PowerShell Conference Europe 2019

Hannover, Germany

June 4-7, 2019

Pester internals and concepts

Jakub Jares

This Session

Understand how Pester works internally (a bit more).

Agenda

- Scoping, Scoping, Scoping
  - how does it work and what hoops Pester needs to jump through to make testing work.

Looking forward to Pester v5:

- Test Discovery
- Test execution
- Result object
- Plugins

This is v2 of the talk

v2: focused on scoping & various bits that are new in v5

v1: from PSConfEU 2018, focuses on Pester v4, gives comprehensive overview of the whole framework

https://www.youtube.com/watch?v=Wc-9B_MqxYs

More talks

- Tomorrow, 11:00 Red Room, Custom Pester assertions are the vocabulary of your tests

- Tomorrow 12:25 Red Room, Community discussion about Pester v5 - https://github.com/pester/Pester/issues/1319

Slide code might not work

- Simplified code to show only relevant pieces

- Condensed PowerShell

- Shortened namespaces
- Questionable line-breaks

nohwnd

Scoping

nohwnd

Scoping

- part of the Pester that is hardest part to understand, because it is the hardest part to **see**
- unless you develop Pester, or your own DSLs you probably never had to deal with it too much (lucky you 😁)

Overview

- establish baseline by repeating basics
- progress to more advanced topics
- show what they are used for in Pester

nohwnd

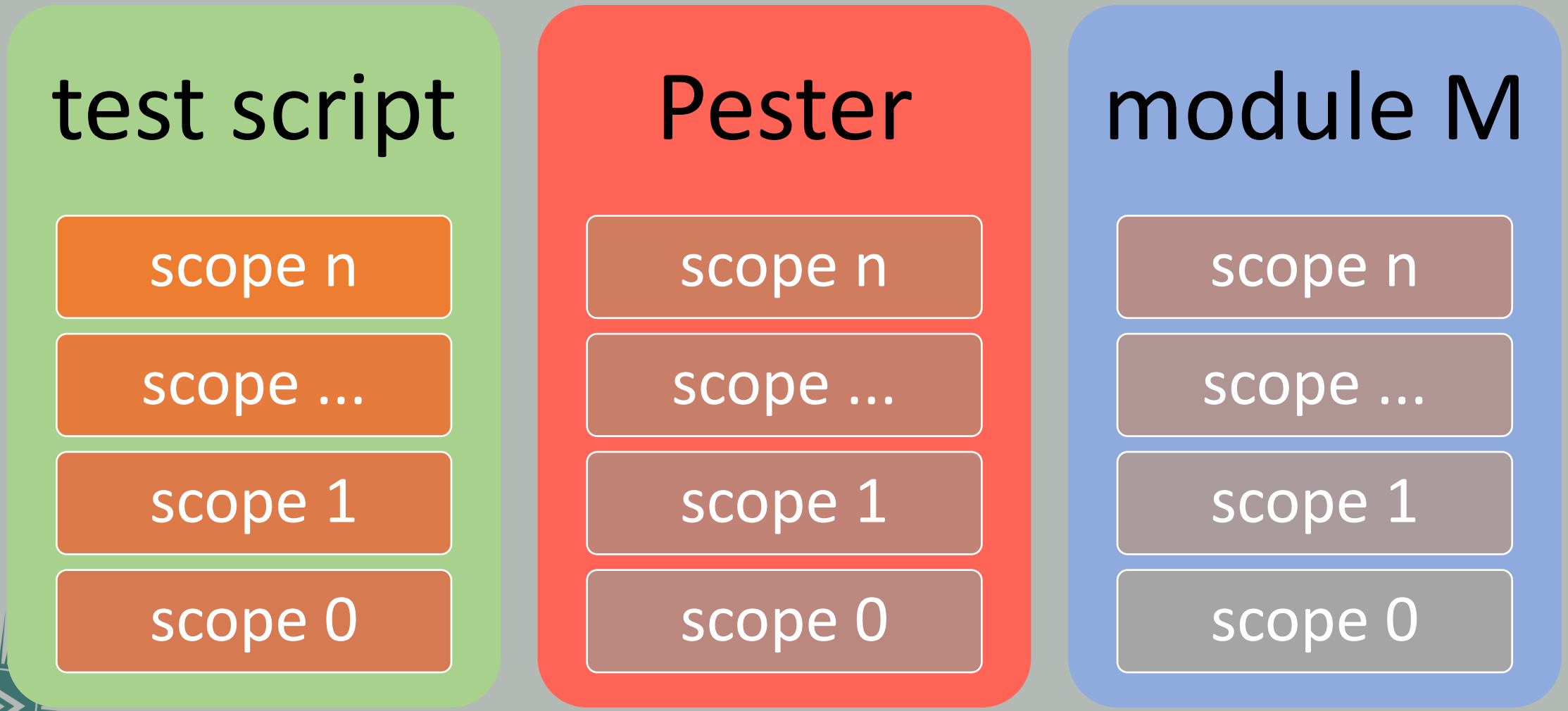
Scope & SessionState

- Scope
  - maintains lifetime of variables, functions (and etc.)
  - some constructs define new scope, some don't
  - when scope ends variables and functions defined in it are removed

