0. TOC

13 May 2020 10:15 PM

S.NO	PRIMARY TOPICS	SUB TOPICS	POINTS TO BE COVERED
1	**PowerShell Introduction**	**What is Powershell ?**	**Compare with Shell in linux Evolution of Powershell**
		Why Powershell ?	Advantages and Uses
		Versions	Powershell Versions Differences between the versions and additional features
		How to use Powershell Interactively ?	Features explanation in short Difference when compared with Command Prompt
		Powershell Help	Update-Help How to use help and all its properties
2	**DataStructures**	**Variables**	**How to store command outputs Notations Used**
		Arrays	How command Results are stored How to read an array and How to manipulate it
		Hashtables	Creating, Updating and Reading Hashtable
		Defining Datatypes	How to use strongly typed datatype
3	**Objects**	**Creating Objects**	**What does an Object Mean ? New-object**
		Using Properties	Adding Properties Read-Only and Read-Write Properties Types of Properties
		Using Methods	Get_ and Set_ Methods Standard Methods Method Calls and Arguments Method Signatures
		Using Real Examples	Use some applications of objects and explain
4	**Conditional**	**Conditional Operators**	**Overview on conditional operators used Logical Operators and Use**
		Where-object	Use of where-object with example
		If-ElseIf-Else	How to use in Powershell
		Switch	Syntax in powershell
5	**Loops**	**ForEach-Object**	**Looping with all conditions**
		Foreach	

		Do and While	Continuation and Abort Conditions
		For	Nested Loops
		switch	How to use switch for looping
		continue statements	Use of the continue statement Labels
6	**Functions and Pipelines**	**Creation and Calling**	**How to create a function** **How to call a function** **Defining parameters**
		Using Pipelines	Explain some of the usage of pipelines with examples
7	**Script Execution**	**Creating and Launching**	**How to create a script file** **Launching the Script**
		Executing Policy	Settings for execution policy
		Arguments handling	Scopes - Variables Profile Scripts
		ConsoleFiles	Use of console files
		Digital Signatures	How to sign a script
8	**Error Handling**	**Suppressing Errors**	**How to suppress the error from not being thrown**
		Try / Catch	How to use try / Catch statements
		Using traps	Use of trap statements
		Exceptions	Understand Exceptions Handle Exceptions Throwing our own Exceptions
		Stepping and Tracing	How to do debug with step and trace
9	**Input / Output**	**Command Line Arguments**	**Using Command line arguments in Powershell (param)**
		Console Read / Write	
		File Read / Write	
10	**Text Processing and Regular Expressions**	**String Formatting**	**Defining the String** **How do we format strings in powershell**
		String Operators	Operators used to process the String
		String Object Methods	Methods availabled which can be used on String Objects
		Regular Expression Pattern and Matching	Defining the patterns How to match the patterns
		Regular Expression activities	Keywords Segmenting Replace Symbols to be known
11	**Configuration using XML**	**XML Parsing**	**How to read the nodes and Data structure used** **Using it as config file**

12	**Windows Registry**	**Creating Keys**	**Available Providers** **Creating Drives and then keys**
		Reading Keys	Reading a key Reading multiple keys
13	**Processes, Services and Event Log Management**	**Process Management**	**Reading the process list** **Getting Process Info** **Calling Process Methods (Kill)**
		Service Management	Reading the service list Service Methods (Start, Stop, Pause, Resume)
		Event Log Management	Reading the logs Writing the log entries
14	**WMI Management**	**Using WMI Objects**	**Exploring the Classes** **Reading the Properties** **Filters**
		Using WMI Methods and properties	Changing the properties Calling the methods
15	**Remote Execution**	**Using Remote Execution**	**How to execute commands remotely?** **Methods for remote execution**
		Using Invoke	Invoke-Session Invoke-Command Passing credential & credssp
		Remote WMI	Connecting to Remote system WMI Provider Passing credentials for authentication
16	**Workflow**	**Using Workflow**	**What is Work flow and its uses?**
		Workflow Concepts	Working with workflows with examples
			Inline script with examples Inline to Inline calling and passing/getting inputs/outputs
			Sequence and Parallel Execution Real time examples
17	**Desired State Configuration (DSC)**	**What is DSC?**	**Introduction to DSC**
		Why DSC?	Use of DSC DSC Features
		Concepts	Overview of the DSC Concepts Working of DSC with basic examples

1. PowerShell Introduction

13 May 2020 10:16 PM

PowerShell Introduction	What is Powershell ?	Compare with Shell in linux Evolution of Powershell
	Why Powershell ?	Advantages and Uses
	Versions	Powershell Versions Differences between the versions and additional features
	How to use Powershell Interactively ?	Features explanation in short Difference when compared with Command Prompt
	Powershell Help	Update-Help How to use help and all its properties

2. DataStructures

13 May 2020 10:16 PM

DataStructures	Variables	How to store command outputs Notations Used
	Arrays	How command Results are stored How to read an array and How to manipulate it
	Hashtables	Creating, Updating and Reading Hashtable
	Defining Datatypes	How to use strongly typed datatype

Array:

- An array is a data structure that is designed to store a collection of items. The items can be the same type or different types.

Creating and initializing an array
$A = 22,5,10,8,12,9,80

To create a single item array named $B containing the single value of 7, type:
$B = ,7

You can also create and initialize an array by using the range operator (..)
$C = 5..8

to determine the data type
$A.GetType()

To create a strongly typed array, cast the variable as an array type, such as string[], long[], or int32[].
[int32[]]$ia = 1500,2230,3350,4000

You can use the array operator to create an array of zero or one object.
$a = @("Hello World")
$a.Count

$b = @()
$b.Count

The array operator is useful in scripts when you are getting objects, but do not know how many objects you get
$p = @(Get-Process Notepad)

Reading an array:
$a
$a[0]
$a[2]
$a[1..4]

$a = 0 .. 9
$a[-3..-1]

$a = 0 .. 9
$a[-1..-3]

Properties of arrays:

Count or Length or LongLength

```
$a = 0..9
$a.Count
$a.Length
```

Building a multidimensional array;

```
$a = @(
  @(0,1),
  @("b", "c"),
  @(Get-Process)
)

[int]$r = $a.Rank
"`$a rank: $r"
```

HASH TABLES:

A hash table, also known as a dictionary or associative array, is a compact data structure that stores one or more key/value pairs.

The syntax of a hash table is as follows:
@{ <name> = <value>; [<name> = <value>] ...}

Creating Hash Tables
To create a hash table, follow these guidelines:

Begin the hash table with an at sign (@).
Enclose the hash table in braces ({}).
Enter one or more key/value pairs for the content of the hash table.
Use an equal sign (=) to separate each key from its value.
Use a semicolon (;) or a line break to separate the key/value pairs.

To create an empty hash table:

```
$hash = @{}

$hash = @{ Number = 1; Shape = "Square"; Color = "Blue"}
```

Displaying Hash Tables:

```
$hash
```

Hash tables have Keys and Values properties:

```
$hash.keys
$hash.values

$hash.Number
$hash.Color

$hash.count
```

Adding and Removing Keys and Values:

```
$hash["<key>"] = "<value>"
```

For example:
$hash["Time"] = "Now"

add keys and values to a hash table by using the Add method:
Add(Key, Value)

For example,
$hash.Add("Time", "Now")

you can add keys and values to a hash table by using the addition operator (+):
$hash = $hash + @{Time="Now"}

You can also add values that are stored in variables:
$t = "Today"
$now = (Get-Date)
$hash.Add($t, $now)

The Remove method has the following syntax:
Remove(Key)
$hash.Remove("Time")

Variables

```
#variables
$var1 = "Hello Jeetu"
$var1
$var1.GetType()

$var2 = 12345
$var2
$var2.GetType()

$var3 = 3.14
$var3
$var3.GetType()

$var1 = 189
$var1.GetType()

$var4 = Get-Date
$var4.GetType()

$var5 = Get-Process
$var5.GetType()


cls
$var6 = "Jeetu"
Write-Host "Hello Mr. $var6"

$var6 = "Singh"
Write-Host "Hello Mr. $var6"
```

Array

```
#method 1
$arr1 = 1,2,3,4,5,6,7,8,9,10
$arr1.GetType()
$arr1[0]
$arr1[6]
$arr1[1..5]
$arr1[-1]
$arr1[-4]
$arr1[-1..-5]

#method 2
$arr2 = @(1,2,3,4,5,6,7,8,9,10)
$arr2
$arr2.GetType()
$arr2[0]
$arr2[6]
$arr2[1..5]
$arr2[-1]
$arr2[-4]
$arr2[-1..-5]

#multi-dim array.
$arr3 = @(
@(1,0),
@("a","b"),
@(Get-Date)
)
$arr3.GetType()
$arr3
$arr3[0][0]
$arr3[1][0]
$arr3[2][0]
$arr3[1][1]
```

Hash Tables

```
#hash tables
<#
   - dictionary
   - key-value pais
#>

#syntax:
$ht1 = @{}        # for empyt hash table
$ht1.GetType()

#$ht2 = @{Name = "Jeetu" ; Techonology = "Powershell" ; Client = "LTI" }         #unordered
hashtable
$ht2 = [ordered]@{Name = "Jeetu" ; Technology = "Powershell" ; Client = "LTI" }  #ordered
hashtable

#print the hash table
$ht2

#print the hash table's Key & Values
$ht2.Keys
$ht2.Values
```

```
#print the elements within hash tables
$ht2.Technology
$ht2.Name

#print the hash table elements
Write-Host "Today is the day ONE for $($ht2.Technology) by $($ht2.Name) for the client
$($ht2.Client)"
$ht2.Count      #display the count

#adding email in hash tables
$ht2.Add("EmailID","jeetu.singh5591@hotmail.com")
$ht2

#removing the element from the HT
$ht2.Remove("Technology")
$ht2

#updating the hash table value.
$ht2["client"] = "LNT"
$ht2

#real time implementation
Get-Service -Name BITS | Select-Object Name, Status, @{ l = "FullName" ; e = {$_.Displayname} }
```

Conversion:

```
#conversion = converting the output in to required file format

Get-Service -Name BITS | Select-Object -Property * | Out-File bits.txt
Get-Service -Name BITS | Select-Object -Property * | ConvertTo-Csv | Out-File bits.csv
Get-Service -Name BITS | Select-Object -Property * | ConvertTo-Html| Out-File bits.html
Get-Service | Select-Object -Property * | ConvertTo-Html| Out-File services.html

Get-Process | `
Select-Object -Unique | `
Select-Object ProcessName, @{ l = "CPU" ; e = { [math]::Round(($_.CPU),2)} } | `
Sort-Object CPU -Descending | `
Select-Object -First 10 | `
ConvertTo-Html | Out-File process.html

Get-Service -Name BITS | ConvertTo-Json | Out-File bits.json

Get-Process | `
Select-Object -Unique | `
Select-Object ProcessName, @{ l = "CPU" ; e = { [math]::Round(($_.CPU),2)} } | `
Sort-Object CPU -Descending | `
Select-Object -First 10 | ` out-File process.html

out-File process.txt
```

3. Objects

15 May 2020 02:44 PM

Creating Objects	What does an Object Mean ? New-object
Using Properties	Adding Properties Read-Only and Read-Write Properties Types of Properties
Using Methods	Get_ and Set_ Methods Standard Methods Method Calls and Arguments Method Signatures
Using Real Examples	Use some applications of objects and explain

An object is simply the programmatic representation of anything.

