



Pester internals and concepts



Pestering since 2013

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

@nohwnd
@pspester
me@jakubjares.com



2018

Pester internals and concepts



Jakub Jareš

Agenda

- Give overview of how Pester works internally
- Go from general concepts to details
- Hopefully get more contributors :)

Help us develop it

- Mocking classes?
 - Relaxing mock types?
 - Timeouts for Its?
 - Options?
 - Extensions?
-
- Or review docs, or fix a simple issue, everything helps!

Help us develop it!

Filters
Labels
Milestones

New issue

X Clear current search query, filters, and sorts

<input type="checkbox"/> 7 Open 3 Closed	Author	Labels	Projects	Milestones	Assignee	Sort
<input type="checkbox"/> <i>!</i> Should -Not -Throw doesn't print or display the actual exception if caught <div>good first issue</div> #1009 opened 22 days ago by ChrisLynchHPE						1
<input type="checkbox"/> <i>!</i> Mocking 'Get-PhysicalDisk' fails <div>Mocking</div> <div>good first issue</div> #980 opened on 31 Jan by peppekerstens						7
<input type="checkbox"/> <i>!</i> Documentation update required due to new assertion added by #959 <div>Documentation</div> <div>good first issue</div> #960 opened on 17 Dec 2017 by it-praktyk 4.4						5
<input type="checkbox"/> <i>!</i> How do I discover Should operators? <div>Documentation</div> <div>Question</div> <div>good first issue</div> #878 opened on 14 Sep 2017 by alx9r						3
<input type="checkbox"/> <i>!</i> Update documentation for Code coverage <div>Documentation</div> <div>good first issue</div> #872 opened on 10 Sep 2017 by nohwnd 4.3						
<input type="checkbox"/> <i>!</i> Move help to PlatyPS and generate pages of wiki based on it too <div>Documentation</div> <div>Feature</div> <div>good first issue</div> #720 opened on 31 Jan 2017 by nohwnd 4.3						7
<input type="checkbox"/> <i>!</i> Pester project needs an error message file <div>Feature</div> <div>good first issue</div> #524 opened on 24 Apr 2016 by juneb 4.3						4

Anyone not familiar with Pester?

Get-Planet.ps1

```
function Get-Planet {  
    @()  
}
```

Get-Planet.Tests.ps1

· \$PSScriptRoot\Get-Planet.ps1

```
Describe 'Get-Planet' {  
    It "Given no parameters, it lists all 8 planets" {  
        # -- Arrange  
  
        # -- Act  
        $allPlanets = Get-Planet  
  
        # -- Assert  
        $allPlanets | Should -HaveCount 8  
    }  
}
```



Code might not work

- Simplified code to show only relevant pieces
- Condensed PowerShell
- Shortened namespaces
- Questionable line-breaks

A test framework

A test framework

- Test discoverer
- Test runner
- Assertion library
- Mocking library
- Output to screen
- Result as nUnit XML

Test discovery & run

Test discovery & run

- Test discovery and run are done in the same step
- Pros:
 - Pester tests are just scripts
 - The structure is very loose
- Cons:
 - Pester cannot list tests without executing them
 - Pester cannot run just a single test

```
$testScripts = Get-ChildItem -Include *.Tests.ps1 -Recurse |  
    where-Object { -not $_.PSIsContainer } |  
    select-Object -ExpandProperty FullName -Unique  
  
foreach ($testScript in $testScripts)  
{  
    try  
    {  
        do  
        {  
            & $testScript # pass parameters  
        } until ($true) # <- single run loop, huh?  
    }  
    catch  
    {  
        # report errors  
    }  
    finally  
    {  
        # clean up mocks and internal state  
    }  
}
```



Describe (& Context)

```

function Describe ($Name, $ScriptBlock) {
    if ($pester -eq $null) {
        $pester = New-PesterState $Path $PSCmdlet.SessionState
    }

    Write-Host "Describe $Name"

    try
    {
        try {
            # setup framework and test
            do
            {
                $null = & $ScriptBlock
            } until ($true)
        }
        finally {
            # teardown test and framework
        }
    }
    catch {
        # report framework and code failures
    }
}

```



It

```

function It ($Name, $ScriptBlock, $TestCases) {
    foreach ($testCase in $TestCases) {
        Invoke-Test $Name $ScriptBlock $params
    }
}

function Invoke-Test ($Name, $ScriptBlock, $Params) {
    Write-Host "It $Name"

    try {
        # setup test
        do {
            $null = & $ScriptBlock @Params
        } until ($true)
    }
    catch {
        $errorRecord = $_
    }
    finally {
        # teardown test
    }

    Write-TestResult $Name $errorRecord
    $pester.AddTestResult( <#...#> )

    # progress report used to be here, but it was tooo slow :)
    # (we are talking 30 minutes vs. 2 minutes slow on macOS)
}

```



Assertions

Assertions

- How they work in Pester?
- How the syntax works?