  - dynamic scoping - values fall through from upper scopes to lower scopes

- Session state
  - holds all variables and functions (and etc.) of a module
  - there is script session state that is bound to "anonymous module"

Scope & SessionState



nohwnd

```powershell
New-Module M {
  $script:a = "module M"
  function Get-Internal () { $script:a }
  function Get-Public () { Get-Internal }
  Export-ModuleMember Get-Public
} | Import-Module



Describe "d" {
  It "i" {
    $expected = "Hello"
    # 1 & 2
    Mock Get-Internal { #4 $expected } -ModuleName M
 #3 Get-Public | Should -Be $expected
  }
}
```

1. from Pester module lookup internal function in module M

2. define mock in module M from inside of Pester module

3. invoke mock from module M, lookup mock behavior inside of Pester

4. invoke the behavior in the test script so we can resolve variables

Warning

⚠ I will most likely say Scope instead of SessionState few times, because I am used to talk about both as scope.

When I say "one scope deeper", "in the parent scope", or anything else vertical I mean Scope.

When I say "module scope", "caller scope", or anything else horizontal" I mean SessionState.

See Bruce Payette – Scoping in depth (psconfeu2017)

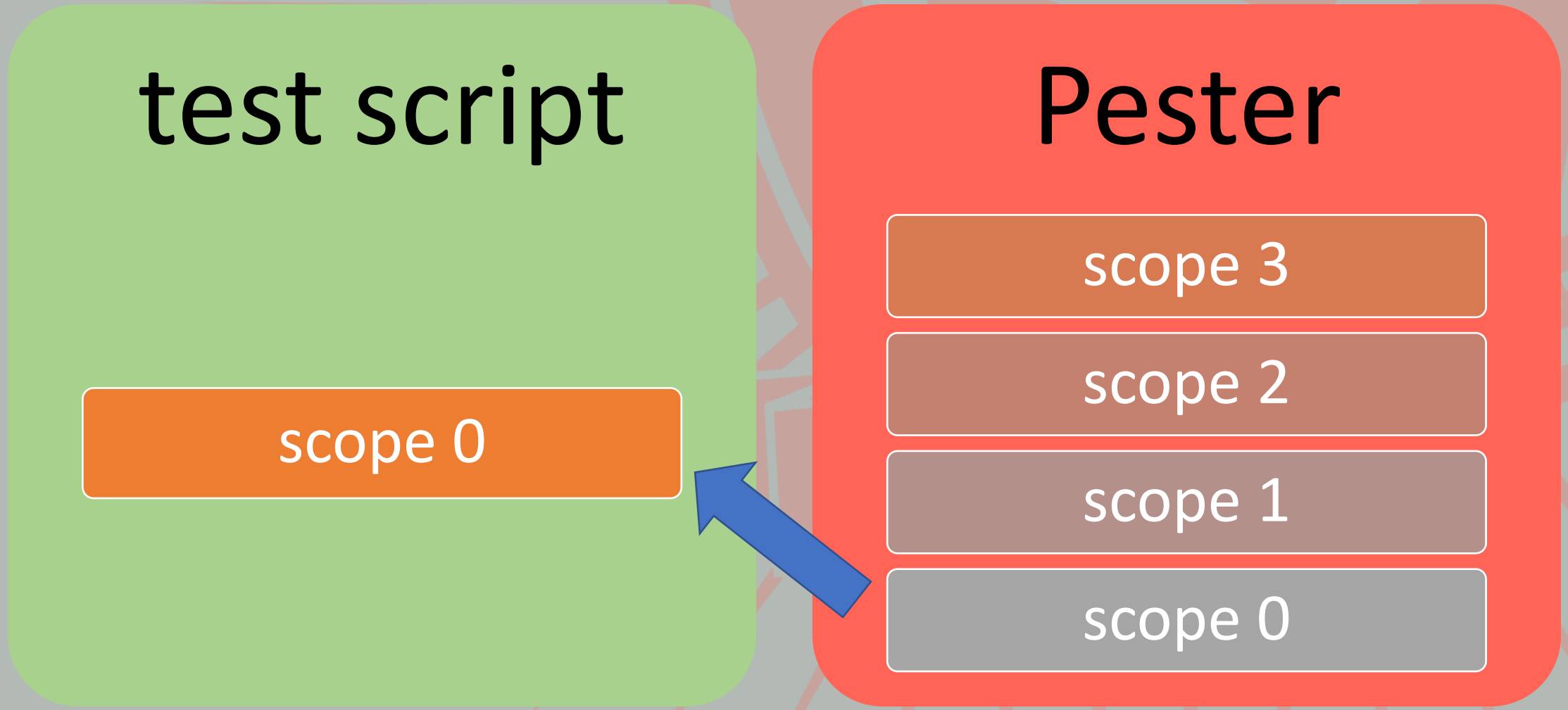
https://www.youtube.com/watch?v=er9Juk51hgw