Objects are usually considered as two types of things:
- Properties, which simply describes the attributes of .NET object.
- Methods, which describe the types of actions, that the .NET object can undertake.

Example to understand:
- Let us consider a car as an example. If we were making a car into a .NET object, then its
  - **Properties** would include its engine, doors, accelerator and brake pedals, steering wheel and headlights and Its
  - **Methods** would include turn engine on, turn engine off, open doors, close doors, press accelerator, release accelerator.

```
#to get the list of files & directories that was accessed after May 10th, 2020.
Get-ChildItem -Path C:\Users\Jeetu\Downloads\ | Where-Object {$_.LastWriteTime -gt "05/10/2020"} | `
select Name

#to list the objects & members.
Get-Member

#The $Pshome automatic variable contains the path of the PowerShell installation directory.
Get-ChildItem $pshome\PowerShell.exe | Get-Member

#To get only the properties of an object and not the methods
Get-ChildItem $pshome\PowerShell.exe | Get-Member -MemberType property

#Command displays the values of all the properties of the PowerShell.exe file.
Get-ChildItem $pshome\PowerShell.exe | Format-List -Property *
```

PowerShell Basics: Custom Objects
- One type of .NET object you can build is the custom object.
- A custom object lets you define the types of data assigned to that object and the actions it can carry out.

Creating a Custom Object
- To create a custom object in PowerShell, you must first use the New-Object cmdlet to build the initial object. This cmdlet lets you create .NET or COM objects from an assortment of classes.
- A custom object is a special type of .NET object based on either the Object or PSObject .NET class.

```
$system = New-Object -TypeName PSObject
$system.GetType()
```

About methods
- PowerShell uses objects to represent the items in data stores or the state of the computer.
- Objects have properties, which store data about the object, and methods that let you change the object.
- A "method" is a set of instructions that specify an action you can perform on the object.
- Details --> *help about_Methods*
- To get the methods of any object,
  - Get-Process | Get-Member -MemberType Method

Basic examples (working):

```
#objects
Get-Process | Select-Object ProcessName,CPU

Get-Process | Get-Member
Get-Date    | Get-Member
Get-Service | Get-Member
Get-Service | gm          #gm is alias for get-member command

Get-Process | Select-Object ProcessName,CPU
Get-Process | Sort-Object ProcessName -Unique | Select-Object ProcessName,CPU

Test-Connection google.com | Select-Object IPV4Address,Destination
Test-Connection google.com | Get-Member
Test-Connection google.com | Select-Object IPV4Address,Address

#create custom object(s)
$obj1 = New-Object -TypeName psobject
$obj1.GetType()
$obj1 | Get-Member

#adding a SINGLE member to the object
Add-Member -InputObject $obj1 -MemberType NoteProperty -Name "demoKey" -Value "demoValue"
$obj1 | Get-Member
Add-Member -InputObject $obj1 -MemberType NoteProperty -Name "demoKey1" -Value "demoValue1"
$obj1 | Get-Member

#adding MULTIPLE members to the object.
$newmembers = @{
prop1 = "val1"
prop2 = "val2"
prop3 = "val3"
prop4 = "val4"
mykey1 = "myValue1"
}

$obj2 = New-Object -TypeName psobject -Property $newmembers
$obj2 | Get-Member
```

[use this... It's working]

Creating Custom object:

- You can use the New-Object cmdlet to generate an object of any type.
- The two choices for custom objects are PSObject and Object:
  - **PSObject** [widely used] creates an object of class System.Management.Automation.PSCustomObject
  - **Object** creates an object of class System.Object

The New-Object cmdlet creates an instance of a .NET Framework

Creating an object:

```
$obj = new-object psobject
```

Adding new member to the objects:

```
Add-Member -InputObject $obj -MemberType NoteProperty -Name customproperty -Value ""

$obj = New-Object psobject
$obj.GetType()          #get type
$obj | gm               #get members
#to add one member at a time
Add-Member -InputObject $obj -MemberType NoteProperty -Name "mYname" -Value "mYvalue"
Add-Member -InputObject $obj -MemberType NoteProperty -Name "mY1name" -Value "mY1value"
$obj | gm               #get members again to verify
```

To add multiple members at a time:

```
#creating new properties
$newobj = @{
Prop1 = "val1"
```

```
Prop2 = "val2"
Prop3 = "val3"
Prop4 = "val4"
}

#creating objects
$oj = New-Object psobject -Property $newobj

#fetching all members
$oj | gm

$oj.Prop1        #listing val from property
$oj | fl         #listing all values in a formatted way
$oj | Select-Object Prop1, Prop4       #listing selective objects
```

Another way of creating new custom object & adding values to it:

```
$a = [pscustomobject]@{
firstname = 'jeetu'
lastname = 'Singh'
}
$a | Select-Object firstname
$a.firstname
$a | Add-Member -MemberType NoteProperty -Name "location" -Value "india"
$a
```

```
#writeing data on console
Write-Output ‘Hello, World’

#storing data in variables
$string = "Hello, World"

#listing members
$string | Get-Member

#printing length
$string.Length

#converting it to upper case
$string.ToUpper()

#converting it to lower case
$string.ToLower()

#replacing "Hello" with "goodbye"
$string.Replace('Hello','goodbye')      #case sensitive

#Create HashTable
$hashtable = @{ Color = ‘Red’; Transmission = ‘Automatic’; Convertible = $false}

#Creating a PowerShell Custom Object
[pscustomobject]$hashtable
```

4. Conditional Operators

15 May 2020 02:44 PM

about_Comparison_Operators

Conditional	Conditional Operators	Overview on conditional operators used Logical Operators and Use
	Where-object	Use of where-object with example
	If-ElseIf-Else	How to use in Powershell
	Switch	Syntax in powershell

- Arithmetic Operators
  - + (Addition)
  - - (Subtraction)
  - * (Multiplication)
  - / (Division)
  - % (Modulus)

- Assignment Operators
  - =
  - +=
  - -=

- Comparison Operators
  - eq (equals)
  - ne (not equals)
  - gt (greater than)
  - ge (greater than or equals to)
  - lt (less than)
  - le (less than or equals to)

- Logical Operators
  - AND (logical and)
  - OR (logical or)
  - NOT (logical not)

- Redirectional Operators
  - > (Redirectional Opeator)

Conditions:
- if statement
- if...else statement
- nested if statement
- switch statement

IF Statement

```
if(Boolean_expression) {
  // Statements will execute if the Boolean expression is true
}
```

If Else Statement

```
if(Boolean_expression) {
  // Executes when the Boolean expression is true
}else {
  // Executes when the Boolean expression is false
}
```

Nested If Else Statement

```
$x = 30
$y = 10

if($x -eq 30){
  if($y -eq 10) {
    write-host("X = 30 and Y = 10")
  }
}
```

Switch Statement

```
switch(3){
  1 {"One"}
  2 {"Two"}
  3 {"Three"}
  4 {"Four"}
  3 {"Three Again"}
}
```

5. Loops

15 May 2020 02:44 PM

Loops	ForEach-Object	Looping with all conditions
	Foreach	
	Do and While	Continuation and Abort Conditions
	For	Nested Loops
	switch	How to use switch for looping
	continue statements	Use of the continue statement Labels

Loops:

- For Loop
- ForEach Loop
- While Loop
- Do..While Loop

For loop:

Ex1:-

```
$array = @("item1", "item2", "item3")
for($i = 0; $i -lt $array.length; $i++){ $array[$i] }
item1
item2
Item3
```

Ex2:-

```
#for loop

$arr3 = @(1,2,3,4,5)
cls
for ( $i = 0; $i -lt $arr3.Length; $i++ ){
   Write-Host "Index Value $i is" $arr3[$i]
}
```

ForEach Loop

Ex1:-

```
> $array = @("item1", "item2", "item3")

> foreach ($element in $array) { $element }
        item1
        item2
        item3

> $array | foreach { $_ }
        item1
```

```
> $array | foreach { $_ }
        item1
        item2
        item3
```

```
Ex2:-

$arr4 = @(1,2,3,4,5)
cls

Write-Host "method 1"
foreach ( $a in $arr4 ){
   Write-Host $a
}
""
Write-Host "method 2"
$arr4 | foreach{
   Write-Host $_
}
```

While Loop

```
#loops - EX1

$arr1 = @("a","b","c")
$c = 0
cls
while ( $c -lt $arr1.Length ){
   $arr1[$c]
   $c += 1
   sleep 1
}
$c = $null
$arr1 = $null
```

```
#loops - EX2

> $array = @("item1", "item2", "item3")
$counter = 0;

while($counter -lt $array.length){
  $array[$counter]
  $counter += 1
}

        item1
        item2
        item3
```

Do..While Loop

```
Ex1:

$array = @("item1", "item2", "item3")
$counter = 0;
```

```
do {
  $array[$counter]
  $counter += 1
} while($counter -lt $array.length)

        item1
        item2
        item3
```

Ex2:

```
#do-while
$arr2 = @(1,2,3,4,5,6,7,8,9,10)
$c = 0
cls
do{
   $arr2[$c]
   $c +=1
}while( $c -lt $arr2.Length )
```

6. Functions and Pipelines

15 May 2020 02:44 PM

Functions and Pipelines	Creation and Calling	How to create a function How to call a function Defining parameters
	Using Pipelines	Explain some of the usage of pipelines with examples

To see how easy it was to send the object from one cmdlet to another with no effort at all.

- Trace-Command -Name ParameterBinding -Expression {get-process notepad| Stop-Process -WhatIf } -PSHost

Powershell functions

Ex1:

```
function Do-Something {
   Write-Host 'Function Called'
}

PS > Do-Something
Function Called
```

Ex2:

```
function Do-Something {
   param( $String )
  Write-Host "Function Called using Param"}

PS > Do-Something -String 'thing'
Function Called using Param
```

```
#EX1:-

#declare the function
function get-message{
   Write-Host "Hello World. This is my first function"
   sleep 1
   get-message

}

cls
#call the funtion
get-message


#EX2:-
function fun{
Write-Output "function called"
}
Fun
```

```
function add{
   param(
   [int]$a,
   [int]$b
   )
   Write-Host "Total of $a & $b is "($a + $b)
}

add -a 10 -b 20
```

Advanced functions:

```
function adv-fun{
[CmdletBinding()]
   param(
   [Parameter(Mandatory)]$String
   )

  Write-Output "Joker"
}
adv-fun -String haha
```

Backbone of a Function

```
Function Test-ScriptBlock
{
   Param
   (
      [int]$Number
   )
   BEGIN
   {
      Write-Host "In Begin block"
   }

   PROCESS
   {
      Write-Host "In Process block"
   }
   END
   {
      Write-Host "In End block"
   }
}
Test-ScriptBlock
```

