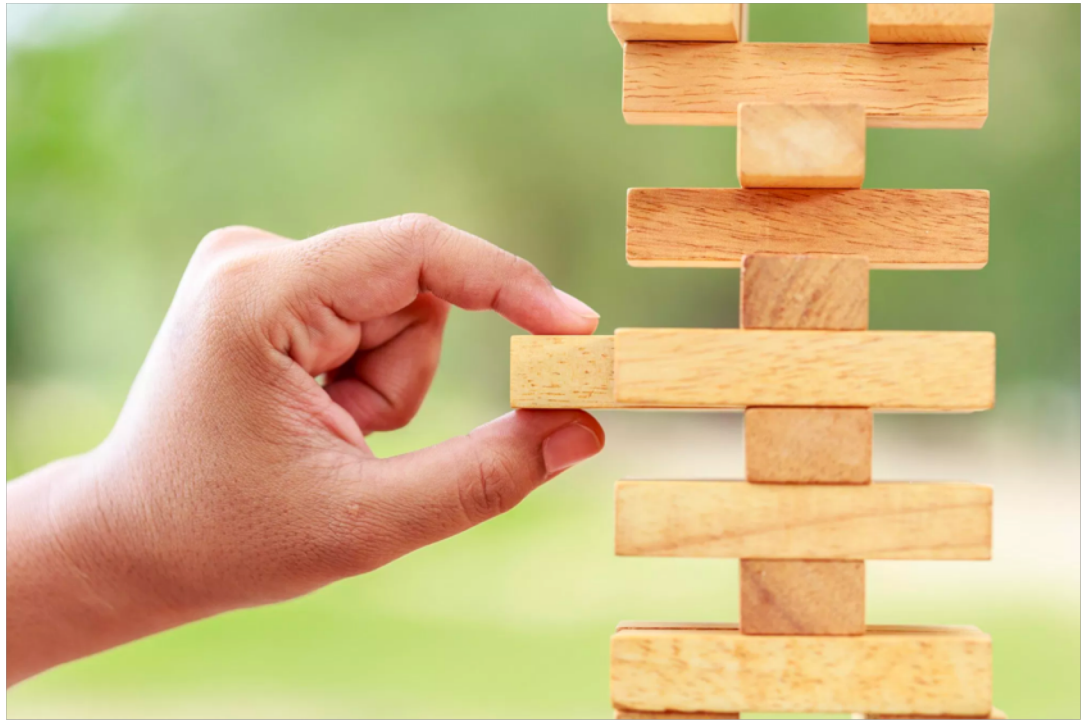

```
function Remove-TempFile ($Path) {
    $TempFiles = Get-ChildItem "$Path/*.tmp"
    $TempFiles | Remove-Item
}

Describe 'Remove-TempFile tests' {
    $TestFile = New-Item /tmp/test.tmp

    It 'Should return nothing' {
        Remove-TempFile -Path /tmp | Should -Be $null
    }
    It 'Should remove the test file' {
        $TestFile | Should -Not -Exist
    }
    It 'Should not return an error' {
        { Remove-TempFile -Path /tmp } | Should -Not -Throw
    }
}
```

This is also an example of the Arrange - Act - Assert testing pattern.

Refactoring

Refactoring is the process of changing code without changing its behaviour.

- Breaking your code in to smaller pieces (functions) so that each does just one thing.
  - Unit tests are about proving correctness.
  - A Unit of code should do only one thing.
  - A Unit test allows you to prove it does that one thing correctly.



A refactored example


```powershell
function Get-TempFile ($Path) {
    Get-ChildItem "$Path/*tmp*"
}

function Remove-TempFile ($Path) {
    $TempFiles = Get-TempFile -Path $Path
    $TempFiles | Remove-Item
}

Describe 'Get-TempFile tests' {
    New-Item TestDrive:/test.tmp
    New-Item TestDrive:/tmp.doc

    It 'Should return only .tmp files' {
        (Get-TempFile -Path TestDrive:/).Extension | Should -Be '.tmp'
    }
}
```

2018

About TestDrive

TestDrive is a PSDrive for file activity limited to the scope of a Describe or Context block.

- Creates a randomly named folder under $env:temp which you can then reference via TestDrive: or $TestDrive
- $TestDrive is the full literal path (useful if you need to do path comparisons).
- Use this as a safe space in which to do file operations, that is cleaned up automatically for you.
- Pester will keep a record of the files present in the location before entering a Context block and will remove any files therefore created by the Context block, but won't revert files that have been modified.

Mocking

Mocking is a facet of unit testing that involves replacing real cmdlets or functions with simulated ones.

This can be useful for a number of reasons:

1. Your code relies on missing external resources.
2. Your code makes destructive or other permanent or undesirable changes.
3. The code you are testing is using some secondary function that already has its own set of tests.

A mocking example


```
Describe 'Remove-TempFile tests' {
    Mock Remove-Item { }

    New-Item TestDrive:/test.tmp
    New-Item TestDrive:/tmp.doc

    $TestResult = Remove-TempFile -Path TestDrive:\

    It 'Should return nothing' {
        $TestResult | Should -Be $null
    }
    It 'Should call Remove-Item' {
        Assert-MockCalled Remove-Item -Times 1 -Exactly
    }
}
```

Mocking cmdlets that aren't present

Pester needs the cmdlet or function you are mocking to exist on the current system, but you can fake it if it doesn't.

- If you attempt to Mock a cmdlet that isn't currently available (e.g in a module not installed on your system) Pester will throw an error.
- You can work around this by declaring a fake function in its place.
- You can then still also declare a Mock, so that you can use Assert-MockCalled.
- This can get complex, so use only when needed. If you can make the required cmdlets/functions available that is more ideal.

Code Coverage

Code coverage is a measure that indicates how much of your code was executed when your tests ran.

- Code Coverage is a useful measure of how thorough your tests are.
- 100% code coverage does not equal 100% working code.
- Getting close to 100% is good, but achieving 100% code coverage isn't always useful.

Why is code coverage reporting useful?

Your code likely has multiple happy (successful) and unhappy (fail, error, exception) paths.

- If your code contains any logical statements (if, switch, try..catch etc.) then there are multiple ways in which it can be executed to either a successful or unsuccessful conclusion.
- The code coverage report helps to highlight which of these paths have not been taken.

How to use the code coverage feature of Pester

The Pester code coverage feature requires PowerShell version 3.0 or greater.

- To check code coverage, you need to execute pester with the –CodeCoverage parameter and send it a list of code files that you are evaluating:

```
Invoke-Pester -CodeCoverage (Get-Childitem .\YourModule\*.ps1 -Recurse)
```

Summary

Writing, maintaining and executing tests is an essential and accessible PowerShell skill.

- Pester is just PowerShell with a few extra commands to learn.
- There is an excellent and detailed Wiki on the official Pester repository that explains all the concepts:
  - https://github.com/pester/Pester/wiki
- There are lots of people in the community ready and willing to help you:
  - The #Pester tag on StackOverflow.com.
  - PowerShell.org forums.
  - r/PowerShell
  - The PowerShell Slack community.

PSDAY.UK

2018

Any questions?

Twitter : @markwragg
Website : http://wragg.io
GitHub : https://github.com/markwragg

PSDAY.UK 2018

Thank you!

DevOpsGroup

System Frontier

eSynergySolutions

TECHSNIPS