DEMO

many scoping demos

nohwnd

Scope & SessionState



Summary

- SessionState is like a silo holding separate state per module
- Scopes hold variables, functions and other stuff
- Scriptblocks are bound to session state that created them and stay bound even if you invoke them from a different session state
- Scopes are tracked independently in each session state
- Scriptblock parameters allow you to pass scriptblocks and command references around

Test discovery

nohwnd

Test discovery

- Find structure of tests **without** executing them

This allows us to:

- overview the test structure without running the tests

- filter tests based on **any criteria** not just name

- skip whole files when no tests in them will run
- skip setups when no test will run afterwards

But how?

- Find structure of tests **without** executing them?

- AST or execution?
  - Learn to love the AST – Anthony Allen, Friday, Blue Room
  - Like a script monkey in the Syntax Tree – Walter Legowski, Thursday, Liebniz room

- Must be able to handle test and blocks:
  - in separate files
  - generated by functions and loops
  - with the same name
  - with expanded variables in names

2-pass execution – Discovery + Run

- Discovery
  - Execute & save Describe/Context blocks
  - Save It blocks

- Run
  - Execute Describe/Context blocks
  - Execute It blocks

All code **must be** in Pester controlled blocks

```powershell
# v4
. "$PSScriptRoot/Get-Greeting.ps1"

Describe "Get-Greeting" {
    # ...
}

# v5
BeforeAll {
    . "$PSScriptRoot/Get-Greeting.ps1"
}

Describe "Get-Greeting" {
    # ...
}
```

```powershell
$result = @() ; $currentBlock = $null ; $isDiscovery = $true

function Describe ($Name, $ScriptBlock) {
    if ($isDiscovery) {
        $currentBlock = @{ Name = $Name; ScriptBlock = $ScriptBlock; Tests = @() }
        $result += $currentBlock
    }
    & $ScriptBlock
}

function It ($Name, $ScriptBlock) {
    if ($isDiscovery) {
        # just save the info
        $currentBlock.Tests += @{ Name = $Name; ScriptBlock = $ScriptBlock }
    } else { # find test and execute it }
}
```

```
$isDiscovery = $true
Describe "d1" {
    It "t1" {
        # this will not execute
        throw "Fail!"
    }
}

@{
    Name = "d1"
    ScriptBlock = ...
    Tests = @(
        @{
            Name = "t1"
            ScriptBlock = { throw “Fail!...
        }
    )
}
```

Upsides

- Run only what is needed
- Filtering on any Describe, Context and It
  - tag filter
  - name filter
  - position filter
  - predicate filter
- Focusing tests
- Better UI integration

Downsides

- Longer time before first test run (execution seems slower)
- All code needs to be in Pester controlled blocks

- No-interactive mode (yet)

DEMO

Discovery, filtering, and focusing

Test execution

Test execution

- invoke 'Describe / Context / It' function
- lookup metadata
- invoke setups (and teardowns)

- Main challenge is keeping scoping correct

```powershell
# look into current block for metadata
$test = Find-CurrentTest -Name $Name -ScriptBlock $ScriptBlock -Id $Id
Set-CurrentTest -Test $test

if (-not $test.ShouldRun) {
    return
}

$test.ExecutedAt = [DateTime]::Now
$test.Executed = $true

Invoke-ScriptBlock `
    -Setup $beforeEach
    -ScriptBlock $ScriptBlock
    -Teardown $afterEach
```

```
# Pester runtime model
& {
    try {
        . $beforeAll
        & {
            try {
                . $beforeEach
                . $test
            } catch { <#...#> }
            finally {  . $afterEach  }
        }
    } catch { <#...#> }
    finally { . $afterAll }
}
```

