Windows PowerShell 4.0 For the IT Professional - Part 1

Module 4: Commands 2

Student Lab Manual

Version 2.0

Conditions and Terms of Use

Microsoft Confidential - For Internal Use Only

Copyright and Trademarks

Contents

Lab 4: Commands 2

Introduction

Windows PowerShell commands can be grouped together and executed as a single unit. In this exercise we will explore this by enclosing commands in script blocks or functions.

Objectives

After completing this lab, you will be able to:

- Use a script block to execute remote commands
- Create a reusable function that accepts input parameters and returns output.

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

60 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 4.1: Script blocks

Introduction

Script blocks are a collection of one or more Windows PowerShell commands executed as a single unit. A script block can accept arguments and return values.

Many aspects of Windows PowerShell use script blocks, including Cmdlets, Functions, Workflows and Desired State Configuration declarations.

This exercise provides experience with script block syntax and usage.

Objectives

After completing this exercise, you will be able to:

- Create a script block
- Execute a script block
- Use a script block to execute commands on remote machines

Task 4.1.1: Script block Syntax & Usage

1. A script block is defined as one or more valid Windows PowerShell commands contained between curly braces '{ }'.
   Type the command below in the Windows PowerShell ISE script pane and review the output.

```
{Get-Winevent –LogName System –MaxEvents 20}
```

2. The previous example defined a script block, returned the body of the script block (list of commands) to the console, but failed to execute it. Next, we will assign the script block to a variable, $sb.

> We will cover variables in more detail later in the course, but as long as the variable name starts with a '$' symbol it can be named almost anything you like.

```
$sb = {Get-Winevent –LogName System –MaxEvents 20}
```

3. Type the variable name and press enter.

```
$sb
```

4. Again, we receive the same output. To actually execute the command contained within the script block, simply prefix the variable with the '&' symbol – known as the "Call operator". A space character is optional between the call operator and variable.

```
& $sb
# Or
```

```
&$sb
```

5. You may be wondering why we create a script block at all. After all, typing the command directly, without the curly braces, will execute the command. One of the most powerful aspects of the shell is the ability to execute commands remotely.

   Review the syntax for the Get-WindowsOptionalFeature Cmdlet.
   Can this Cmdlet be executed against multiple remote machines with a single command? (Does it take a parameter for an array of computer names?)
   Yes / No

NO. Does not have the –ComputerName parameter

6. The Invoke-Command Cmdlet provides the functionality to execute a single command on many remote machines. The Cmdlet's –Scriptblock parameter requires a script block as a value. Type and execute the command below.

```
Invoke-Command -ComputerName 2012R2-MS –ScriptBlock {
Get-WindowsOptionalFeature -Online -FeatureName MediaPlayback
}
```

7. Review the help for Invoke-Command. How can we execute this command on multiple machines? Adjust the previous command to execute on the 2012R2-DC, 2012R2-MS and WIN8-WS machines and record it below.

```
Invoke-Command -ComputerName 2012R2-MS,WIN8-WS,2012R2-DC -ScriptBlock {
Get-WindowsOptionalFeature -Online -FeatureName MediaPlayback
}
```

8. Script blocks, like Cmdlets, can accept named input parameters. The Param() statement enables a script block to do this. Note that script blocks use exactly the same syntax as a Cmdlet.
   The Param() syntax is shown below.

```
{
Param ($parameter1, $parameter2, $parameter<n>)

<statement list>
}
```

9. Modify the previous script block to accept one input parameter by adding the Param() statement. Type and execute the following command. The script block can be saved in a variable ($sb) so that we can reuse it.

```
$sb = {param ($featureName) Get-WindowsOptionalFeature -Online -FeatureName
$featureName}
```

10. Execute the script block. Supply "WindowsMediaPlayer" as the value for the script block's –featureName parameter.

```
&$sb –featureName “WindowsMediaPlayer”
```

11. Now execute the script block using Invoke-Command. Specify the Script block’s –featureName parameter value for Invoke-Command’s –ArgumentList parameter.

```
Invoke-Command –ComputerName 2012R2-DC, 2012R2-MS, WIN8-WS –Scriptblock $sb `
–ArgumentList “WindowsMediaPlayer”
```

12. Finally, extend the command to export the output in XML format to the file C:\PShell\Labs\Lab_4\RemoteMediaFeature.xml.

```
Invoke-Command –ComputerName 2012R2-DC,
2012R2-MS, WIN8-WS –Scriptblock $sb `
–ArgumentList “WindowsMediaPlayer” |
Export-Clixml –Path
C:\PShell\Labs\Lab_4\RemoteMediaFeature.xml
```

Exercise 4.2: Functions

Introduction

Functions are "named" script blocks. Therefore, everything learned about script blocks in the previous exercise also applies to functions.