Asssertions

How they work

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'."  
    }  
}
```

Model of a Throw Assertion

```
function Should-Throw ([ScriptBlock]$ScriptBlock) {  
    $exceptionWasThrown = $False  
    try  
    {  
        $null = & $ScriptBlock  
    }  
    catch  
    {  
        $exceptionWasThrown = $True  
        $_  
    }  
  
    if (-not $exceptionWasThrown)  
    {  
        throw 'Expected an exception to be thrown but no  
              exception was thrown.'  
    }  
}
```

```
# Pester's Contain assertion
# (because PesterBe is a bad place to start)
```

```
function PesterContain(
    $ActualValue, $ExpectedValue, [switch] $Negate)
{
    [bool] $succeeded = $ActualValue -contains $ExpectedValue
    if ($Negate) { $succeeded = -not $succeeded }

    if (-not $succeeded) {
        return New-Object psobject -Property @{
            Succeeded = $false
            FailureMessage = "... "
        }
    }

    return New-Object psobject -Property @{
        Succeeded = $true
    }
}
```



```

# Importing an assertion

# on Import-Module all *.ps1 in Assertions are dot-sourced
# including Contain.ps1
function PesterContain {
    # ...
}

Add-AssertionOperator -Name Contain `
    -Test $function:PesterContain `
    -SupportsArrayInput

# ---- end of Contain.ps1

function Add-AssertionOperator($Name, $ScriptBlock) {
    $script:AssertionOperators[$Name] = @{
        Name = $Name
        ScriptBlock = $ScriptBlock
    }
}

```



Executing an assertion

```
function Get-AssertionOperator($Name) {  
    $script:AssertionOperators[$Name]  
}  
  
function should {  
    $entry = Get-AssertionOperator `br/>        -Name ??? # <- we get the name somehow  
  
    $testResult = & $entry.Test @BoundParameters  
    if (-not $testResult.Succeeded) {  
        $errorRecord = New-ShouldErrorRecord `br/>            -Message $testResult.FailureMessage `br/>            -File $file `br/>            -Line $lineNumber `br/>            -LineText $lineText  
  
        throw $errorRecord  
    }  
}
```



Assertions

Assertion syntaxes

Should & Be syntaxes

Legacy syntax

1 | Should Be 9

```
function Should {  
    # [Parameter(ParameterSetName = 'Legacy')]  
    param ($__LegacyParam1, $__LegacyParam2,  
           $__LegacyParam3)  
}
```

Modern syntax

1 | Should -Be 9

```
function Should {  
    param ()  
}
```

where does the -Be come from?



```
# Add -Be assertion operator
```

```
Add-AssertionOperator -Name Be -Test $function:PesterBe  
                        -Alias 'EQ' -SupportsArrayInput
```

```
function PesterBe($ActualValue, $ExpectedValue,  
                  [switch] $Negate, [string] $Because) { }
```

```
# make sure assertion is unique, and then call
```

```
function Add-AssertionDynamicParameterSet ($AssertionEntry) {
```

```
    # generate a parameter set like this
```

```
    $params = param (  
        [Parameter(ParameterSetName = 'Be', Mandatory)]  
        [switch] $Be,  
        [Parameter(ParameterSetName = 'Be')]  
        $ActualValue, $ExpectedValue,  
        [switch] $Not, [string] $Because  
    )
```

```
$script:AssertionDynamicParams.Add(  
    $AssertionEntry.Name, $params)
```

```
}
```



```
# Use assertion operators on should
```

```
1 | Should -Be 9
```

```
function Should {  
    dynamicparam {  
        Get-AssertionDynamicParams  
    }  
  
    $entry = Get-AssertionOperator -Name  
            $PSCmdlet.ParameterSetName  
    # ... execution of the assertion  
}
```