Pester runtime reality

https://github.com/pester/Pester/blob/016c80bfe54a87e4decf9c2bc645911f0dcc7df8/new-runtimepoc/Pester.Runtime.psm1#L1255

```
if ($null -ne $_____parameters.Setup -and $_____parameters.Setup.Length -gt 0) {
    if ($_____parameters.EnableWriteDebug) { &$_____parameters.WriteDebug "Running inner setups" }
    foreach ($_____current in $_____parameters.Setup) {
        if ($_____parameters.EnableWriteDebug) { &$_____parameters.WriteDebug "Running inner setup
        $_____parameters.CurrentlyExecutingScriptBlock = $_____current
        . $_____current @_____innerSplat
    }
    $_____current = $null
    $_____parameters.Setup = $null
    if ($_____parameters.EnableWriteDebug) { &$_____parameters.WriteDebug "Done running inner setup
}
else {
    if ($_____parameters.EnableWriteDebug) { &$_____parameters.WriteDebug "There are no inner setup
}

if ($_____parameters.EnableWriteDebug) { &$_____parameters.WriteDebug "Running scriptblock { $($___
$_____parameters.CurrentlyExecutingScriptBlock = $_____parameters.ScriptBlock
. $_____parameters.ScriptBlock @_____innerSplat

if ($_____parameters.EnableWriteDebug) { &$_____parameters.WriteDebug "Done running scrtptblock" }
```

Result object

nohwnd

Result object

- Tree structure closely copying structure of test file

```
$runResult[0].Blocks[0].Blocks[0].Tests[3].Name

$runResult[0].Blocks[0].Blocks[0].Tests[3].ShouldRun
$runResult[0].Blocks[0].Blocks[0].Tests[3].Executed
$runResult[0].Blocks[0].Blocks[0].Tests[3].Passed
$runResult[0].Blocks[0].Blocks[0].Tests[3].ErrorRecord
$runResult[0].Blocks[0].Blocks[0].Tests[3].Duration
```

Result object - motivation

- Test files are hierarchy of Describe and It, so result object should be as well
- Easier to process results
- Easy to combine from multiple test suites / test runs
- Extensibility

- State separated per block / test instead of global mutable state
- better post-mortem -> less debugging
- Audit of what happened

DEMO

demo of result object

Plugins

nohwnd

Plugins

- e.g. Write Screen, Mock, TestDrive, TestRegistry

- de-couple the execution from additional behavior
- easier to re-organize what happens where

- implement one concern in one place
- enable / disable whole plugin if needed

- **Internal only right now**

Plugins

- a collection of scriptblocks that are called at appropriate times
- $Context is provided pointing to the current block, test and plugin configuration

- Multiple plugins can use the same step

- setups are called in the order that they are provided, teardowns are called in reverse order, e.g.

Write-Screen, Test-Drive, Mock ... Mock, Test-Drive, Write-Screen

```powershell
function New-PluginObject {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory = $true)]
        [String] $Name,
        [Hashtable] $Configuration,
        [ScriptBlock] $Start,
        [ScriptBlock] $DiscoveryStart,
        [ScriptBlock] $ContainerDiscoveryStart,
        [ScriptBlock] $BlockDiscoveryStart,
      ... etc
        [ScriptBlock] $EachTestTeardownEnd,
        [ScriptBlock] $OneTimeTestTeardownEnd,
      ... etc
    )

    $h = @{
        Name = $Name
        Configuration           = $Configuration
        Start                   = $Start
        DiscoveryStart          = $DiscoveryStart
        ContainerDiscoveryStart = $ContainerDiscoveryStart
      ... etc
```

```powershell
# mock plugin (reality)
New-PluginObject -Name "Mock" `
    -EachBlockSetupStart {
        param($Context)
        $Context.Block.PluginData.Mock = @{
            Hooks       = @()
            CallHistory = @{}
            Behaviors   = @{}
        }
    } -EachTestSetupStart {
        param($Context)
        $Context.Test.PluginData.Mock = @{ # same as setup start }
    } -EachTestTeardownEnd {
        param($Context)
        Remove-MockHook -Hooks $Context.Test.PluginData.Mock.Hooks
    } -EachBlockTeardownEnd {
        param($Context)
        Remove-MockHook -Hooks $Context.Block.PluginData.Mock.Hooks
    }
```

```powershell
# plugin data are attached on block / test
# so they can be resolved through current block if needed

function Get-MockDataForCurrentScope {
    [CmdletBinding()]
    param(
    )

    $location = $currentTest = Get-CurrentTest
    $inTest = any $currentTest

    if (-not $inTest) {
        $location = Get-CurrentBlock
    }

    $location.PluginData.Mock
}
```

DEMO

mock metadata and trail

Summary

- Knowing your scopes and sessionstates can help you make PowerShell behave, the way you want. It is especially useful when you are dealing with DSLs
- A lot of new stuff is coming in Pester v5, come discuss it tomorrow during lunch
- Try it for yourself:

```
Install-Module Pester –AllowPrerelease -Force
```

- Thanks to all contributors, and users of Pester!

nohwnd

Slides and demo code

```
Start-Process -FilePath https://github.com/psconfeu/2019
```

nohwnd

Questions?

Use the conference app to vote for this session:
https://my.eventraft.com/psconfeu

nohwnd

about_Speaker

Jakub Jareš

@nohwnd

@pspester

nohwnd