Windows PowerShell 4.0 For the IT Professional - Part 1

Module 17: Modules

Student Lab Manual

Version 2.0

Conditions and Terms of Use

Microsoft Confidential - For Internal Use Only

Copyright and Trademarks

Contents

Lab 17: Modules

Introduction

A module is a package that contains Windows PowerShell commands, such as cmdlets, providers, functions, workflows, variables, and aliases. People who write commands can use modules to organize their commands and share them with others. People who receive modules can add the commands in the modules to their Windows PowerShell sessions and use them just like the built-in commands.

Objectives

After completing this lab, you will be able to:

- Understand the different ways to collect functions, scripts and related functionalities into "modules" that can be persisted and reused.

Scenario

You are a developer or administrator of Windows systems within Contoso.com. You wish to share your scripts and custom cmdlets with others.

Prerequisites

Start all VMs provided for the workshop labs.

Logon to WIN8-WS as:

Username: **Contoso\Administrator**

Password: **PowerShell4**

Estimated time to complete this lab

45 minutes

NOTE: These exercises use many Windows PowerShell commands. You can type these commands into the Windows PowerShell Integrated Scripting Environment (ISE) or the Windows PowerShell console. For some exercises, you can load pre-typed lab files into the Windows PowerShell ISE, allowing you to select and execute individual commands. Each lab has its own folder under **C:\PShell\Labs** on **WIN8-WS**.

Some exercises in this workshop may require running the Windows PowerShell console or ISE as an elevated user (Run as Administrator).

We recommend that you connect to the virtual machines (VMs) for these labs through Remote Desktop rather than connecting through the Hyper-V console. This allows you to use copy and paste between VMs and the host machine. If you are using the online hosted labs, then you are already using Remote Desktop.

Exercise 17.1: Module Types

Introduction

Outline the four kinds of modules and explore where they are located.

Objectives

After completing this exercise, you will be able to:

- Identify the four types of modules
- Review where they are located
- Understand the contents of a module

Task 17.1.1: Script Modules

A script module is a file (.psm1) that contains any valid Windows PowerShell code. Script developers and administrators can use this type of module to create modules whose members include functions, variables, and more.

1. Explore the primary Windows PowerShell module environment for .psm1 (script module) files.

```
Get-ChildItem $PSHOME\modules\*.psm1 –recurse
```

2. Examine the contents of a .psm1 module file

```
Get-Content $PShome\modules\SMBshare\SmbScriptModule.psm1 | more
```

Task 17.1.2: Binary Modules

A binary module is a Windows PowerShell module whose root module is a .NET Framework assembly (.dll) that contains compiled code. Cmdlet developers can use this type of module to create modules that contain cmdlets, providers, and more.

1. Explore the Windows PowerShell module environment for .dll files.

```
Get-ChildItem $PSHOME\modules\*.dll –Recurse
```

Task 17.1.3: Manifest Modules

A manifest module is a module that includes a manifest (described in a following section) to describe its components, but that does not specify a root module in the manifest. A module manifest does not specify a root module when the "ModuleToProcess" key of the manifest is blank. In most cases, a manifest module also includes one or more nested

modules using script modules or binary modules. A manifest module that does not include any nested modules is useful when you want a convenient way to load assemblies, types, or formats.

1. Explore the Windows PowerShell module environment for manifest data (.psd1) files that have the key value ModuleToProcess=''.

```
$file = Get-ChildItem $PShome\modules\*.psd1 –Recurse |
Where-Object {Get-Content $_ | Select-String -Pattern "ModuleToProcess = ''"}

$file
```

Take note of the type of single/double quotes used in the pattern parameter value and the spaces around the equals sign!

2. Examine the contents of an identified .psd1 manifest file

```
Get-Content $file | more
```

Task 17.1.4: Dynamic Modules

A dynamic module is a module that does not persist to disk. This type of module enables a script to create a module on demand that does not need to be loaded or saved to persistent storage. By default, dynamic modules created with the New-Module cmdlet intend to be short-lived and therefore not accessible by the Get-Module cmdlet.

1. Create an in-memory module with a single simple function.

```
New-Module –ScriptBlock {function Hello {“Hello!"}}
Get-Module
```

2. Note that while Get-Module does not report the Dynamic module, the function(s) defined will be available.

```
Get-Command Hello
```

This simple example is enough for now. A dynamic module may strike you as strange, but for remote sessions, it is a powerful way to package up script or code, stream it over to the target and process it into a loadable module available on that target for the life of the session.

