Common mistakes in Pester tests

@nohwnd
@pspester
me@jakubjares.com

Tested in PowerShell before it was cool.

Owner of Pester, developer, talks about testing to anyone who listens.

2018

Common mistakes in Pester tests

Jakub Jareš

Agenda

- Show mistakes I did myself
- Show problems others commonly encounter

- Show how to avoid them

- Make your test code better
- Make your tests give you more value

Presntation structure

- What I show is not a prescription, just a suggestion.
- Decide for yourself what brings value.

- Go from actionable items to more abstract concepts.

```
# Anyone not familiar with Pester?



# Get-Planet.ps1
function Get-Planet {
    @()
}



# Get-Planet.Tests.ps1
Describe 'Get-Planet' {
  It "Given no parameters, it lists all 8 planets" {
      Get-Planet | Should -HaveCount 8
  }
}
```

PowerShell Conference EU 2018

Tests that never fail

```
# A reinforced test

Describe "Get-Weather" {
  It "is always sunny in Prague" {
    try {
      Get-Weather -City Prague |
          Should -Be 'Sunny'
    } catch { }
  }
}
```

Problems

- Ultimate tool to keep management happy. High code coverage and no failed builds! :D

- Tests never fail
- You have false sense of security
- You have no idea if your module / script works

```
# Model of an Assertion
```

```
function Should-BeEqual ($Expected, $Actual) {
  if ($Expected -ne $Actual)
  {
    throw [Exception]`
    "Expected the actual value to be '$Expected'" +
    "but it was '$Actual'."
  }
}
```

Solutions

- Don‘t catch all exeptions in It

- Assertions are just ifs + exceptions
- You can do assertions without Should or register your own in Should

- See Custom Assertions for Pester Tests - https://mathieubuisson.github.io/pester-custom-assertions/
- See Assert - https://github.com/nohwnd/Assert
- See Axiomatic assertions - http://jakubjares.com/2017/12/20/axiomatic-assertions/

Non-descriptive test names

What's happening?

- Test1
- Do something
- Fails
- Runs correctly
- Negative values
- Positive values
- Empty file
- Wrong file

- Load file
- Returns result
- Add operation
- Does not fail
- Has error
- Returns message
- Add 1 and 2
- Delete locked file

Wut?

PowerShell Conference EU 2018

Solutions

- Your test names can include spaces and can be arbitrarily long, use that power
- Include both what you are testing and the expected outcome
- Have a project / team wide test naming convention, and use it

Now I know!

Given 1 and 5 it returns 6

Given a valid username and password it logs the user in

Given invalid password it throws an unauthorized exception

Given values that are out of allowed range it throws argument exception

Given two collections '<expected>' '<actual>' of the same size it returns `$true

Unreadable tests

```powershell
# ugh? what?

It "Returns the correct value" {
  Setup-TestCase
  Assert-Test1Passed
}

It "Returns the correct value" {
  Setup-TestCase
  Assert-Test2Passed
}
```

Problems

- No intent in code
- Code is too DRY
- To understand anything you have to dig deep
- Shared data tend to grow
- Every test depends on multiple other tests

- Total opposite of what we want in tests.

Solutions

- Strive for readability not DRYness (aka DAMP)
- Hide generic setup in helper functions, but not the test data
- Avoid parameters in helper functions

```
# too damp

It "Give numbers from 0 to 100 it gives the average 50" {
  # setup
  $password = ConvertTo-SecureString “Password100!”
                -AsPlainText -Force
  $credentials = New-Object PSCredential "admin", $password
  $session = Get-DeepMindSession -Credential $credentials
                -Url http://deepmind1.test.com
  # data
  $values = 0..100
  $expected = 50

  $actual = Get-DeepMindAverage
                -Session $session -Values $values

  $actual | Should -Be $expected
}
```

```powershell
# damp just right

It "Give numbers from 0 to 100 it calculates the average 50" {
  $session = Get-TestDeepMindAdminSession

  $values = 0..100
  $expected = 50

  $actual = Get-DeepMindAverage -Session $session `
     -Values $values

  $actual | Should -Be $expected
}
```

```
# make the AAA sections explicit

It "Give numbers from 0 to 100 it calculates the
average 50" {
  # -- Arrange
  $session = Get-TestDeepMindAdminSession

  $values = 0..100
  $expected = 50

  # -- Act
  $actual = Get-DeepMindAverage -Session $session `
     -Values $values

  # -- Assert
  $actual | Should Be $expected
}
```

```powershell
# use -Because where appropriate

It "Deletes a user from database" {
  # -- Arrange
  $dbConnection = Get-AdminSqlConnection

  $id = Create-User $dbConnection testUser1
  $createdUser = Get-User $dbConnection $id

  $createdUser | Should -Not -Be $null -Because `
     "we just created the test user, " +
     "and without it this test is invalid"

  # -- Act
  Delete-User $dbConnection $id

  # -- Assert
  $actual = Get-User $dbConnection $id
  $actual | Should -Be $null
}
```

