# Slide 3 – This bit is import

Click to display “time to fire PowerShell”

Start with Slide3\MessageBox1.ps1

Add-Type -AssemblyName PresentationCore,PresentationFramework

[System.Windows.MessageBox]::Show("Look a message box","Title")

Turn into a function MessageBox2.ps1

function MsgBox([string] $message,[string] $title)

{

[System.Windows.MessageBox]::Show($message,$title)

}

MsgBox("Look a message Box","Title")

Use this to show what’s happening (code in slide3\function1.ps1)  
function f($p1,$p2)

{

Write-Output "P1 is"

$p1.GetType()

if ($p2 -ne $null)

{

Write-Output "P2 is"

$p2.GetType()

}

}

f(1,2)

Show we don’t need parenthesis to create an array  
F 1,2

Then do – Point out because we’re a shell we use spaces  
F 1 2

Finally   
F @()

Fix by changing message box call to

MsgBox "Look a message Box" "Title"

Show names by changing message box call to

MsgBox "title" "Message"

then

MsgBox -title "title" -message "Message"

# Slide 4 – Returning stuff

Open up slide4\multiplier1.ps1

Run it..

Then add get type code to show what’s happening(code in slide4\multiplier2.ps1)

$d=ThreeThings 4 5 6

Write-Host $d.GetType()

Write-Host "The answer is $d"

Then add out-null (code in slide4\multiplier3.ps1)

MultiplyBy4 $a | Out-Null

MultiplyBy5 $a | Out-Null

# Slide5 – Write stuff to outputs

New demo code Slide5\WriteDemo1.ps1

function TestOutput

{

Write-Host "Hello"

Write-Output "There"

Write-Host "World"

}

TestOutput

Change to Slide5\WriteDemo2.ps1

$a=TestOutput

Write-Host $a

BACK TO POWERPOINT and click to display other write types

BACK TO Powershell ISE

Add these to the function Slide5\WriteDemo3.ps1

Write-Verbose "Verbose stuff"

Write-Debug "Debug Stuff"

Try to run with .\WriteDemo3.ps1 -Verbose

BACK TO POWERPOINT and click to display “Hey What”

BACK TO Powershell ISE

Put in a dummy parameter section with [CmdLetBinding()] Slide5\WriteDemo4.ps1

[CmdLetBinding()]

Param()

Run with .\WriteDemo4.ps1 -Verbose & -Debug

Remember to mention can be set via code using $VerbosePreference

Mention $ErrorActionPreference =”Stop” or ‘inquire’

# Slide 6 - Pester

IMPORTANT Start this by opening visual studio then open the pesterdemo.sln make sure you’re on branch slide6\_1

Start by creating a powershell script in VS2015 then add a matching test

Use branch Slide6\_1 add a powershell test to it, change the test to look like this

Describe "Rename" {

Context "Rename Item is called" {

$filename= 'c:\temp\test.txt'

Mock Rename-Item {}

It "Runs" {

.\Rename.ps1 -Path $filename

Assert-MockCalled Rename-Item -ParameterFilter {$path -eq $filename -and $newName -eq "c:\temp\test.txt.bak"}

}

}

Create our script called rename.ps1 (already done)

Param (

[Parameter(Mandatory=$true)]

[String] $Path

)

Write-Host $path

Add this to top of tests script and re-run

Write-Host "current directory = $(Get-Location)"

Then do ”New-Fixture deploy clean” in the interactive powershell window

notepad .\deploy\clean.Tests.ps1

show it and add the following to the top of the file

$here = Split-Path -Parent $MyInvocation.MyCommand.Path

Set-Location $here

**Don’t forget to close notepad down then do “rd .\deploy -recurse”**

Cheat and change the branch to Slide6\_4 as waiting to watch you type this in is boring

Now our test will fail actually running our script so lets add our rename function…

$failed= $false

function invoke-Rename ($path)

{

try

{

$newPath=$path +".bak"

Rename-Item -Path $Path -NewName $newPath

}

catch

{

$failed=$true

}

}

invoke-rename $Path

if ($failed)

{

Write-Output "Failed to rename $path"

}

Great so now lets use the test drive

Context "Rename Integration" {

$fileName='TestDrive:\test.txt'

New-Item -Path $fileName -Type File

It "Runs"{

.\Rename.ps1 -Path $fileName

"$filename.bak" | Should Exist

"$filename" | Should Not Exist

}

}

**Checked in as branch Slide6\_5**

Hmm fails… Lets add some verbose logging….