Objectives

After completing this exercise, you will be able to:

- Create & execute functions
- Specify function input parameters

Task 4.2.1: Function Syntax

1. The purpose of a function is to collect a sequence of commands together into a single unit with a unique name. A function is defined using the function keyword, unique name and input parameters, followed by a pair of curly braces containing the commands.

```
function <name> (parameters) {
     <statement list>
}
```

   Type the following code exactly as shown below into the Windows PowerShell ISE. Press <F5> to run it. Do you receive any output?

NO. Function is now defined. It has not yet been executed.

```
function Get-SystemInfo {

Get-Volume | Sort-Object SizeRemaining –Descending
Get-NetAdapter | Select-Object –Property Name, Interface*, Status
}
```

2. A function must be defined before it can be executed - usually referred to as 'calling' a function. To call a function, simply type its name and press <Enter>
   Type 'Get-'and wait for the IntelliSense menu to appear then type 'sys'. The new function name is highlighted.

   Press <Enter> to select it then press <Enter> run it.

```
Get-SystemInfo
```

Task 4.2.2: Function Parameters

1. We can also add named parameters to a function. One syntax for this is shown below. The param() statement allows a function to accept one or more input parameters. Parameters are declared as variables and separated by commas.

```
function <name> {

param($parameter1,$parameter2)

    <statement list>
}
```

The example below adds a parameter called $machineName in the Param() statement, that takes the name of a computer as its value. A remote CIM session (covered later in this course) then executes remotely using the supplied computer name.

```
Function Get-SystemInfo {

param($machineName)

$session = New-CimSession –ComputerName $machineName

Get-Volume –cimsession $session
Get-NetAdapter –cimsession $session |
Select-Object –Property Name, InterfaceIndex, Status, PSComputerName
}
```

> We can also use an alternate simplified syntax where the param() statement isn't used. Instead, the function input parameters are defined in parentheses, directly after the function name. This syntax is similar to other scripting and programming languages.

```
function <name> ($parameter1, $parameter2) {

    <statement list>
}
```

2. Execute the Get-SystemInfo function against one virtual machine.

```
Get-SystemInfo –machineName "2012R2-DC"
```

> NOTE: Just like Cmdlet parameters, when calling a function do not enclose the parameters in parentheses.

3. Modify the Get-SystemInfo function to include an additional parameter called $Timeout. Within the function, use this variable as the value for the New-CimSession's –OperationTimeoutSec parameter.

   Save the function in the following file C:\PShell\Labs\Lab_4\Get-SystemInfo.ps1.

```
Function Get-SystemInfo {

param($machineName, $timeOut)

$session = New-CimSession –ComputerName
$machineName –OperationTimeoutSec $timeOut

Get-Volume –cimsession $session
Get-NetAdapter –cimsession $session |
Select-Object –Property Name,
InterfaceIndex, Status, PSComputerName
}
```

Exercise 4.3: Remote Commands

Introduction

Windows PowerShell includes a robust remote command execution infrastructure, known as Windows PowerShell Remoting. You can run remote commands on a single computer or on multiple computers by using a temporary or persistent connection. You can also start an interactive session with a single remote computer.

In this exercise, we will explore these remote execution methods to perform administrative operations.

Objectives

After completing this exercise, you will be able to:

- Use Cmdlets that support the –ComputerName parameter
- Create persistent and non-persistent remote sessions to multiple machines
- Create interactive remote sessions

Task 4.3.1: Remote operations with –ComputerName parameter

1. Many Cmdlets implement the -ComputerName parameter to allow the Cmdlet to access and execute against a remote machine. The protocol used to achieve this is transparent to the user and the success of the command may depend on many different firewall ports being open.

   Type the command below to list Cmdlets that support the –ComputerName parameter.

```
Get-Command –ParameterName ComputerName
```

2. Use the Get-Service Cmdlet to list the services on the remote machine 2012R2-DC.

```
Get-Service –ComputerName 2012R2-DC
```

3. From the WIN8-WS machine, use the Get-EventLog Cmdlet's –ComputerName parameter to list the 10 newest events, from the System event log, on the computer 2012R2-MS.

   Get-EventLog -Newest 10 -ComputerName 2012R2-MS

4. Finally, restart the remote 2012R2-MS machine using a Cmdlet that supports the –ComputerName parameter. Execute the command you used to do this and then write it below.