Unreadable test cases

```
#  What are test cases?

It "Given valid -Name '<Filter>', it returns
'<Expected>'" -TestCases @(
    @{ Filter = 'Earth'; Expected = 'Earth' }
    @{ Filter = 'ne*'  ; Expected = 'Neptune' }
    @{ Filter = 'ur*'  ; Expected = 'Uranus' }
    @{ Filter = 'm*'   ; Expected = 'Mercury', 'Mars' }
) {
    param ($Filter, $Expected)

    $planets = Get-Planet -Name $Filter
    $planets.Name | Should -Be $Expected
}
```

```
It "Compares two values" -TestCases @(
  @{ Left = $null; Right = $null; Result = $true },
  @{ Left = ""; Right = ""; Result = $true },
  @{ Left = $true; Right = 'True' ; Result = $true },
  @{ Left = $false; Right = 'False' ; Result = $true },
  @{ Left = "abc"; Right = @("abc") ; Result = $true },
  @{ Left = $null; Right = ""; Result = $false },
  @{ Left = $true; Right = 'False'; Result = $false },
  @{ Left = 1; Right = -1; Result = $false },
  @{ Left = {abc}; Right = {def}; Result = $false },
  @{ Left = (1,2,3); Right = (1,2,3,4); Result =$false },
  @{ Left = ([pscustomobject] @{ Name = 'Jakub' }); Right
= "a"; Result = $false},
  @{ Left = 'a'; Right = ([pscustomobject] @{ Name =
'Jakub' }); Result = $false}
) {
  param($Left, $Right, $Result)
  Compare-Value -Left $Left -Right $Right
      | Should -Be $Result
}
```

Problems

- Your test cases grow and grow until no-one knows what is going on

Solutions

- Split to more tests with the same body and add intent to their name
- Add Because string to the test case and pass it to assertion / test name
- Or at least split the test cases in groups and add comments

```
It "Given two values '<left>' and '<right>' that are the same it
returns `$true" -TestCases @(
  @{ Left = $null; Right = $null; Result = $true },
  @{ Left = ""; Right = ""; Result = $true }
) {}

It "Given two values '<left>' and '<right>' that both implicitly
cast to the same value it returns `$true" -TestCases @(
    @{ Left = $true; Right = 'True' ; Result = $true }
) {}

It "Given two values '<left>' and '<right>' that are different
but are special case that is fixed it returns `$true" -TestCases
@(
  @{ Left = $false; Right = 'False'; Result = $true },
) {}

It "Given two values '<left>' and '<right>' that are different
it returns `$false" -TestCases @(
  @{ Left = 1; Right = 8; Result = $false }
) {}
```

Too many assertions

```
# Assertion roulette

It "Given a name it returns appropriate user"{
  $actual = Get-User -Name "Jakub"

  $actual.Name | Should -Be "Jakub"
  $actual.Age | Should -Be 29
  $actual.Location | Should -Be "Prague"
  $actual.DrinksTooMuchCoffee | Should -Be $true
}
```

Problems

- Partial information on failure
- Extremely difficult to understand a test failure on a build server

- Seemingly random results as we progress to fix the bug

PowerShell Conference EU 2018

Solutions

- Write more tests (more work, not maintainable)

- Use Assert
- and use object equivalence - https://github.com/nohwnd/Assert#comparing-whole-objects
- or compound assertions (concept) - https://github.com/nohwnd/Assert/issues/19

```
# Use more tests

Context “Given a name it returns appropriate user” {
  $actual = Get-User -Name "Jakub"

  It “has the correct name” {
    $actual.Name | Should -Be "Jakub“
  }

  It “has the correct age” {
    $actual.Age | Should -Be 29
  }

  # etc. a lot of work...
}
```

```powershell
 # or use object equivalence

It "Returns correct user" {
  $expected = [PsCustomObject]@{
    Name = "Jakub"
    Age = 29
    Location = "Prague"
    DrinksTooMuchCoffee = $true
  }

  $actual = Get-User -Name "Jakub"

  $actual | Assert-Equivalent $expected
}

# awesome to test complement operations
# A compound assertion concept

It “Given a name it returns appropriate user"{
  $actual = Get-User -Name "Jakub"

  Combine-Assertion {
      $actual.Name | Should -Be "Jakub"
      $actual.Age | Should -Be 29
      $actual.Location | Should -Be "Prague"
      $actual.DrinksTooMuchCoffee | Should -Be $true
  }
}
```