```
function Get-AssertionDynamicParams  
{  
    return $script:AssertionDynamicParams  
}
```



Mocking

```
# A typical test
```

```
function Get-Planet {  
    # ...  
}
```

```
Describe 'Get-Planet' {  
    It "It loads planets from json" {  
        # -- Arrange  
        Mock Get-Content { <#...#> }  
  
        # -- Act  
        Get-Planet  
  
        # -- Assert  
        Assert-MockCalled Get-Content `  
            -ParameterFilter {$Path -like '*planets.json'}  
    }  
}
```



```
# A poor man's mock
```

```
function Get-Planet {  
    # ...  
}
```

```
Describe 'Get-Planet' {  
    It "It loads planets from json" {  
        # -- Arrange  
        function Get-Content ($Path) {  
            if ($Path -like '*planets.json') {  
                $script:gc = 1  
            }  
        }  
  
        # -- Act  
        Get-Planet  
  
        # -- Assert  
        $script:gc | Should -Be 1  
    }  
}
```

Function shadowing

- We can shadow functions
- But there is no Set-Function cmdlet with –Scope parameter :(
- And no remove function that removes just the function in the current scope and not the global function.
- So we define globally and have to clean up
- Luckily aliases are the first to be picked up, so we use that instead.

A model of Mock

```
$script:MockHistory = @{}
```

```
function 0981 () {  
    $script:MockHistory["Get-Content"] += 1;  
}
```

```
Set-Alias -Name 'Get-Content' -Value '0981 ' -Scope Script
```

```
function Assert-MockCalled ($Name) {  
    if ($script:MockHistory[$Name] -lt 1) {  
        throw "Expected $Name to be called at least once."  
    }  
}
```

```
Assert-MockCalled -Name 'Get-Content'
```



The real deal

- Find function
 - Generate the "random" function
 - Add aliases
 - Filter parameters
 - Count calls
-
- Call thru
 - Clean up after run

```

function Mock ($CommandName, $ScriptBlock, $Filter = {$true}) {
    $command = Resolve-Command $CommandName

    $mock = $mockTable[$command.Name]
    if (-not $mock) {
        $metadata = [CommandMetaData]$command
        $paramBlock = [ProxyCommand]::GetParamBlock($metadata)

        $guid = [Guid]::NewGuid()
        $body = Generate-Mock $command $paramBlock
        $mockScript = [ScriptBlock]::Create($body)
        $mockTable[$command.Name] = @{
            Command = $command
            CallHistory = @()
            BootstrapFunction = $guid
            FilterAndBody = @()
        }

        New-Function -Name $guidName -Definition $mockScript
        Set-Alias -Name $command.Name -Value $guid -Scope Script
    }
    $mock.FilterAndBody += @{ f = $Filter; b = $ScriptBlock }
}

```



not very interesting, but notice that usage
of cmdlets is avoided

```
function Resolve-Command ($Name) {  
    $command = $ExecutionContext  
                .InvokeCommand.GetCommand($Name, 'All')  
  
    if ($command.CommandType -eq 'Alias') {  
        $command = $command.ResolvedCommand  
    }  
    $command  
}
```

```
function New-Function ($Name, $Definition) {  
    $ExecutionContext.InvokeProvider.Item.Set(  
        "Function:\script:$Name", $Definition, $true, $true)  
}
```



```

function Generate-Mock ($Command, $ParamBlock) {
    # from MockPrototype function
    $prototype = @'
    $_arguments = $null
    if (#CANCAPTUREARGS#) {
        $_arguments = Get-Variable args -ValueOnly -Scope Local
    }
    $_psCmdlet = Get-Variable PSCmdlet -ValueOnly -Scope Local
    $_sessionState = if ($_psCmdlet) { $_psCmdlet.SessionState }

    Invoke-Mock `
        -CommandName '#NAME#' `
        -BoundParameters $PSBoundParameters `
        -ArgumentList $_arguments `
        -CallerSessionState $_sessionState `
        -FromBlock '#BLOCK#' `
        -MockCallState $_mockCallState
    #INPUT#
'@
    <# ... next slide #>
}

```



```

function Generate-Mock ($Command, $ParamBlock) {
    $prototype = "Invoke-Mock -CommandName '#NAME#' `
                  -FromBlock '#BLOCK#' #INPUT#"