In order to turn our function into an advanced function

```
[CmdletBinding()]
Param
(
   [Parameter(ValueFromPipeline)]
   [int]$Number
```

```
)
```

```
#basic function
function fun-name ($msg){
Write-Host "Write Host data : " $msg
Write-Verbose "Write Verbose data"
}
cls
fun-name "Hi Jeetu" -verbose

#advance function
function smvrb-smnoun {
[CmdletBinding()]
param(
   [string]$str = "some string value"
)
Write-Verbose $str

}
cls
smvrb-smnoun -Verbose
```

```
#basic function using PSCUSTOM objects

function get-dirsize{
param (
[Parameter(Mandatory=$true)]
[string]$dirname
)

if(Test-Path -Path $dirname){
   Get-ChildItem -Path $dirname -File | ForEach-Object {$size += $_.Length}
   [PSCustomObject]@{'Directory' = $dirname; 'SizeInMB' = $size / 1MB}
}
else{
   Write-Error "Cannot find the directory"
}

}
cls
get-dirsize -dirname C:\Windows\System32\WindowsPowerShell\v1.0\en-US
```

```
#Advance function using PSCUSTOM objects + get values from pipeline

function get-dirsize1{
param (
[Parameter(Mandatory=$true, ValueFromPipeline = $true)]
[string]$dirname
)

if(Test-Path -Path $dirname){
   Get-ChildItem -Path $dirname -File | ForEach-Object {$size += $_.Length}
   #[PSCustomObject]@{'Directory' = $dirname; 'SizeInMB' = $size / 1GB}
   $s = [math]::Round(($size / 1GB),2)
   [PSCustomObject]@{'Directory' = $dirname; 'SizeInGB' = $s}
```

```
}
else{
   Write-Error "Cannot find the directory"
}

}
cls
$dir = "C:\Users\Jeetu\Downloads\Justice.League.Dark.Apokolips.War"
Get-Item $dir | get-dirsize1
```

Basic functions

Get-cube

```
function Get-cube([int]$x)
{
  $result = $x * $x * $x
  return $result
}

$x = Read-Host 'Enter a value'
$result = Get-cube $x
Write-Output "$x * $x * $x = $result"
```

PowerShell parameter attributes

- Mandatory parameters
- Parameter validation

Creating function using **Mandate** parameter.

```
function mandate{
[cmdletbinding()]
   param(
      [Parameter(Mandatory)]
      [string]$a
   )
cls
Write-Host "You have sent '$a' using mandate parameters"

}

mandate -a haha
```

Creating function using **validation** parameter

```
function validation{
[cmdletbinding()]
   param(
      [Parameter(Mandatory)]
      [ValidateSet('joker','haha')]
      [string]$a
   )
cls
```

```
Write-Host "You have sent '$a' using validation parameters"

}

validation -a joker     #working
validation -a haha      #working
validation -a jeetu     #error
```

Passing the values using pipeline:

```
function mandate{
[cmdletbinding()]
   param(
      [Parameter(Mandatory=$true,
      ValueFromPipeline=$true
      )]
      [string]$a
   )
Write-Host "You have sent '$a' using mandate parameters"
}
#notepad.exe test.txt
$file = gc .\test.txt
$file | mandate
```

But, it won't work as expected. As it will give only last value from the file.

To get it right:

```
function mandate{
[cmdletbinding()]
   param(
      [Parameter(Mandatory=$true,
      ValueFromPipeLine=$true
      )]
      [string]$a
   )

Write-Host "You have sent '$a' using mandate parameters"
}
cls
#notepad.exe test.txt
$file = Get-Content .\test.txt

$file | ForEach { mandate -a $_}
```

Ping function:

```
function get-ping{
param([string]$ws)

if(Test-Connection $ws -Count 1 -EA SilentlyContinue){
   Write-Host "$ws is pingable" -ForegroundColor Green
}else{
   Write-Host "$ws is NOT pingable" -ForegroundColor Red
}

}
```

```
cls
get-ping -ws google.com
get-ping -ws microsoft.com
get-ping -ws lti.com
```

Ping function:

```
function get-ping{
param([string]$ws)
   if(Test-Connection $ws -Count 1 -EA SilentlyContinue){
      Write-Host "$ws is pingable" -ForegroundColor Green
   }else{
      Write-Host "$ws is NOT pingable" -ForegroundColor Red
   }
}
cls
$sites = @("google.com","microsoft.com","lti.com")
$sites | foreach{
   Get-ping $_
}
```

7. Script Execution

15 May 2020 02:44 PM

Creating and Launching	**How to create a script file** **Launching the Script**
Executing Policy	Settings for execution policy
Arguments handling	Scopes - Variables Profile Scripts
ConsoleFiles	Use of console files
Digital Signatures	How to sign a script

#execution policy
<#
- it will allow or deny the exec. of a script.
-

#>

#binding the script with the certificate.
Get-Help Set-AuthenticodeSignature -Examples

$cert=Get-ChildItem Cert:\CurrentUser\my\2A8CD499223B213CF209E09335CD34178055FEF0 -CodeSigningCert

Set-AuthenticodeSignature -FilePath C:\LTI-ps\lti-16Dec-21-Dec\day2\getdate.ps1 -Certificate $cert

Set-AuthenticodeSignature -FilePath C:\LTI-ps\lti-16Dec-21-Dec\day2\dummy.ps1 -Certificate $cert

Get-AuthenticodeSignature -FilePath C:\LTI-ps\lti-16Dec-21-Dec\day2\getdate.ps1

8. Error Handling

15 May 2020 02:46 PM

Error Handling	Suppressing Errors	How to suppress the error from not being thrown
	Try / Catch	How to use try / Catch statements
	Using traps	Use of trap statements
	Exceptions	Understand Exceptions Handle Exceptions Throwing our own Exceptions
	Stepping and Tracing	How to do debug with step and trace

Help about_try_catch_finally

Use Try, Catch, and Finally blocks to respond to or handle terminating errors in scripts.

Terminating and Non-Terminating Errors

- A terminating error is an error that will halt a function or operation.
- Non-terminating errors allow Powershell to continue and usually come from cmdlets or other managed situations.

ERRORS IN POWERSHELL

```
> $error                          #shows all errors
> $error.count                    #count all errors
> $error[0]                       #listing 1st error
> $error[0].InvocationInfo        #listing WHO raised the command
> $error[0].InvocationInfo.Line   #listing WHO raised the command, with line (cmd)
> $error[0].Exception             #The exception that raised the error can be accessed.
> $error[0].Exception.Message     #exception's message in string format.
```

$ERRORACTIONPREFERENCE

Stop
- Display error, and stop execution.

Inquire
- Display error, and ask to continue.

Continue (Default)
- This is the default setting. Display error, then continue execution.

Suspend
- This one is for workflows. A workflow job is suspended to investigate what happened, then the workflow can be resumed.

SilentlyContinue
- No error is displayed, execution is continued.

EXAMPLE:

```
Get-ChildItem -Path 'C:\Windows\appcompat' -Recurse;Write-Host 'Test'
```

now with $ErrorActionPreference

```
$ErrorActionPreference = 'Stop'
Get-ChildItem -Path 'C:\Windows\appcompat' -Recurse;Write-Host 'Test'
```

ERROR HANDLING

basic method - 1:

```
if (Get-ChildItem Z:\ -ErrorAction SilentlyContinue) {
   Write-Host 'I can list the contents of Z:!'
} else {
   Write-Host 'I cannot list the contents of Z:!'
}
```

basic method - 2:

```
$z = Get-ChildItem Z:\ -ErrorAction SilentlyContinue
cls
if ($z) {
   Write-Host 'I can list the contents of Z:!'
} else {
   Write-Host 'I cannot list the contents of Z:!'
}
```

TRY/CATCH/FINALLY

- The Try, Catch, and Finally blocks in PowerShell allow us to capture terminating errors.
- The **Try** block contains the code you'd like to execute, and catch any potential errors that happen.
- The **Catch** block contains the code you'd like to execute after a terminating error has occurred. The current error will e accessible via the automatic variable $_.
- The **Finally** block contains the code you'd like to run after the event has occurred. This is good for clean-up tasks.
- It is worth noting that finally block is not required.

Example1:

```
try {
   someRandomCmd
}
catch {
   Write-Host "An error occurred:"
   Write-Host $_
}
```

Example2: using try, catch, finally

```
Try {
Write-Output "IN TRY BLOCK"`n
$command = 'get-FakeCommand'
Write-Host "Attempting to run: [Invoke-Expression -Command $command]"`n
Invoke-Expression -Command $command
}

Catch {
Write-Output "IN Catch BLOCK"`n
Write-Host $_.Exception.Message`n
}

Finally {
Write-Output "IN Finally BLOCK"`n
Write-Host "Clean up: `$commmand = `$null"
$commmand = $null
}
```

Another example: divide by zero

```
cls
$one=1
$zero=0

try{
 $one/$zero
}

catch [System.DivideByZeroException]{
"divide by zero error"
}

catch {
"something else went wrong!!!"
}

finally {
$one=$null
$zero=$null
"All variables cleaned"
}

write-host "1st vari = $one"
write-host "2nd vari = $zero"
```

Debugging using write-debug:

```
function get-cube {
[CmdletBinding()]
param(
$a
)
Write-Debug "received `$a value as $a"
$ans = $a * $a * $a
Write-Debug "Cube calculated & stored in `$ans variable"
Write-Debug "Printing Cube calculated value"
return $ans
}
cls
get-cube 5
```

Using verbose as debugging tool:

```
function get-cube {
[CmdletBinding()]
param(
$a
)
Write-verbose "received `$a value as $a"
$ans = $a * $a * $a
Write-verbose "Cube calculated & stored in `$ans variable"
Write-verbose "Printing Cube calculated value"
return $ans
}
cls
get-cube 5 -Verbose
```

Traps

- The Trap statement includes a list of statements to run when a terminating error occurs.
- By default, this will trap any terminating error or optionally you may specify an error type.
- A script or command can have multiple Trap statements.
- Trap statements can appear anywhere in the script or command.

#general syntax:

- trap [[<error type>]] {<statement list>}
  OR
- trap {<statement list>}

#this trap gives an error, as it needs "CONTINUE"

```
Trap { 'Something terrible happened!' }
1/0
```

#below trap works perfectly, when an error occurs

```
trap { 'Something terrible happened!'; continue }
1/$null
```

WORKING EXAMPLES:

```
$ErrorActionPreference = "silentlycontinue"
function Get-Eth1 {
  Trap {Write-Output "There is a terminating error."}
  Get-NetAdapters
}
cls
#Get-Eth1

#####################################################################

function Get-Eth2 {
  Trap {Write-Output "There is a terminating error: $_"}

  Get-NetAdapters
}
cls
#Get-Eth2

#####################################################################

function Get-Eth3 {
  Trap {Write-Output "Uknown terminating error."}
  Trap [System.Management.Automation.CommandNotFoundException]{
    Write-Output "There is a terminating error: $_"}

  Get-NetAdapters
}
cls
#Get-Eth3

#####################################################################

function Get-Eth4 {
  Trap {Write-Output "Uknown terminating error."}
```

```
   Trap [System.Management.Automation.CommandNotFoundException]{
      Write-Output "There is a terminating error: $_"}

   Get-NetAdapters

   Write-Output "This is the END of the FUNCTION"
}

Write-Output "Script started processing..."
Write-Output "Processing..."
Write-Output "More processing..."

Get-Eth

Write-Output "This is the END of the SCRIPT"
cls
#Get-Eth4

#####################################################################

Function Do-Something {
  Trap {
    Write-Host 'Error in function' -fore white -back red
    Continue
  }
  Write-Host 'Trying' -fore white -back black
  gwmi Win32_BIOS -comp localhost,not-here -ea stop
  Write-Host 'Tried' -fore white -back black
}
cls
Write-Host 'Starting' -fore white -back green
Do-Something
Write-Host 'Ending' -fore white -back green
```

```
#example without ANY ISSUE:
        trap {
        Write-Host “you are inside trap now”
        continue
        }

        function one{
        Write-Host "in function one"
        }

        function two {
        Write-Host "in function two"
        }
        cls

        one
        Write-Host "in middle"
        two
```

```
#example - using function with traps
        trap {
        Write-Host “inside script trap”
        continue
        }
        function A {
        trap {
        Write-Host “inside function trap”
        continue
        }
        write-host “I am inside function A”
        gwmi win32_service -ComputerName notavailable -ea stop
        Write-Host ” this message is after error ”
        }
        A
        write-host “w are done with function A. Moving to function B”
        B
        function B{
        Write-host “I am inside function B”
        }
```

Debugging

Types of errors:

1. Syntax errors
2. Logical errors

To start debugging, follow below steps:

1. Create a script
2. [Most IMP], save the script to debug.

To toggle break point on any script.