   Restart-Computer –ComputerName 2012R2-MS -Force

Task 4.3.2: Invoking remote commands

1. The Invoke-Command Cmdlet uses the Windows PowerShell Remoting infrastructure to run commands on one, or more, remote machines. Via a single remote network port, Windows PowerShell creates a new session running on the remote machine. The session is removed once the command has completed and the output has been returned.

   To execute a command on a remote machine use the Invoke-Command Cmdlet. Specify the block of commands you wish to run as the argument to the –Scriptblock parameter. Ensure the parameter value is contained within curly braces {}.
   Type the command below. This command starts a new process on the remote machine 2012R2-MS.

```
Invoke-Command –ComputerName 2012R2-MS –Scriptblock {Start-Process notepad.exe -
PassThru}
```

2. Note the Id property value returned by the previous command and use it to display the new process on the remote machine.

```
Get-Process –ComputerName 2012R2-MS –Id <Id>
```

   The process no longer exists because the remote PowerShell session that launched it has already ended.

3. Get the volume information within a remote session on the 2012R2-MS machine and store the result in a new variable named volInfo.

```
Invoke-Command –ComputerName 2012R2-MS –Scriptblock {Get-Volume –OutVariable
volInfo}
```

4. Next, get the volInfo variable on the remote machine using Invoke-Command.

```
Invoke-Command –ComputerName 2012R2-MS –Scriptblock {Get-Variable –Name volInfo}
```

   What does this command return?

   Nothing. The session and namespace has been destroyed

5. For example, you may want to disable the Windows Update service in a test environment. Accomplish this by invoking PowerShell on remote machines.

```
Invoke-Command –ComputerName 2012R2-MS, 2012R2-DC, WIN8-WS –Scriptblock {

Get-Service –DisplayName “Windows Update” |
Set-Service –StartupType Disabled

Get-Service –DisplayName “Windows Update” |
Stop-Service
}
```

Task 4.3.3: Persistent remote sessions

1. As we saw in the previous task, Invoke-Command creates a new remote Windows PowerShell session each time it is executed, then removes the session after the command completes, therefore resources will not persist.
   To overcome this limitation we can create a persistent remote Windows PowerShell session.
   Create a new session between the local machine and 2012R2-MS, using the code below and save it in the variable $mySession.

```
New-PSSession –ComputerName 2012R2-MS –Outvariable mySession
```

2. List the new session using the Get-PSSession Cmdlet.

```
Get-PSSession
```

3. Invoke a command against the new session to create and populate the $volInfo variable. The $mySession variable, identifying the persistent session, is used with the -Session parameter.

```
Invoke-Command –Session $mySession –Scriptblock {Get-Volume –OutVariable volInfo}
```

4. Now read the $volInfo variable value from the remote session. This time the value should contain the name of the remote machine, proving the remote session is in fact persistent.

```
Invoke-Command –Session $mySession –Scriptblock {Get-Variable –Name volInfo -
valueOnly}
```

5. Find the Cmdlet that can remove the session. Execute the command and record it below.

   Remove-PSsession –Session $mySession

   _____

   _____

6. Commands may be invoked on many remote machines at once. Create a new variable that contains the names of all machines in the lab environment.

```
Get-ADComputer –Filter * |
Select-Object –ExpandProperty DNSHostName –Outvariable computers
```

7. Create new sessions to each remote machine name contained in the $computers variable.

```
New-PSSession –ComputerName $computers –Outvariable remoteSessions
```

8. Execute the following script block against all of the sessions stored in the $remoteSessions variable. This will start a new instance of notepad.exe on each machine.

```
Invoke-Command –session $remoteSessions –Scriptblock {
Start-Process –FilePath Notepad.exe –Passthru
}
```

9. Confirm the notepad processes are still running

```
Get-Process –ComputerName $computers –Name notepad
```

10. Using the persistent session, stop the notepad processes on each machine.
11. Remove the session using the appropriate Cmdlet.

Task 4.3.4: Interactive remote sessions

1. One of the ways to execute a Windows PowerShell command remotely is to create a remote interactive session. When the session starts, the commands that you type run on the remote computer, just as though you typed them directly on the remote computer. You can connect to only one computer at a time in each interactive session. Enter the command below to create an interactive session to 2012R2-DC.

```
Enter-PSSession –ComputerName 2012R2-DC
```

   Windows PowerShell prefixes the prompt with the remote computer name, in square brackets.
2. Execute the command below.

```
Get-CimInstance -ClassName Win32_ComputerSystem
```

   Does it display the local or remote computer information?

   Remote

   _____

3. Disconnect from the remote session

```
Exit-PSSession
```