Exercise 17.2: Module Cmdlets and Variables

Introduction

Outline the basic Cmdlets for managing modules.

Objectives

After completing this exercise, you will be able to:

- Understand the primary cmdlets to manage modules.
- Understand the variables that can influence module load behavior.

Task 17.2.1: Get Module

1. List the modules currently loaded

```
Get-Module
```

2. List the commands, per module

```
Get-Command | Sort ModuleName, Name | FT Name, ModuleName -GroupBy ModuleName
```

3. List the available modules found under $PSmodulepath

```
Get-Module -ListAvailable
```

4. List the available system modules by selecting those that are under System32. The list will vary depending on operating system, features installed, etc.

```
Get-Module -ListAvailable | Where-Object {$_.Path -Match "System32"}
```

5. List the available modules that are elsewhere

```
Get-Module -ListAvailable | Where-Object {$_.Path -NotMatch "System32"}
```

6. Review the modules located within $PSHOME, the Windows PowerShell home.

```
Get-Childitem $PSHOME\modules
```

7. Consider you may have (or wish to) deploy modules for yourself or for groups, etc. The $PSmodulePath will show the various potential (or actual) locations.

```
$Env:PSmodulePath
```

8. Do you have any modules in your own location, on this machine?

```
Get-Childitem "$HOME\Documents\WindowsPowerShell\Modules"
```

Task 17.2.2: Import Module

This command loads the module into memory and imports the public commands from that module.

> **NOTE:** Beginning in Windows PowerShell 3.0, installed modules are automatically imported to the session when you use any commands or providers in that module.

1. Import the Active Directory module explicitly.

```
Import-Module ActiveDirectory
```

2. Confirm it is now available

```
Get-Module
```

3. Review the commands that are now available from that module

```
Get-Command –Module ActiveDirectory
```

4. It is possible to alter the automatic import behavior by setting a preference variable. See: Get-Help About_Preference_Variables

```
$PSModuleAutoLoadingPreference="None"
$PSModuleAutoLoadingPreference="All"
```

Task 17.2.3: Remove Module

This command removes the module from memory and removes the imported members.

1. Remove the Active Directory module explicitly.

```
Remove-Module ActiveDirectory
```

2. Confirm that it no longer shows as loaded.

```
Get-Module
```

Exercise 17.3: $PSModulePath

Introduction

Explore the paths where modules are located.

Objectives

After completing this exercise, you will be able to:

- Understand and identify the locations searched to load a module.

Task 17.3.1: $PSModulePath

This environment variable contains a list of the directories in which Windows PowerShell modules are normally stored and searched. Windows PowerShell uses the value of this variable when importing modules automatically and updating Help topics for modules.

1. Examine the value of this environment variable.

```
$Env:PSModulePath
$Env:PSModulePath -split ";"
```

2. Extract the first path from the list, using split to generate an array of paths.

```
($Env:PSModulePath -split ";")[0]
```

3. Now for each location selected as above, look to see what (if anything) is present.

```
Get-Childitem ($Env:PSModulePath -split ";")[0]
Get-Childitem ($Env:PSModulePath -split ";")[1]
Get-Childitem ($Env:PSModulePath -split ";")[2]
Get-Childitem ($Env:PSModulePath -split ";")[3]
```

Exercise 17.4: Creating a Script Module

Introduction

In this exercise, you will create a simple module from an existing script.

Objectives

After completing this exercise, you will be able to:

- Understand how to take scripts and convert them into modules.

Task 17.4.1: Script Module

A script module is a file (.psm1) that contains any valid Windows PowerShell code. We will take an existing script file (.ps1), convert it to a "script module" (.psm1), and then deploy it.

1. First, review the destination of deployed Modules.

```
$Env:PSModulePath
```

2. Next, we will define where our source script resides and view it.

```
$lab = "C:\PShell\Labs\Lab_17"
Get-Childitem $lab\ACEmodule.ps1
Get-Content $lab\ACEmodule.ps1
```

3. Review the function(s) available within the script. All functions will be made visible by default, unless we are explicit about which functions are exported. The Export-ModuleMember command allows the module author to expose some commands and hide others from the module user. At the bottom of the file this command exports just the "New-*" functions.
4. Copy the file and rename the extension to ".psm1" (a script module)

```
Copy-Item $lab\ACEmodule.ps1 $lab\ACEmodule.psm1 -Force –Verbose
```