- Top menu -> Debug -> Toggle breakpoint (F9) on the start line.
- Run the script.
- Hit F11 to proceed further.

URL: https://devblogs.microsoft.com/scripting/use-the-powershell-debugger/

9. Input / Output

15 May 2020 02:47 PM

Input / Output	Command Line Arguments	Using Command line arguments in Powershell (param)
	Console Read / Write	
	File Read / Write	

notepad.exe hah.txt
$file = gc .\hah.txt
ls | Set-Content .\hah.txt

10. Text Processing and Regular Expressions

18 May 2020 01:36 PM

Text Processing and Regular Expressions	**String Formatting**	**Defining the String** **How do we format strings in powershell**
	String Operators	Operators used to process the String
	String Object Methods	Methods availabled which can be used on String Objects
	Regular Expression Pattern and Matching	Defining the patterns How to match the patterns
	Regular Expression activities	Keywords Segmenting Replace Symbols to be known

Help: about_regular_expressions

```
# This statement returns true because book contains the string "oo"
'book' -match 'oo'

# This expression returns true if the pattern matches big, bog, or bug.
'big' -match 'b[iou]g'

# This expression returns true if the pattern matches any 2 digit number.
42 -match '[0-9][0-9]'  #true
42 -match '[5-9][0-1]'  #false

#searching a keyword in a line
$message = 'there is an error with your file'
$message -match 'error'
$message -like '*error*'

#split
'haha,Joker,is,back' -split ','

#-replace
$message = "Hi, my name is Jitendra"
$message -replace 'Jitendra','Jeetu'

#replace more:
#Match only if at the beginning of the line: ^
'no I am not MAD' -replace '^no',''

#Match only if at the end of the line: $
'There must be some way out of here said the joker to the joker' -replace 'joker$','thief'

'Why so Serious, Said by joker' -replace 'joker','heath ledger'

“Smile, because it confuses people. Smile, because it's easier than explaining what is killing you inside.” -replace "^smile","Dont CRY"
```

```
#notepad.exe C:\Users\Jeetu\Desktop\ps-wiki.txt
#gsv | Select-Object DisplayName | sort DisplayName | Out-File C:\Users\Jeetu\Desktop\ps-wiki.txt

#getting the data in a variable
$text = gc C:\Users\Jeetu\Desktop\ps-wiki.txt

#searching & counting the keywords
$text | Select-String -Pattern 'Windows Update' | measure
$text | Select-String -Pattern 'Hyper-V' | measure
$text | Select-String -Pattern 'Cellular Time' | measure

#replacing text with new & storing it in a new variable
$new = $text -replace "Cellular Time","time kharab hai re"

#verify old value
$new | Select-String -Pattern 'Cellular Time' | measure

#verify new value
$new | Select-String -Pattern 'time kharab hai re' | measure


https://powershellexplained.com/2017-07-31-Powershell-regex-regular-expression/
```

```
<#
#Regex quick start
\d digit [0-9]
\w alpha numeric [a-zA-Z0-9_]
\s whitespace character
.  any character except newline
() sub-expression
\  escape the next character

URL: https://powershellexplained.com/2017-07-31-Powershell-regex-regular-expression/

#>
```

11. Configuration using XML

18 May 2020 11:38 PM

Configuration using XML	XML Parsing	How to read the nodes and Data structure used Using it as config file

#data URL: https://docs.microsoft.com/en-us/previous-versions/windows/desktop/ms762271(v%3Dvs.85)

```
notepad.exe C:\Users\Jeetu\Desktop\dummy.xml
        [xml]$xdata = gc C:\Users\Jeetu\Desktop\dummy.xml

#get variable type
        $xdata.GetType()

#listing authors name
        $xdata.catalog.book | select author
        $xdata.catalog.book | select author,title
        $xdata.catalog.book | select author,title, Price
        $xdata.catalog.book | select author,title, Price, publish_date

#fetching XML data in details
        $xdata.catalog.book

#fetching XML data in details - well-formated
        $xdata.catalog.book | Format-Table -AutoSize

#reading element-by-element
        $xdata.catalog.book[0] | Format-Table -AutoSize
        $xdata.catalog.book[1] | Format-Table -AutoSize
        $xdata.catalog.book[2] | Format-Table -AutoSize
        $xdata.catalog.book[3] | Format-Table -AutoSize

#searching using "ID"
        $xdata.catalog.book | Where-Object {$_.id -match "bk106"}
```

Audio Recording

```
#searching using "ID" + well-formated
        $xdata.catalog.book | Where-Object {$_.id -match "bk106"} | ft -AutoSize

#searching using "ID" + well-formated + full text display
        $xdata.catalog.book | Where-Object {$_.id -match "bk106"} | ft -AutoSize -Wrap
```

12. Windows Registry

18 May 2020 11:39 PM

```
#Getting Registry Key Values Locally with PowerShell
        get-psdrive
        Get-PSDrive -PSProvider Registry

#navigate to HKLM content
        cd HKLM:\

#or

        set-location -path HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\

#listing
        Get-childitem
        Get-childitem  | measure

#listing properties
        Get-Item -path HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\wsman

#listing HKCU
        cd hkcu:\

#Creating a Registry Key with PowerShell
        New-Item –Path "HKCU:\demo" –Name JeetuRegKey -Force

#creating new property for the above reg key
        New-ItemProperty -Path "HKCU:\demo\JeetuRegKey" -Name "demoKey" -Value "demoValue"  -PropertyType "String"

#to edit the registry key value
        Set-ItemProperty -Path "HKCU:\demo\JeetuRegKey" -Name "demoKey" -Value "0"

#delete the “JeetuRegKey” parameter
        Remove-ItemProperty -Path "HKCU:\demo\JeetuRegKey" -Name "demoKey"

#remove the key “JeetuRegKey” itself
        Remove-Item -Path "HKCU:\demo\JeetuRegKey" -Recurse
```

13. Processes, Services and Event Log Management

19 May 2020 12:09 AM

Processes, Services and Event Log Management	Process Management	Reading the process list Getting Process Info Calling Process Methods (Kill)

Process:

```
Get-Process

Get-Process -FileVersionInfo

Get-Process -FileVersionInfo -ErrorAction SilentlyContinue

Get-Process | Select-Object *

#listing top 10 process
Get-Process | select -Unique | Select-Object ProcessName, CPU | Sort-Object CPU -Descending | select -First 10

#listing top 10 process with 2 digits
Get-Process `
| select -Unique `
| Select-Object ProcessName, @{L="CPU";E={[math]::Round(($_.cpu),2)}} `
| Sort-Object CPU -Descending `
| select -First 10

Get-WmiObject Win32_Process | select name -Unique

Start-Process notepad -Wait

Stop-Process -Name Notepad

Get-Process Notepad -ea 0 | ForEach-Object { $_.CloseMainWindow() }
```

Service:

```
Get-Service

Get-Service Spooler

Stop-Service Spooler

Stop-Service Spooler -Force

Get-WmiObject Win32_Service | select Name
```

Logs

```
#list all event categories
```

```
	Get-EventLog -List

#get system logs
	Get-EventLog -LogName System

#get top 5 event entries
	Get-EventLog -LogName System -EntryType Error -Newest 5

#get top 5 event entries + wrap
	Get-EventLog -LogName System -EntryType Error -Newest 5 | ft -AutoSize -Wrap

#get top 5 event entries in detailed
	Get-EventLog -LogName System -EntryType Error, Warning -Message *Time* -Newest 5 |
	Select-Object TimeWritten, Message

#create a new log
	New-EventLog -LogName Application -Source PowerShellScript

#write data in log "ADMIN MODE REQUIRED"
	Write-EventLog -LogName Application -Source PowerShellScript -EntryType Information -
	EventId 123 -Message 'This is my first own event log entry'

#display the log
	Get-EventLog -LogName Application -Source PowerShellScript

#open event viewer
	Show-EventLog

#remove custom generated log entry,
	Remove-EventLog -Source PowerShellScript
```

14.WMI Management

20 May 2020 08:19 AM

WMI Management	Using WMI Objects	Exploring the Classes Reading the Properties Filters
	Using WMI Methods and properties	Changing the properties Calling the methods

```
#WMI objects
<#
   - WIndows mgmt instrumentation
   - implementation of CIM
      CIM - Common Information Model
         - gets the inform of
            - h/w
            - s/w
            - firmware
            - service
            - process
            - local / remote machine
         - Opensource
   - WMI is MS implementaiton of CIM.

#>

#command
Get-WmiObject   #or
gwmi          #alias of get-wmiobject

Get-WmiObject -List
Get-WmiObject -List | measure
Get-WmiObject -List | Where-Object {$_.name -match "^Win32_"}
Get-WmiObject -List | Where-Object {$_.name -match "^Win32_"} | measure
Get-WmiObject -Class win32_bios
Get-WmiObject -Class win32_operatingsystem
Get-WmiObject -Class win32_computersystem
Get-WmiObject -Class win32_logicaldisk
Get-WmiObject -Class win32_PhysicalMemory
Get-WmiObject -Class win32_battery
Get-WmiObject -Class win32_networkadapter
Get-WmiObject -Class win32_cdrom
Get-WmiObject -Class win32_process
Get-WmiObject -Class win32_service

Get-WmiObject -Class win32_bios -ComputerName 127.0.0.1
Get-WmiObject -Class win32_bios -ComputerName 127.0.0.1, localhost, 192.168.1.137
Get-WmiObject -Class win32_bios -ComputerName 127.0.0.1, localhost, 192.168.1.137 | Format-
Table
```

Get disk usage:

```
cls
$comp1 = Get-Content C:\day4\servers.txt

function get-diskusage{
   Write-Host "Total disk information" -ForegroundColor Black -BackgroundColor Yellow

   Get-WmiObject -Class win32_logicaldisk -ComputerName $comp1 | `
   Where-Object DeviceID -EQ "c:" | `
   Select-Object PSComputerName, `
   @{ l = "FreeSpace(GBs)" ; e = { [math]::Round(($_.FreeSpace/1GB),2) } }, `
   @{ l = "TotalSize(GBs)" ; e = { [math]::Round(($_.Size/1GB),2) } }
}

get-diskusage
```

15. Remote Execution

20 May 2020 08:20 AM

Remote Execution	Using Remote Execution	How to execute commands remotely? Methods for remote execution
	Using Invoke	Invoke-Session Invoke-Command Passing credential & credssp
	Remote WMI	Connecting to Remote system WMI Provider Passing credentials for authentication

For remote computers with PowerShell, we have three options.