    $p = $prototype -replace '#NAME#', $Command.Name
    return @"
param ($ParamBlock)
begin { #
    `$_mockCallState = @{}
    $($p -replace '#BLOCK#', 'Begin' -replace '#INPUT#')
}
process {
    $($p -replace '#BLOCK#', 'Process' `
                  -replace '#INPUT#', '-InputObject @($input)')
}
end {
    $($p -replace '#BLOCK#', 'End' -replace '#INPUT#')
}
"@
}

```



The real code :)

underscores to prevent naming conflicts?

we use \${} and spaces instead

user could mock Get-Variable?

keep Get-Command safe and use indirect invocation

```
${get Variable Command} = & (Pester\SafeGetCommand) `
    -Name Get-Variable `
    -Module Microsoft.PowerShell.Utility `
    -CommandType Cmdlet
```

```
[object] ${a r g s} = $null
if (${#CANCAPTUREARGS#}) {
    ${a r g s} = & ${get Variable Command} `
        -Name args `
        -ValueOnly `
        -Scope Local `
        -ErrorAction ${ignore preference}
}
if ($null -eq ${a r g s}) { ${a r g s} = @() }
```



```

function Invoke-Mock ( $CommandName,
                      $BoundParameters,
                      $FromBlock)
{
    $mock = $mockTable[$CommandName]
    $body = Find-Body $mock.FilterAndBody $BoundParameters
    if ($body) {
        Execute-Body $mock $body $BoundParameters
    }
    # else call the real command
}

function Execute-Body ($Mock, $Body, $BoundParameters) {
    $Mock.CallHistory += @{
        CommandName = $Mock.Command.Name
        BoundParameters = $boundParameters
    }

    & $Body @BoundParameters
}

```




```
function ToParamBlock ($BoundParameters) {
    $params =
        ($BoundParameters.Keys | % { "`${$_}" }) -join ', '
    "param ($params)"
}
```

```
function Find-Body ($FilterAndBody, $BoundParameters) {
    # in reality we iterate from bottom
    # (from latest specific mock to oldest general mock)
    $FilterAndBody | foreach { # <- array of @{b =; f =; }
        $paramBlock = ToParamBlock $BoundParameters
        $filter = [ScriptBlock]::Create("
            $paramBlock
            Set-StrictMode -Off
            $($_.f)" # <- mock filter
        )

        if (& $filter @BoundParameters) {
            return $_.b # <- mock body
        }
    }
}
```



```

function Assert-MockCalled
(
    $CommandName, $Times = 1, $Filter = {$True}) {
    $command = Resolve-Command $CommandName
    $mock = $mockTable[$command.Name]

    $matchingCalls = @()
    $mock.CallHistory | foreach {
        # like when we looked up the correct body for a mock
        if (Test-ParameterFilter $Filter $BoundParameters) {
            $matchingCalls.Add($_)
        }
    }

    if ($matchingCalls.Count -lt $times) {
        throw "Expected $CommandName to be called " +
            "at least $times times, but was called " +
            "$($matchingCalls.Count) times."
    }
}

```



Scoping

```
# A typical test
# Planets.psm1
function Get-Planet ($Name) {
    $planets | Filter-Planet $Name
}
function Filter-Planet {} # <- internal
Export-ModuleMember -Function Get-Planet
```

```
# Planets.Tests.ps1
Import-Module Planets.psm1
InModuleScope -ModuleName Planets {
    Describe "Filter-Planet" {
        It "Finds no Alpha Centauri" {
            Filter-Planet "Alpha Centauri" |
                Should -BeNullOrEmpty
        }
        It "Finds Earth" {
            Filter-Planet "Earth" |
                Should -Be "Earth"
        }
    }
}
```

The problem

- Run test code inside of a module
- Generate scriptblock inside of Pester and run it in user context

Session states

- Silos that hold variables, functions, and scopes
- One silo per module + one for user code
- Everything is a script block
- Script blocks attach to their creators session state
- See Bruce Payette – Scoping in depth

<https://github.com/psconfEU/2017>

<https://www.youtube.com/watch?v=er9Juk51hgw>



DEMO

Quick overview of scoping in Pester

Summary

- Assertions operators and exceptions
- Mocking is clever function shadowing
- Describes and Its are try catch blocks around user provided scriptblock
- Scoping is a bit hacky :D

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

Questions?