5. Choose our local user module location and create the target module folder.

```
$MyModules=($Env:PSmodulePath -split ";")[0]
$MyModules
New-Item –Path $MyModules\ACEmodule –ItemType directory
```

6. Now that we have the correct location and folder, we can copy our .psm1 file to the Module folder.

```
Copy-Item $lab\ACEmodule.psm1 $MyModules\ACEmodule -Force -Verbose
```

7. Our module will now appear in the list of available modules.

```
Get-Module -ListAvailable ACEmodule | Format-List -Property Name, Path
```

8. Import the module.

```
Import-Module -Verbose ACEmodule
```

9. Confirm that the module commands are now available.

```
Get-Command -Module ACEmodule
```

10. There is a special automatic variable used inside script modules: $PSScriptRoot. This variable contains the directory from which the script module is executed. It enables scripts to use the module path to access other resources. In Windows PowerShell 2.0, this variable is valid only in script modules. Beginning in Windows PowerShell 3.0, it is valid in all scripts.

```
# Create a test script module that shows $PSScriptRoot and $PSCommandPath
'Write-Host "`$PSScriptRoot is:`t" $PSScriptroot; Write-Host "`$PSCommandPath
is:`t" $PSCommandPath' > $Env:tmp\CmdPath.psm1

# $PSScriptRoot is the full path to the containing folder
# $PSCommandPath is the full path to the file
# Import the module to see the difference
Import-Module $Env:tmp\CmdPath.psm1

# This works from scripts (.ps1) as well as script modules (.psm1)
# Try it from a script
'"`$PSScriptRoot is:`t" + $PSScriptroot; "`$PSCommandPath is:`t" + $PSCommandPath'
> $Env:tmp\CmdPath.ps1

& $Env:tmp\CmdPath.ps1
```

Exercise 17.5: Module Manifest

Introduction

In this exercise, you will create some simple module manifests. A module "manifest" is a text file with a .psd1 file extension that contains a hash table. The keys and values in the hash table do the following:

- Describe the contents and attributes of the module.
- Define the prerequisites.
- Determine how the components are processed.

Objectives

After completing this exercise, you will be able to:

- Create a module manifest.

Task 17.5.1: New-ModuleManifest (simple)

We shall create a module manifest for the ACEmodule we created earlier.

> **NOTE:** Manifests are not required for a module.
>
> Modules can reference script files (.ps1), script module files (.psm1), manifest files (.psd1), formatting and type files (.ps1xml), cmdlet and provider assemblies (.dll), resource files, help files, localization files, or any other type of file or resource that is bundled as part of the module.
>
> For an internationalized script, the module folder also contains a set of message catalog files. If you add a manifest file to the module folder, you can reference the multiple files as a single unit by referencing the manifest.

1. Reference our local user module location and confirm the target module folder (ACEmodule) is present.

```
$MyModules=($Env:PSmodulePath -split ";")[0]
Get-Childitem $MyModules
```

2. Create the module manifest within that target location. Specify that this module is the root module for this manifest.

```
New-ModuleManifest -RootModule ACEmodule -Path $myModules\ACEmodule\ACEmodule.psd1
```

3. Import the module, which will load via the manifest, and confirm its commands are available. Use the -Force parameter to overwrite the currently-imported module version.

```
Import-Module ACEmodule -Force
Get-Command -Module ACEmodule
Get-Module ACEmodule -List | Format-List -Property *
```

4. Get the content of the manifest file and review it.
5. If we have a heavily customized manifest with multiple references, we can verify a module manifest accurately describes the components of a module by verifying the files listed in the module manifest file (.psd1) exist in the specified paths.

```
Test-ModuleManifest -Path $myModules\ACEmodule\ACEmodule.psd1
```

Task 17.5.2: New-ModuleManifest (complex)

Create a module manifest that combines the use of a ***.psm1*** and ***.dll*** (binary). Use the module to create and manipulate Microsoft Office documents.

1. Define where our source .psm1 script module and .dll are located.

```
$lab = "C:\PShell\Labs\Lab_17"
```

2. Reference our local user Module location and create the target module folder (OFFICEmodule) for our new module.

```
$MyModules=($Env:PSmodulePath -split ";")[0]
New-Item -Path $MyModules\OFFICEmodule –ItemType directory
```

3. Create the module manifest within that target location. We will specify that this module is the root module for this manifest and specify the .dll this module must load.