- You can open an interactive session with the Enter-PSSession cmdlet (One-to-One Remoting).
- The Invoke-Command cmdlet, which allows you to run remote commands on multiple computers (which is why it is called One-to-Many Remoting).
- The third option is to use one of those cmdlets that offer a ComputerName parameter.

Ex:

```
Invoke-Command -Computername $RemoteComputer -ScriptBlock { Get-ChildItem "C:
\Program Files" }
Invoke-Command -ComputerName PC1,PC2,PC3 -FilePath C:\myFolder\myScript.ps1
Invoke-Command -ComputerName . -Scriptblock {Get-Process}
```

Testing if Remoting is enabled

```
If (Test-Connection -ComputerName $RemoteComputers -Quiet)
{
    Invoke-Command -ComputerName $RemoteComputers -ScriptBlock {Get-ChildItem “C:
\Program Files”}
}
```

The following command kills Notepad on the remote computer:

```
Invoke-Command -ComputerName $RemoteComputer -ScriptBlock {(Get-Process | Where -
Property ProcessName -eq notepad).Kill()}
```

Enabling PowerShell remoting in a domain

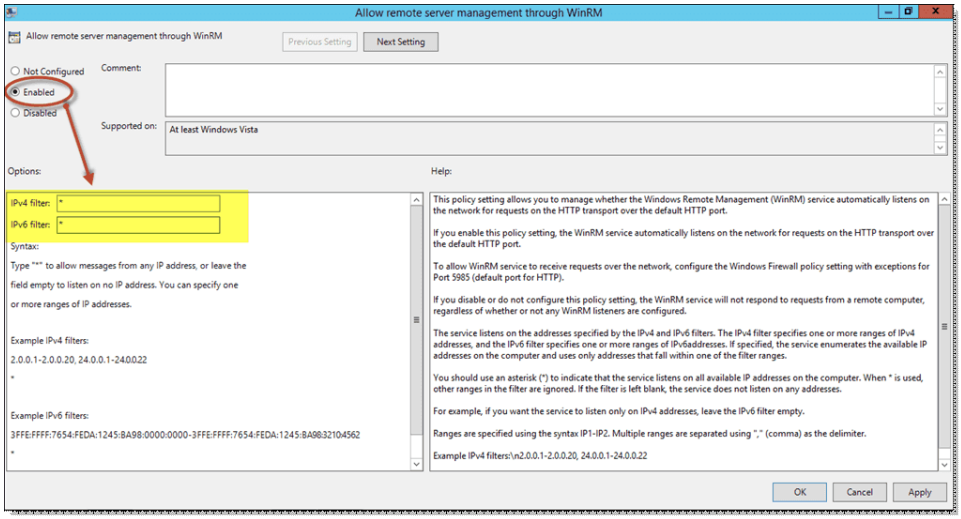
```
Enable-PSRemoting –Force
```

listing all remoting commands

```
Get-Command *remoting*
```

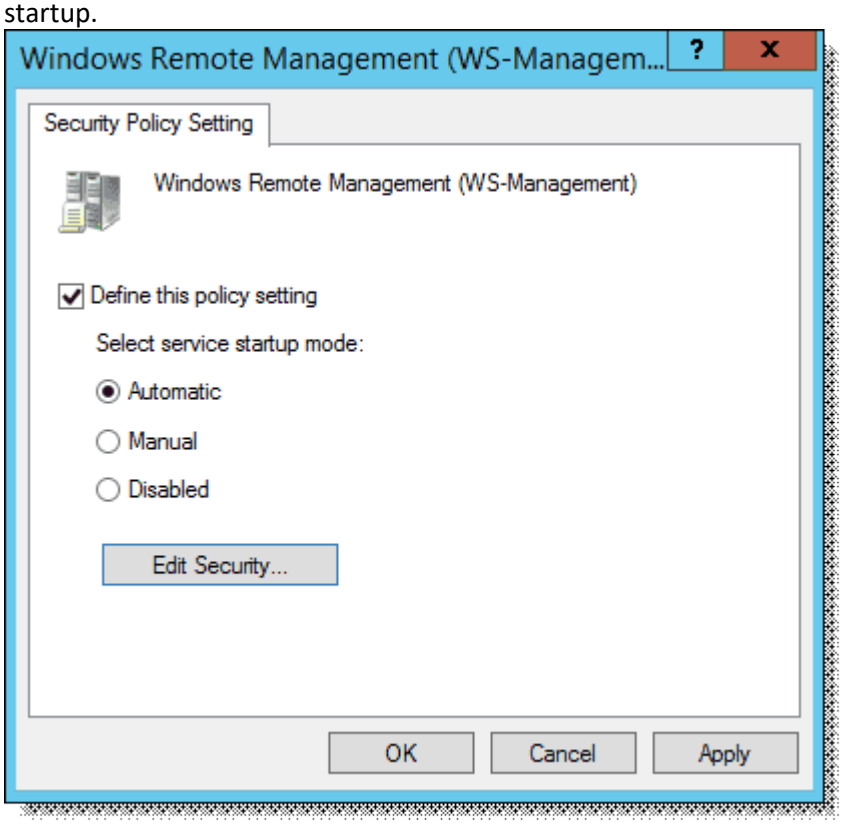
Remote server management through WinRM:

```
Computer Configuration\Policies\Administrative Templates\Windows Components\Windows
Remote Management (WinRM)\WinRM Service
```



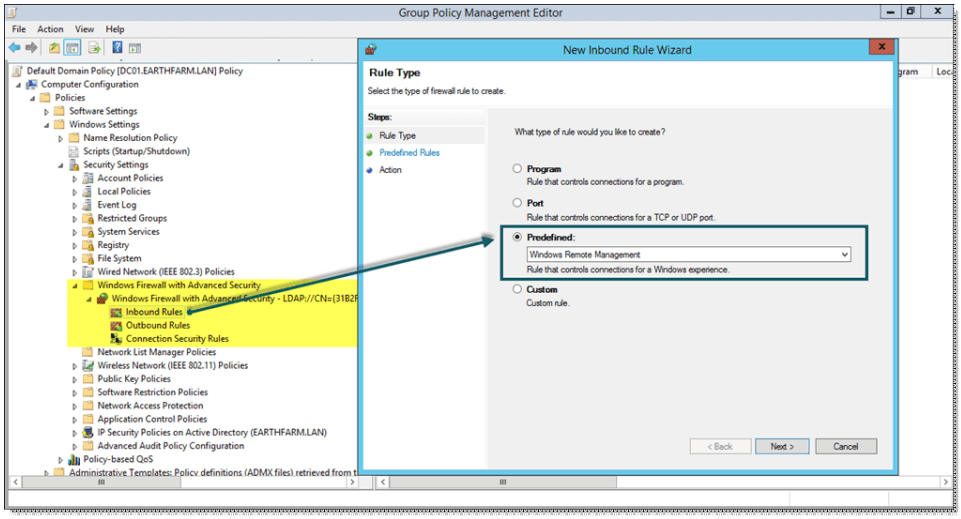
Set WinRM service to automatic startup:
Computer Configuration\Policies\Windows Settings\Security Settings\System Services

Select the Windows Remote Management (WS-Management) service and set it for automatic startup.



Configure Windows Firewall

Computer Configuration\Policies\Windows Settings\Security Settings\Windows Firewall with Advanced Security\Windows Firewall with Advanced Security



Testing Remote Access:

```
Enter-PSSession –Computername <hostname>
```

run the following command on both client and servers machines

```
enable-psremoting -force
```

Once you have done this, you can then use the client machine to send commands to be run on other machines, using invoke-command:

```
Invoke-Command -ComputerName RemoteServerName -ScriptBlock {
  get-childitem
  get-service
}
```

```
# Set up credentials for non-interactive connection.
$password =  ConvertTo-SecureString -String $env:Password -AsPlainText -Force
$cred = New-Object System.Management.Automation.PSCredential $env:Username,
$password
```

After that, you can then use your credential like this:

```
Invoke-Command -ComputerName RemoteServerName -credential $cred -ScriptBlock {
  get-service
}
```

NOTE: To jump from machine 1 to machine 2, we can use invoke-command.
but from machine 2 to machine 3 to will throw errors as invoke-command is low privileged command.
to resolve this, we have "credssp"

CMDLET:

```
> Get-WSManCredSSP
> Enable-WSManCredSSP -Role Client -DelegateComputer {domain-name}
> Get-WSManCredSSP
```

URL: https://4sysops.com/archives/enable-powershell-remoting/

Non-persistent remoting:

```
<#
  Remoting:
  - accessing the remote server using CLI. (port number 5985/5986)
  - RDP: accesing remote svr using GUI. (port number: 3389)
```

```
   two ways of remoting:
   1. Persitent method
      - where the connection, doesnot break/ disconnect automatically.
      CMD:- Get-Command -Noun pssession

   2. Non-Persitent method
      - where the connection, WILL break/ disconnect automatically.
      CMD:- Invoke-Command
#>

#Non-Persitent method
Invoke-Command -ComputerName member1 -ScriptBlock{
   #New-Item -Path c:\ -Name test.txt -ItemType File -Force
   Set-Content -Path c:\test.txt -Value "I think all are SLEEPING!!!!!!!!!!!!!!!!" -Force
}


$comp1 = Get-Content C:\day4\servers.txt
Invoke-Command -ComputerName $comp1 -ScriptBlock{
   New-Item -Path c:\ -Name batman.txt -ItemType File -Force
   Set-Content -Path c:\batman.txt -Value "I bought the BANK...." -Force
}
```

Persistent remoting:

```
<#
Persitent method
  - where the connection, doesnot break/ disconnect automatically.
    CMD:- Get-Command -Noun pssession
#>

Get-Command -Noun pssession

Get-PSSession
New-PSSession -ComputerName member1

Enter-PSSession -Id 7
Get-WindowsFeature -Name web-server
Install-WindowsFeature -Name web-server -IncludeManagementTools -Verbose
Get-WindowsFeature -Name web-server

Exit-PSSession
Get-PSSession
Remove-PSSession -Id 7
Get-PSSession

New-PSSession -ComputerName member1 -Name UninstallIIS
Enter-PSSession -Name uninstalliis
Get-WindowsFeature -Name web-server
Uninstall-WindowsFeature -Name web-server -Restart -Verbose
```

16. Workflow

20 May 2020 08:20 AM

Workflow	Using Workflow	What is Work flow and its uses?
	Workflow Concepts	Working with workflows with examples
		Inline script with examples Inline to Inline calling and passing/getting inputs/outputs
		Sequence and Parallel Execution Real time examples

Windows PowerShell workflows are designed for scenarios where these attributes are required:

- Long-running activities.
- Repeatable activities.
- Frequently executed activities.
- Running activities in parallel across one or more machines.
- Interruptible activities that can be stopped and re-started, which includes surviving a reboot of the system against which the workflow is executing.