Write-Verbose "Renaming $path to $newPath"

Rename-Item -Path $Path -NewName $newPath

Write-Verbose "Rename done"

}

catch

{

Write-Verbose "Error was $\_"

And add a -Verbose switch to our test…..

Bad news the error was because I had a full path and Rename item doesn’t want a path, so add

$newName=Split-Path $newPath -Leaf

Yeah our test passes (need to change old mocking test)!

$newName -eq 'test.txt.bak'}

Lets do code coverage (need powershell interactive window for this)

Invoke-Pester -CodeCoverage .\Rename.ps1

**Checked in as branch Slide6\_8**

Okay lets add an error test

Context "Rename Item is called and errors happen" {

$filename= 'c:\temp\test.txt'

Mock Rename-Item { throw "bad thing"}

It "Runs" {

.\Rename.ps1 -Path $filename | Should belike "Failed \*"

Assert-MockCalled Rename-Item -ParameterFilter {$path -eq $filename -and $newName -eq "test.txt.bak"}

}

}

Hmm we went through… So lets see what the script analyser has to say..

Invoke-ScriptAnalyzer .\Rename.ps1

Ah time to talk about scope…. Open slide6\scope2.ps1 in **PowerShell ISE**

$magicNumber =10

function Test

{

$a=$magicNumber \*3

Write-Host "3 times $magicNumber is $a"

$magicNumber= 11

$a=$magicNumber \*3

Write-Host "3 times $magicNumber is $a"

}

Test

$a=$magicNumber \*3

Write-Host "3 times $magicNumber is $a"

Run it then add a script: to this script

$script:magicNumber= 11

Mention additional scopes

Global, Local, Script,private, numbered (0 is current local, useful when recursing) $using scope for transferring when running across machines.

Rerun the VS2015 stuff with the test.

Rerun coverage stuff

Back to PowerPoint

# Slide 6 -What did we learn

# Slide 7 – Other things

# Slide 8 – Important stuff

Pester doesn’t deal with collection types well

# Slide 9 – Bad year for celebrities

one at a time

# Slide 10 – Cocktails

Espresso martini – Tell Kate Moss (allegedly) story

Cocktail demo (I was on late Friday afternoon so was going to make one on stage)

Jon Skeet bits

e

# Slide 11 – Cocktails three ways - Webservice

Webservices…. Build up interactively to this (Slide11\Cocktail1.ps1)

$service=New-WebServiceProxy -uri http://localhost:30000/cocktail.svc -class cocktail

$results=$service.GetByAvailableIngredients(@("Gin","Vermouth - Sweet Red","Campari"))

Write-Output 'You can make'

$results

Change last line to (Slide11\Cocktail2.ps1)

Write-Output "You can make $results"

Talk about difference between “” and ‘’

Build up calling

$recipe=$service.GetRecipe($results[0])

$recipe

Add

$recipe.Ingredients | Format-Table

Finally

$recipe.Ingredients | Format-Table -Property Amount,Unit,Ingredient

By this point we’re at Slide11\cocktail3.ps1

Add a format string (Slide11\format1.ps1) run it takes too long to type in

$fmtIngredients=@{Expression={$\_.Amount};Label='Amount'},

@{Expression={$\_.Unit};Label='Unit'},

@{Expression={$\_.Ingredient.Name};Label='Name'}

$recipe.Ingredients | Format-Table $fmtIngredients

Done so far is in Slide11\Cocktail4.ps1

Open up Slide11\cocktail5.ps1

Talk through adding parameters Show where we’ve used them in the uri.

Run it

Point out the error mentions drives. Show   
$env:path  
Get-PSDrive  
Set-Location -Path “HKCU:”  
Set-Location -Path “D:”

Fix it by changing to

$service=New-WebServiceProxy -uri "http://${serverName}:${portNumber}/cocktail.svc" -class cocktail

Final bit saved as Slide11\Cocktail6.ps1

Open up Slide11\Cocktails7.ps1

Scroll down and show our function + pipeline

Show Get-Recipes

Modify and set GetRecipes to set the recipeName to be something from the pipeline

param(

[Parameter(ValueFromPipeline=$true)]

[string] $cocktailName

)

Next sort out an instance of the service steal code from Get-Ingredients

Then add to the process bit

begin

{

if ($script:service -eq $null)

{

$script:service=New-WebServiceProxy -uri "http://${serverName}:${portNumber}/cocktail.svc" -class cocktail

}

}

process

{

$script:service.GetRecipe($cocktailName)

}

Final position is Slide11\Cocktails8.ps1

Run Invoke-ScriptAnalyze

Back to powerpoint

# Slide 12 – Cocktails three ways – Convert to powershell module

Start by opening Slide12\CocktailPSM\Cocktail.sln in VS2015

**Make sure branch is Slide12\_1**

Show the CocktailPSM.psm1

Point out we’ve created two script level variables, and changed the scope of the service

Open the test and point out we’re explicitly importing the module.