Using Should -Not -Throw

```
 It "Adds two numbers" {
     # no assertion? is this test complete?
     Add-Number 10 10
}

It "Adds two numbers" {
     # there is an assertion
     # but does the same thing
     Add-Number 10 10 | Should -Not -Throw
}
```

Problems

- Every line is an implicit Should –Not –Throw
- Makes incomplete test look complete

Solutions

- Don’t use it
- Think harder about what your test proves, and add assertion for it
- Write tests from the Assert part
- Every Set-* should have a Get-* function

```
It "Adds two numbers" {
    # a proper test
    Add-Number 10 10 | Should -Be 20
}
```

Not failing in assertion

PowerShell Conference EU 2018

Problems

- You write test that fails in some place
- Then you fix it to pass

- You write test that immediately passes

PSCONF.EU

Solutions

- Make sure your test fails in the assertion, otherwise you don’t know that the test works

- When test passes immediately, think about the smallest change to make it fail in assertion, and do it, then revert it

Logic in tests

```
# Untested condition
if ($PSVersionTable.PsVersion.Major -ge 5){
  It “A test running only on powershell 5”{
    # code
  }
}

# Conditional test setup
It "Updates user in db"{
  $dbConnection = Get-AdminSqlConnection
  $name = "testUser1"
  if ($dbConnection) {
    Create-User $dbConnection $name
  }

  Update-User -Name $name -User $updatedData

  <# etc. #>
}
```

```
# Untested loop

$configuration:Servers | foreach {
  It “A test” {
    # code
  }
}
```

Problems

- Tests contain untested logic which makes them complex
- Tests do not fail-fast
- Loops don’t run when data is empty, lack of data makes your tests pass

Solutions

- Put the condition in a function and test it (or just set the tests to be skipped)
- Let the test fail when it cannot set-up itself
- Use test cases instead of a loop
- Throw when you get no data to loop on

```
# Tested function to conditionally run the test
Invoke-OnPowerShell5OrLaterOnly {
  It “A test running only on powershell 5=”{
    # code
  }
}

# Condition removed
It "Updates user in db"{
  $dbConnection = Get-AdminSqlConnection
  $name = "testUser1"

  Create-User $dbConnection $name # <- will fail here
  # (another reason to use the AAA comments)

  Update-User -Name $name -User $updatedData

  <# etc. #>
}
```

```
# Protected loop
function Assert-HasItems ($collection) {
    if (-not $collection) {
        throw "Collection is null"
    }

    if ($collection.Count -lt 1) {
        throw "Collection has no items"
    }

    $collection
}

Assert-HasItems ($configuration:Servers) |
foreach {
  It “A test” {
    # code
  }
}
```

Not testing your environment checks

```
 Describe 'Disk health checks' {
    It 'Has at least 10% of free space' {
        $disk = Get-WmiObject win32_logicaldisk |
            where DeviceId -eq 'C:'

        $diskSize = $disk.Size
        # 10% of the total size
        $expected = $diskSize * 0.1
        $expectedInGB = [Math]::Round(
                             $expected / 1GB, 2)

        $freeSpace = $di.FreeSpace
        $freeSpaceInGB = [Math]::Round(
                             $freeSpace / 1GB, 2)
        $freeSpaceInGB |
            Should -BeGreaterThan $expectedlnGB
    }
}
```

Problems

- Test is complex
- Can fail because test code is wrong, or because environment is wrong, and we cannot know which one it is

Solutions

- Extract test bodies to functions
- Use minimum amount of parameters, possible (while still keeping it practical)
- Test the functions

- See Environment testing with Pester: Testing your tests - http://jakubjares.com/2017/12/07/testing-your-environment-tests/

PSCONF.EU

DEMO

Quick run through environment tests

TDD

Problems

- You write tests that immediately pass.
- You edit test code and production code in the same step.
- You don’t fail in assertion.
- You write no acceptance tests.
- You try fixing your code without thinking about it.
- You try to do TDD all the time.

Solutions

- Don’t do the things on the problem list.
- Come talk to me later :)

PowerShell Conference EU 2018

The ideal case

Ideal test

Fast **I**solated **R**epeatable **S**elf-validating **T**imely

Small

Readable

Failing with a descriptive message

Failing in assertion

Summary

- Think about the next person working on your project (or your contributors)
- Write a lot of pure functions
- Review how much test failure tells you, before you start debugging
- Have your code and tests reviewed

- Keep it simple :)

PowerShell Conference EU 2018

Feedback please!

- Please post feedback on http://powershell.help and press the button if you liked.

PSCONF.EU

Questions?

PowerShell Conference EU 2018

Next Steps

- Now: 15 min break

- Grab a coffee
- Stay here to enjoy next presentation
- Change track and switch to another room

- Ask me questions or meet me in a breakout session room afterwards