Example:

"Hello World"
}
workflow helloworld {

- Windows PowerShell Workflow is a specialized scripting technology that uses the Windows Workflow Foundation (WWF) technology of Microsoft .NET Framework 3.5 and later versions.
- A workflow script actually run Windows PowerShell commands as activities.

Parallel and sequence blocks:

- Parallel {} blocks. The activities or commands in a Parallel block are executed concurrently.
- Sequence {} blocks. The commands or activities in a Sequence are always executed one at a time

A PowerShell workflow is the PowerShell implementation of the WWF (Windows workflow Framework). It brings a cool set of functionalities such as the possibilities to execute code in parallel, to create scripts that are persist to reboot, and many more.

Basic workflow:

```
WorkFlow My-AwesomeWorkFlow {
   #My Cool Powershell commands
}
```

Workflow working:

```
workflow Test-it {
    parallel {
        #The following commands will be executed in parallel
            New-item -ItemType file "C:\temp\TagFile.txt" -Force
            move-item "C:\temp\TagFile.txt" "c:\"
            Get-Service "WinRm"
    }

}
Test-it
```

Parallel-Sequence:

```
workflow Test-Workflow {
    "This will run first"

    parallel {
        "Command 1"
        "Command 2"
        sequence {
            "Command A"
            "Command B"
        }
        "Command 3"
        "Command 4"

    }
}
cls
Test-Workflow
```

Parallel-sequence:

```
Workflow Test-Workflow {
    "This will run first"
    parallel {
        "Command 1"
        "Command 2"
    sequence {
            "Command A"
            "Command B"
            "Command C"
            "Command D"
            "Command E"
            "Command F"
            "Command G"
            "Command H"
        }
        "Command 3"
        "Command 4"
        "Command 5"
        "Command 6"
        get-service | measure
        "Command 7"
        Get-Process | measure
    }
}
Test-Workflow
```

17. Desired State Configuration (DSC)

20 May 2020 08:21 AM

What is DSC?	Introduction to DSC
Why DSC?	Use of DSC DSC Features
Concepts	Overview of the DSC Concepts Working of DSC with basic examples

PS – DSC

- Desired State Configuration (DSC) is a powerful configuration technology that is included in Windows Management Framework (WMF) 4.0.
- On Windows 8.1 and Windows Server 2012 R2 operating systems, you must make sure that update KB2883200 is installed.
- Without that update, you will be unable to write configuration scripts or process Managed Object Format (MOF) files.
- The overall purpose of DSC is to accept a configuration file that describes how a given node (computer) should be configured, and to make sure that the computer is always configured as specified.
- Actually, configuring the computer to match your description.
- DSC consists of three main components.
    - **Extensions to the Windows PowerShell scripting language**. These extensions enable you to use Windows PowerShell to write configuration scripts. Windows PowerShell compiles these scripts to MOF files.
    - **The Local Configuration Manager (LCM).** The LCM runs on all computers from WMF 4.0. It reads MOF files and runs the DSC resources specified by the MOF files.
    - The LCM can be configured to check a web server or a file server for new MOF files. It is usually configured to recheck its configuration every 15 minutes & reconfigures the node automatically.
    - **DSC resources**. These are special Windows PowerShell script modules that check a managed node's current configuration and configure a node as specified.
- Creating DSC Configuration Files:
    - A configuration script consists of a main Configuration element. That includes one or more Node elements.
    - Configuration elements can accept input parameters.

The main advantages of DSC are:

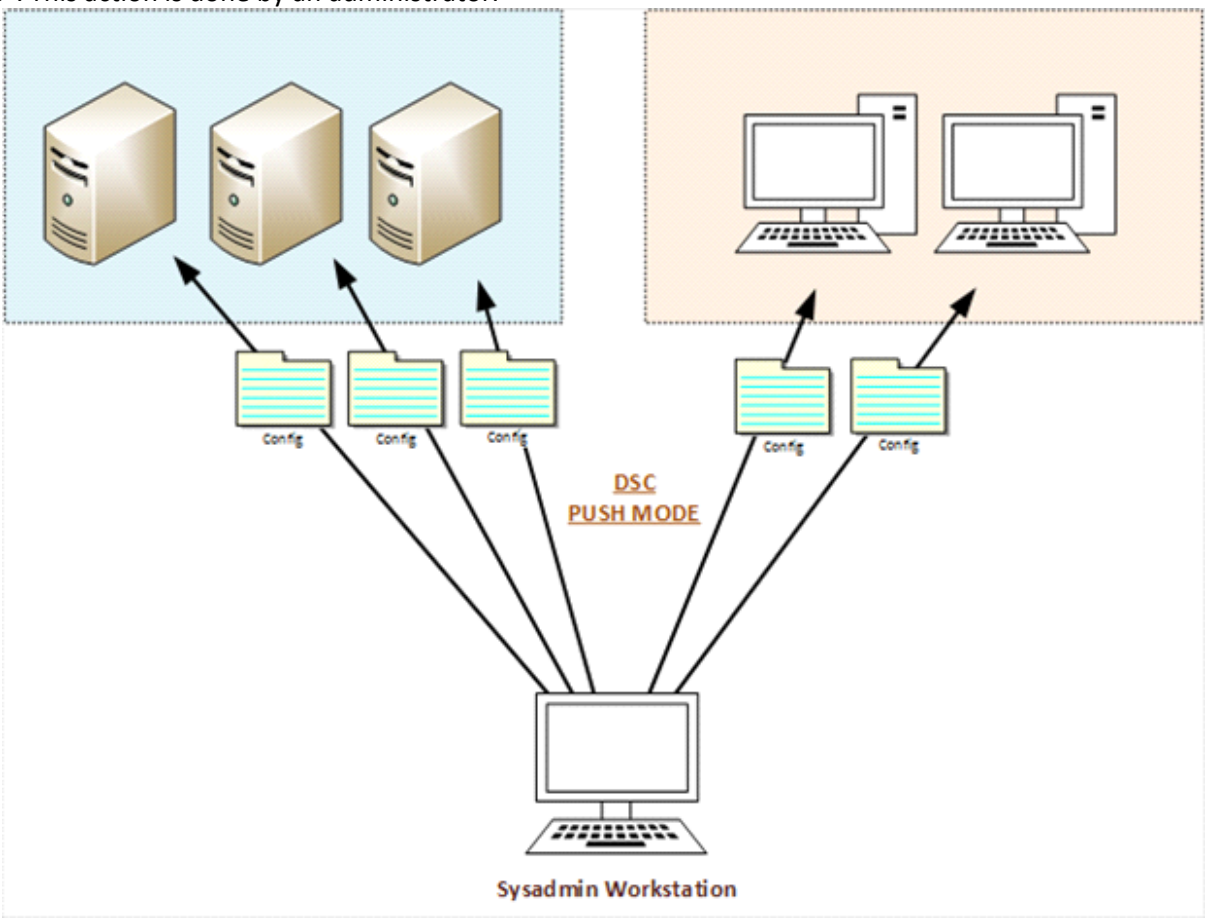
- To simplify your sysadmin task by configuring one or more devices automatically
- To be able to configure machines identically with the aim to standardise them
- To ensure, at a given time, that the configuration of a machine always be identical to its initial configuration, so as to avoid drift
- Deployment on demand as a Cloud strategy, is largely automated and simplified

Desired State Configuration is integrated to Framework 4.0, which means that the compatible platforms with DSC are:

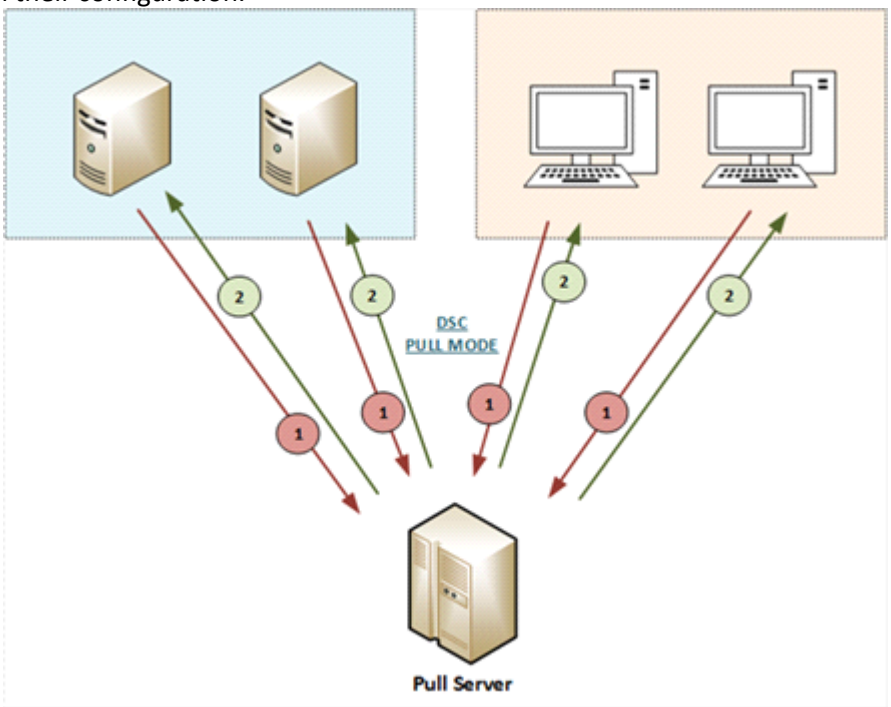
There are two types of architecture with DSC:

- **Push mode**: the configurations are sent ("pushed") manually towards one or more units that we call

"node". This action is done by an administrator.



- **Pull mode**: a "pull server" is created and the nodes contact this server at regular intervals so as to obtain their configuration.



The advantages of PUSH mode:

- Set up costs.
- The simplicity of the architecture
- It is ideal to test the functioning of "Desired State Configuration"

The disadvantages of PUSH mode:

- The complexity required to manage the machines connected.

The advantages of PULL mode:

- Automation of the deployment of the configurations
- The management of numerous machines, connected or not. As soon as the machine is connected to the network, it asks to the “Pull Server” for its configuration.