> **NOTE:** The back-tick at the end of the lines below is purely for readability, and you would usually issue this as one long command.

```
New-ModuleManifest -RootModule OFFICEmodule `
                   -RequiredAssemblies "DocumentFormat.OpenXml.dll" `
                   -Path $myModules\OFFICEmodule\OFFICEmodule.psd1
```

4. Copy the .dll and .psm1 script module into the target module folder.

```
Copy-Item $lab\DocumentFormat.OpenXml.dll $myModules\OFFICEmodule -Force
Copy-Item $lab\OFFICEmodule.psm1 $MyModules\OFFICEmodule -Force
Get-Childitem -Path $myModules\OFFICEmodule
```

5. Our module should now be available to import.

```
Import-Module OFFICEmodule –Force
Get-Command –Module OFFICEmodule
```

6. Test our new command by creating a Word document, using the Get-SystemInfo script created in Module 11.

```
. C:\PShell\Labs\Lab_11\Get-SystemInfo.ps1 # dot-source script with the function
```

```
New-WordDocument -DocumentName "SystemInfo" -TextData (Get-SystemInfo)
```

7. Confirm the creation of the new Word document. It should be located in the root of the Documents folder. Ensure the document contains output from the Get-SystemInfo function.

Exercise 17.6: Script Module Signing

Introduction

We have created some simple modules. However, you should digitally sign your modules to ensure compatibility when the Windows PowerShell execution policy is changed to allow only signed scripts to execute.

Objectives

After completing this exercise, you will be able to:

- Digitally sign a script module.

Task 17.6.1: Script signing

Earlier we created the ACEmodule script module.

1. Launch the Windows PowerShell ISE with "Run as Administrator".
2. Reference the local user Module location and confirm the target module folder (ACEmodule) is present.

```
$myModules=($env:psmodulepath -split ";")[0]
$loc="$myModules\ACEmodule"
Get-Childitem $loc
Get-Content $loc\ACEmodule.psm1
```

3. Enforce the signing of scripts and attempt to import the unsigned module.

```
Set-ExecutionPolicy AllSigned -Force
Import-Module ACEmodule -Force
```

What error do you receive?

FAIL! All scripts MUST be signed.

4. We are now required to sign the script. First, determine whether we have any code-signing digital certificates available.

```
Get-ChildItem CERT: -Recurse -CodeSigning
```

A certificate is already deployed and located in CurrentUser\My

5. Get the code-signing certificate, from the Cert: PSDrive, to sign the script module.

```
$cert=(Get-ChildItem -Path CERT:\CurrentUser\My -CodeSigningCert)
```

6. Sign the script using the certificate.

```
Set-AuthenticodeSignature -FilePath $loc\ACEmodule.psm1 -Certificate $Cert
```

7. Examine the newly signed script module.

```
Get-Content $loc\ACEmodule.psm1
```

8. Confirm we can load the script module. Use the –Force switch parameter to overwrite the module if it was previously imported.

```
Import-Module ACEmodule –Force
Get-Module
```

Exercise 17.7: Zip it!

Introduction

In this workshop, we have covered the fundamentals of Windows PowerShell. We encourage you to take the scripts examples to use, both at work and in subsequent Windows PowerShell workshops.

Objectives

After completing this exercise, you will be able to:

- Compress the C:\PShell folder to take with you.

Task 17.7.1: Zip the C:\PShell folder

.NET 4.5 gives us the ZIP capabilities
System.IO.Compression.ZipFile

Compress the C:\PShell and $MyModules folders.

1. Launch the Windows PowerShell ISE with "Run as Administrator".
2. Reference our local user Module location and confirm the target module folder (ZIP) is present.

```
$myModules=($env:psmodulepath -split ";")[0]
$loc="$myModules\ZIP"
Get-Childitem -Path $loc
Get-Content $loc\ZIP.psm1
```

3. Turn off the enforced signing from the previous exercise.

```
Set-ExecutionPolicy RemoteSigned -Force
Import-Module ZIP -Force
```

4. Review the commands available from this module.

```
Get-Command –Module ZIP
Get-Help New-ZipFile –Full
```

5. Now, using this command, zip the folders.

```
New-ZipFile –SourceDirectory C:\Pshell\Labs –DestinationArchive $HOME\pshell.ZIP
New-ZipFile –SourceDirectory $myModules –DestinationArchive $HOME\modules.ZIP
```