Hmm we’ve had an error because it doesn’t know about the commands because we haven’t exported them. So open snippets and copy the export module stuff and paste into the bottom of the PSM1 file.

We also need to modify manifest change RootModule & FunctionsToExport to:

RootModule = 'CocktailPSM.psm1'  
FunctionsToExport = '\*'

Saved away as branch Slide12\_2

Run Tests

Back to the test

Something isn’t right I’m comparing against (‘Negroni’,Americano,hello) so test should fail. Remember when I said pipe line comparisons didn’t work right, well this is one of those case so to fix we need to do this.

,$(Get-ByIngredient $ingredients) | Should MatchArrayUnordered ('Negroni','Americano','Hello')

Saved away as branch Slide12\_3

Remove hello and run test, test passes

Open powershell ISE, open

Slide12\DemoModule.ps1

Run it.

Type help Get-ByIngredient

From snippets.txt grab help & add to Get-ByIngredient

import-module .\CocktailPSM\CocktailPSM -Force

redo help

Check in as branch Slide12\_4

# Slide 13 – Cocktails as a binary module

Change to branch Slide13\_1

Open Recipe class

Inherits from PSCmdLet

CmdLet attribute gives Verb + name to give us get-Recipe

Parameters are public properties on the class

We have two parameter sets Default & Ingredients both of which take either values from the pipeline. (You can do the same in a powershell script)

Futher down show begin, process and end.

Do a build!

Now lets test it by opening up PowerShell ISE Change directory to Slide13 and open up AlcoholicCocktail.ps1

Cut and paste Import-module

Import-Module ..\CocktailFinder\CocktailModule\bin\debug\CocktailModule -Force

Cut and paste $ingredients line

$ingredients=Get-Ingredient

Display list

$ingredients

$ingredients[‘Gin’]

Next do the alcoholic bit

$alcoholic = $ingredients.Values | Where-Object {$\_.AlcoholByVolume -gt 30}

get-recipe -Ingredient $alcoholic

$alcoholic | get-recipe

Do the ,$() bit followed by get-recipe with names

$negroni = get-recipe ‘Negroni’  
$negroni  
$negroni.Ingredients

Lets load in a format file

Flip to VS2015 and show the format file

Do $negroni.Ingredients

$negroni

Change the format for Cocktail.recipe change script block to

<ScriptBlock>$\_.Ingredients | Format-Table </ScriptBlock>

.\updateformat.ps1 in powershell ISE

Do $negroni again, show the mess of format stuff

Mention folks suggest nesting using GroupBy and copying from the builtin filesystem format

Better solution Add Out-String to script block

Reimport and $negroni

Go back to visual studio and uncomment the line in module manifest

Checked in at this point branch Slide13\_2

$alcoholic | get-recipe doesn’t do anything.

Why nothing… Powershell expands out the ingredients and we process them one at a time and a drink with just one ingredient isn’t a cocktail.

get-service | Where-Object {$\_.Name -like '\*MS\*'}

If we pipe this into stop-service it would stop the services.

Psedit .\getcommand.ps1

Run it

Talk about InputObject

Close down Powershell ISE

Change branch to Slide13\_3

I’ve already created some powershell tests…

Lets talk through them and run them

Uncomment our final test.

Put breakpoint in get-recipe on Can-Make line Lets fix it, add variable

private List<Ingredient> inStock = null;

Initialise list in our BeginProcessing

inStock = new List<Ingredient>();

In processRecord

foreach(var ingredient in InputObject)

{

inStock.Add(ingredient);

}

Cut and paste code in EndProcessing

var finder = new RecipeFinder(recipes);

var canMake = finder.GetByAvailableIngredients(inStock);

foreach (var name in canMake)

{

WriteObject(recipes[name]);

}

Wrap in if statement

Checked in as Slide13\_4