The disadvantages:

- You need to deploy one more server

DSC Resources:

- To display the configuration settings for the Service DSC resource, run the following command.
- Get-DSCResource Service | Select –Expand Properties

List of resources:

- File
- Archive
- Environment
- Log
- Package
- Registry
- Script
- Service
- User

Execute below code:

```
<#
   - DSC
      - Desired state config
      - admin implements the rules over the environment.
      - in case the user breaks the rules, DSC will check the implemented rules after a
      specific time/duration
      - 2 modes is DSC
         1. Push mode -> implementing
         2. Pull mode


#>
#DSC code config
cls
Import-Module -Name PSDesiredStateConfiguration

configuration bits-service{

   Node member1 {
      service bits{
         Name = "BITS"
         State = "Running"
      }
   }

}

#generate the MOF file.
bits-service
```

```
#apply config
Start-DscConfiguration -Path C:\day5\bits-service -Wait -Verbose

#testing the config
Test-DscConfiguration -CimSession member1
```

```
cls

Import-Module -Name PSDesiredStateConfiguration

configuration IIS-service{

   Node member1 {
      WindowsFeature IIS{

         name = "web-server"
         Ensure = "present"

      }
   }

}

#generate the MOF file.
IIS-service

#apply config
Start-DscConfiguration -Path C:\day5\IIS-service -Wait -Verbose

#testing the config
Test-DscConfiguration -CimSession member1
```

DSC PUSH mode: Configuration

On client system,

- Get-windowsfeature -name web-server

```
PS C:\Users\administrator.ALPHA> get-windowsfeature -name web-server

Display Name                                            Name                        Install State
------------                                            ----                        -------------
[ ] Web Server (IIS)                                    Web-Server                      Available
```

Syntax:

```
Configuration IIS
{
   Node $comp
   {
        WindowsFeature IIS
        {
            Ensure = “Present”
            Name = “Web-Server”
        }
   }
}
```

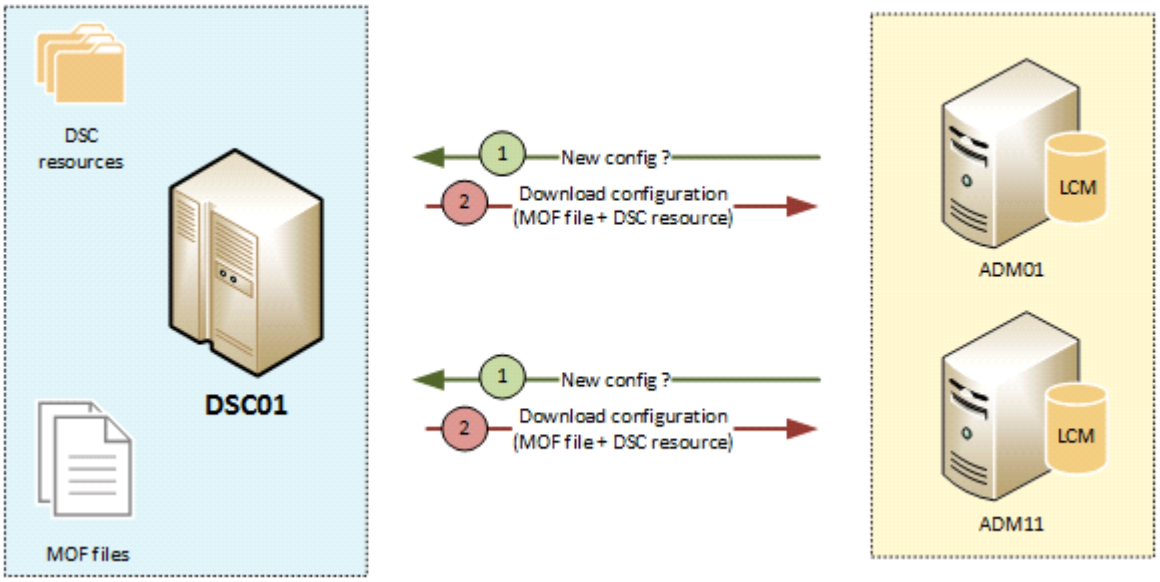
DSC: enabling LCM config on target

```
#this will change the LCM time interval to 30mins with auto update.
#-----------------------------------------------------------------
#get DSC cmdlets
Get-Command -Noun dsc*

#get available resources
Get-DscResource | select Name

#what we can configure for each resource
Get-DscResource service | select -ExpandProperty properties

#generate LCM config
configuration LCMConfig {
    Param(
        [string]$computername="localhost"
    )
    #target node
    Node $computername {
        LocalConfigurationManager {
            ConfigurationMode = "ApplyAndAutoCorrect"
            ConfigurationModeFrequencyMins = 30
        }
    }
}
#generate MOF file
#instead of member .\comp.txt can be passed as variable
LCMConfig -computername member

#check LCM settings on target node.
Get-DscLocalConfigurationManager -CimSession member

#apply LCMconfig on target node.
Set-DscLocalConfigurationManager -Path .\LCMConfig

#check LCM config
Get-DscLocalConfigurationManager -CimSession member
```

DSC PUSH: starting BITS service

```
Configuration bits-service {
    param(
        [string]$computername = "localhost"
    )

    Node $computername {
        service bits {
            Name = "BITS"
            state = "Running"
        }
    }
}

#generate MOF files:
bits-service -computername member

#apply config to target.
Start-DscConfiguration -Path .\bits-service -Wait -Verbose

#view deployed config
Get-DscConfiguration -CimSession member

#test config
Test-DscConfiguration -CimSession member
```

DSC PULL mode: Configuration



- A pull server is just an Internet Information Services (IIS) Web server that will contain the MOF files for your nodes.
- It will also respond to requests from each node's Local Configuration Manager (LCM) to distribute configurations

A pull server can distribute the MOF files via two protocols:

- **Server Message Block (SMB):** This method uses shared folders on a file server.
- **HyperText Transfer Protocol (http/https)**: This is the easiest way to manage machines located in different networks. It uses http protocol and just requires an IIS server for communications.

URL: ***https://www.red-gate.com/simple-talk/sysadmin/powershell/powershell-desired-state-configuration-pull-mode/***

Installation of the Pull Server:

- **Note**: You must install the "DSC resources kit" in order to use the resource xDscWebService contained in the xPSDesiredStateConfiguration module.
- To install a Pull server, you will need to add the "Desired State Configuration" Windows feature and configure two Web Services.
  - The first web service will allow nodes to download their configuration and
  - Second is called "Compliance Server" to avoid configuration-drift on the nodes.

CreatePullServer.ps1

```
configuration CreatePullServer
{
     param
     (
          [string[]]$ComputerName = 'localhost'
     )
Import-DscResource -ModuleName xPSDesiredStateConfiguration
Node $ComputerName
{
     WindowsFeature DSCServiceFeature
     {
          Ensure = "Present"
          Name   = "DSC-Service"
     }
     xDscWebService PSDSCPullServer
     {
          Ensure = "Present"
          EndpointName = "PSDSCPullServer"
                Port = 8080
                PhysicalPath = "$env:SystemDrive\inetpub\wwwroot\PSDSCPullServer"
                CertificateThumbPrint = "AllowUnencryptedTraffic"
```

```
                ModulePath = "$env:PROGRAMFILES\WindowsPowerShell\DscService
\Modules"
                ConfigurationPath = "$env:PROGRAMFILES\WindowsPowerShell
\DscService\Configuration"
                State = "Started"
                DependsOn = "[WindowsFeature]DSCServiceFeature"
                UseSecurityBestPractices = $false
        }
    xDscWebService PSDSCComplianceServer
        {
            Ensure = "Present"
            EndpointName = "PSDSCComplianceServer"
            Port = 9080
            PhysicalPath = "$env:SystemDrive\inetpub\wwwroot
\PSDSCComplianceServer"
            CertificateThumbPrint = "AllowUnencryptedTraffic"
            State = "Started"
            DependsOn = ("[WindowsFeature]DSCServiceFeature", "[xDSCWebService]
PSDSCPullServer")
            UseSecurityBestPractices = $false
        }
    }
}

CreatePullServer
```

run the following command to configure our Pull Server:

- PS> Start-DSCConfiguration -Path "C:\DSC\CreatePullServer" -Wait -Verbose

Setting up the Pull Server:

verify, is the DSC feature installed?

- PS C:\Users\administrator> Get-WindowsFeature -Name DSC-Service

Secondly, were two Web Services created?

- PS C:\Users\administrator> Get-Website | ft -Autosize

Everything seems fine. Last check using the Get-DscConfiguration cmdlet that will display the current DSC configuration (remember the "current.mof" file. This cmdlet will query this file).

```
PS C:\Users\administrator> Get-DscConfiguration
```

We can validate the proper functioning of the web service from any node through a web browser (from above command "get-dscConfiguration":

- http://dc.alpha.corp:9080/PSDSCPullServer.svc

Install IIS web server management tools using Server dashboard.

```
configuration CreatePullServer
{
    param
    (
        [string[]]$ComputerName = 'localhost'
    )
Import-DscResource -ModuleName xPSDesiredStateConfiguration
Node $ComputerName
{
    WindowsFeature DSCServiceFeature
    {
        Ensure = "Present"
        Name   = "DSC-Service"
    }
    xDscWebService PSDSCPullServer
    {
        Ensure = "Present"
        EndpointName = "PSDSCPullServer"
```

```
                Port = 8080
                PhysicalPath = "$env:SystemDrive\inetpub\wwwroot\PSDSCPullServer"
                CertificateThumbPrint = "AllowUnencryptedTraffic"
                ModulePath = "$env:PROGRAMFILES\WindowsPowerShell\DscService
\Modules"
                ConfigurationPath = "$env:PROGRAMFILES\WindowsPowerShell
\DscService\Configuration"
                State = "Started"
                DependsOn = "[WindowsFeature]DSCServiceFeature"
                UseSecurityBestPractices = $false
        }
     xDscWebService PSDSCComplianceServer
        {
            Ensure = "Present"
            EndpointName = "PSDSCComplianceServer"
            Port = 9080
            PhysicalPath = "$env:SystemDrive\inetpub\wwwroot
\PSDSCComplianceServer"
            CertificateThumbPrint = "AllowUnencryptedTraffic"
            State = "Started"
            DependsOn = ("[WindowsFeature]DSCServiceFeature", "[xDSCWebService]
PSDSCPullServer")
            UseSecurityBestPractices = $false
        }
    }
}

CreatePullServer

#run the following command to configure our Pull Server:
Start-DSCConfiguration -Path "C:\Users\Administrator\CreatePullServer" -Wait -
Verbose

#verify, is the DSC feature installed?
Get-WindowsFeature -Name DSC-Service

#Secondly, were two Web Services created?
Get-Website | ft -Autosize

#Everything seems fine. Last check using the Get-DscConfiguration cmdlet that
#will display the current DSC configuration (remember the “current.mof” file.
#This cmdlet will query this file).
Get-DscConfiguration


#create new GUID
$guidmember= [GUID]::NewGuid()

configuration LCMPullMode
{
    param
    (
        [string]$ComputerName,
        [string]$GUID
    )

    node $ComputerName
    {
        LocalConfigurationManager
        {
            ConfigurationID                = $GUID
            ConfigurationMode              = 'ApplyAndAutocorrect'
            RefreshMode                    = 'Pull'
            DownloadManagerName            = 'WebDownloadManager'
            DownloadManagerCustomData = @{
               ServerUrl = 'http://dc.alpha.corp:9080/PSDSCPullServer.svc'
               AllowUnsecureConnection = 'true' }
            RebootNodeIfNeeded             = $true
        }
    }
}

LCMPullMode -ComputerName member -GUID 0be808c7-2bbb-434a-8125-baba21faf57d

#Now, we can apply the metaconfiguration on the nodes:
Set-DscLocalConfigurationManager -Path  C:\Users\Administrator\LCMPullMode

#What remains is to verify if changes are applied on both nodes. Everything seems
ok.
$session1 = New-CimSession -ComputerName member
Get-DscLocalConfigurationManager -CimSession $session1

#Let’s check that the pull server’s URL is correct:
(Get-DscLocalConfigurationManager).DownloadManagerCustomData
```

```powershell
#Here is the configuration that we will be using:
configuration AudioServiceConfig
{
    param([string[]] $computerName)
    node $computerName
    {
        Service Audio
        {
            Name        = 'Audiosrv'
            StartupType = 'Automatic'
            State       = 'Running'
        }
    }
}
AudioServiceConfig -ComputerName member

(Get-DscResource -Name Service).Properties | ft -AutoSize

New-DSCCheckSum *

$destination = 'C:\Program Files\WindowsPowerShell\DscService\Configuration'

copy 'C:\Users\Administrator\LCMPullMode' -Destination $destination
```

Download images using invoke-webrequest

Tuesday, October 18, 2022 9:22 PM

DOWNLOAD MULTIPLE IMAGES FROM A WEBSITE:

```
#search on google.com --> "cute dog pinterest"

$wc = New-Object System.Net.WebClient
$req = Invoke-WebRequest -Uri "https://in.pinterest.com/petloversden/cute-dog-pictures/" -
UseBasicParsing
$images = $req.Images | Select -ExpandProperty src
$count = 0
foreach($img in $images){
  $wc.DownloadFile($img,"C:\Users\Demo\dem\img$count.jpg")
  $count++
}
#[Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::Tls12
```

DOWNLOAD PDF FROM INTERNET:

```
$source = "https://udaygade.files.wordpress.com/2015/04/linux-bible-by-christopher-negus.pdf"
$destination = "C:\Users\Demo\dem\linuxBible.pdf"
Invoke-WebRequest -Uri $source -OutFile $destination -Verbose
```

GET WEBSITE CONTENT ON SCRIPT:

```
Invoke-WebRequest -Uri https://en.wikipedia.org/wiki/PowerShell
$resp = Invoke-WebRequest -Uri https://en.wikipedia.org/wiki/PowerShell
$resp
$resp |gm
$resp.Content
$resp.RawContent
$resp.ParsedHtml
```

ACCESS ANY WESBITE DETAILS ON PS SCRIPT WITH DIFFERENT CREDS:

```
$creds = Get-Credential
wget -Uri "https://mail.google.com/" -Credential $creds
$mail = wget -Uri "https://mail.google.com/" -Credential $creds
$mail | gm
$mail.AllElements.Count
$mail.AllElements
$mail.Links | foreach {$_.href}
$mail.Images
$mail.Headers
$mail.StatusCode
```

Web scraping using PS

Tuesday, October 18, 2022 10:41 PM

USING GOOGLE.COM FOR WEB SCRAPING

```
$google = Invoke-WebRequest –Uri google.com
$google.Links | ft
$google.Links | select href | ft -AutoSize -Wrap
$google.Links | select innerText, href
```

DOWNLOAD IMAGES FROM WEBSITES.

```
#getting images
$news = Invoke-WebRequest –Uri ndtv.com –UseBasicParsing
$news.Images
$news.Images.src

#download images
@($news.Images.src).foreach({
$fileName = $_ | Split-Path -Leaf
Write-Host “Downloading image file $fileName”
Invoke-WebRequest -Uri $_ -OutFile “C:$fileName”
Write-Host ‘Image download complete’
})
```

```
#getting images
$pint = Invoke-WebRequest –Uri reddit.com
$pint.Images
$pint.Images.src

#download images
@($pint.Images.src).foreach({
   $fileName = $_ | Split-Path -Leaf
   Write-Host “Downloading image file $fileName”
   Invoke-WebRequest -Uri $_ -OutFile “C:$fileName”
   Write-Host ‘Image download complete’
})
```

VM creation using PS

Tuesday, November 22, 2022 6:57 PM

```
$ErrorActionPreference = "stop"
<#
   #check if AZ module is installed or not
   Get-Module -Name AZ
   Find-Module -Name AZ
   Install-Module -Name AZ -Force
   Import-Module Az
#>
#login to azure account
   #Login-AzAccount
   Disconnect-AzAccount
   Connect-AzAccount

   Write-Host "Establishing connection" -ForegroundColor Cyan

#list the active subscription details
   #Get-AzResourceGroup
   #Get-AzResourceGroup | Select-Object ResourceGroupName, Location, ProvisioningState

#get new RG name, location info
   $rgname = Read-Host "Enter a new RG name"
   $location = Read-Host "Enter location (like: eastus, westus): "

#create a new RG
   New-AzResourceGroup -Name $rgname -Location $location
   Write-Host "RG created" -ForegroundColor Cyan

#get vNet, subnet, dns server info
#create new virtual network
   $vnetname = $($rgname+$location)+"01"
   $vnet = New-AzVirtualNetwork `
   -ResourceGroupName $rgname `
   -Location $location `
   -Name $vnetname `
   -AddressPrefix 172.16.0.0/16

   Write-Host "VNET created" -ForegroundColor Cyan
#create subnet - 1st subnet
   $subnet = Add-AzVirtualNetworkSubnetConfig `
   -Name sub-1 `
   -AddressPrefix 172.16.1.0/24 `
   -VirtualNetwork $vnet

   Write-Host "1st subnet created" -ForegroundColor Cyan

#associating subnet with vnet - write subnet to the vnet - 1st subnet
   $vnet | Set-AzVirtualNetwork

#create subnet - 2nd subnet
   $subnet = Add-AzVirtualNetworkSubnetConfig `
   -Name sub-2 `
```

```
   -AddressPrefix 172.16.2.0/24 `
   -VirtualNetwork $vnet

   Write-Host "2nd subnet created" -ForegroundColor Cyan

#associating subnet with vnet - write subnet to the vnet - 2nd subnet
   $vnet | Set-AzVirtualNetwork

#setting custom DNS
   $array = @("172.16.1.4","8.8.8.8","8.8.4.4")
   $newObject = New-Object -type PSObject -Property @{"DnsServers" = $array}
   $vnet.DhcpOptions = $newObject
   $vnet | Set-AzVirtualNetwork

   Write-Host "DNS added" -ForegroundColor Cyan

#create 1st VM with existing vnet
   $vmname = "dcvm01"
   $image = "Win2016Datacenter"
   $size = "Standard_DS3_v2"
   $vnet = $vnet
   $vnsg = "myvnet010101"
   $mypubip = "myvirpubip"

#create NSG rules

#create NSG
   $rdprule = New-AzNetworkSecurityRuleConfig -Name ps-rdp `
   -Description "It allows RDP - Allow access" `
   -Access Allow `
   -Protocol Tcp `
   -Direction Inbound `
   -Priority 100 `
   -SourceAddressPrefix internet `
   -SourcePortRange * `
   -DestinationAddressPrefix * `
   -DestinationPortRange 3389

$sshrule = New-AzNetworkSecurityRuleConfig -Name ps-ssh `
   -Description "It allows SSH - Allow access" `
   -Access Allow `
   -Protocol Tcp `
   -Direction Inbound `
   -Priority 101 `
   -SourceAddressPrefix internet `
   -SourcePortRange * `
   -DestinationAddressPrefix * `
   -DestinationPortRange 22

$nsg = New-AzNetworkSecurityGroup `
   -ResourceGroupName $rgname `
   -Name psmynsg010101 `
   -Location $location `
   -SecurityRules $rdprule, $sshrule `
   -Force

   Write-Host "RDP, SSH rules added to the NSG for DC" -ForegroundColor Cyan
```

```
#spin a new VM
New-AzVM -Name $vmname `
   -ResourceGroupName $rgname `
   -Location $location `
   -VirtualNetworkName $vnet `
   -AddressPrefix 172.16.0.0/16 `
   -SubnetAddressPrefix 172.16.1.0/24 `
   -SubnetName "sub-1" `
   -Credential (Get-Credential) `
   -PublicIpAddressName $mypubip `
   -OpenPorts 3389,80 -Image $image -Size $size `
   -DataDiskSizeInGb 130 -SecurityGroupName $nsg

Write-Host "1st VM created" -ForegroundColor Cyan

#fetching public IP address
   $pip = Get-AzPublicIpAddress -Name $mypubip -ResourceGroupName $rgname
   #$pip.IpAddress

#access RDP
   #mstsc.exe /v $pip.IpAddress

#########################################################################
#create 1st VM with existing vnet
   $vmname2 = "member01"
   $image = "Win2016Datacenter"
   $size = "Standard_D2s_v3"
   $vnet = $vnet
   $vnsg2 = "myvnet020202"
   $mypubip2 = "mypubip2"

#create NSG rules

#create NSG
$rdprule = New-AzNetworkSecurityRuleConfig -Name ps-rdp `
   -Description "It allows RDP - Allow access" `
   -Access Allow `
   -Protocol Tcp `
   -Direction Inbound `
   -Priority 100 `
   -SourceAddressPrefix internet `
   -SourcePortRange * `
   -DestinationAddressPrefix * `
   -DestinationPortRange 3389

$sshrule = New-AzNetworkSecurityRuleConfig -Name ps-ssh `
   -Description "It allows SSH - Allow access" `
   -Access Allow `
   -Protocol Tcp `
   -Direction Inbound `
   -Priority 101 `
   -SourceAddressPrefix internet `
   -SourcePortRange * `
   -DestinationAddressPrefix * `
   -DestinationPortRange 22
```

```
$nsg = New-AzNetworkSecurityGroup `
   -ResourceGroupName $rgname `
   -Name psmynsg020202 `
   -Location $location `
   -SecurityRules $rdprule, $sshrule `
   -Force

Write-Host "RDP, SSH rules added to the NSG for MEMBER01" -ForegroundColor Cyan

#spin a new VM
New-AzVM -Name $vmname2 `
   -ResourceGroupName $rgname `
   -Location $location `
   -VirtualNetworkName $vnet `
   -AddressPrefix 172.16.0.0/16 `
   -SubnetAddressPrefix 172.16.1.0/24 `
   -SubnetName "sub-1" `
   -Credential (Get-Credential) `
   -PublicIpAddressName $mypubip2 `
   -OpenPorts 3389,80 -Image $image -Size $size `
   -DataDiskSizeInGb 130 -SecurityGroupName $nsg

#fetching public IP address
   $pip2 = Get-AzPublicIpAddress -Name $mypubip2 -ResourceGroupName $rgname
#$pip2.IpAddress

#access RDP
#mstsc.exe /v $pip2.IpAddress

#listing both IP
   Write-Host "$($vmname) IP address is $($pip.IpAddress)"
   Write-Host "$($vmname2) IP address is $($pip2.IpAddress)"

#adding required ports in both NSG

Get-AzNetworkSecurityGroup -Name "psmynsg010101" `
   -ResourceGroupName $rgname | `
   Add-AzNetworkSecurityRuleConfig -Name "ps-psremoting1" `
   -Description "Allow RDP" `
   -Access "Allow" `
   -Protocol "Tcp" `
   -Direction "Inbound" `
   -Priority 103 `
   -SourceAddressPrefix "Internet" `
   -SourcePortRange "*" `
   -DestinationAddressPrefix "*"`
   -DestinationPortRange "5985" | Set-AzNetworkSecurityGroup

Get-AzNetworkSecurityGroup -Name "psmynsg020202" `
   -ResourceGroupName $rgname | `
   Add-AzNetworkSecurityRuleConfig -Name "ps-psremoting2" `
   -Description "Allow RDP" `
   -Access "Allow" `
   -Protocol "Tcp" `
   -Direction "Inbound" `
   -Priority 104 `
   -SourceAddressPrefix "Internet" `
```

```
   -SourcePortRange "*" `
   -DestinationAddressPrefix "*"`
   -DestinationPortRange "5986" | Set-AzNetworkSecurityGroup

Write-Host "WinRM rules added to the NSG for DC & Member01" -ForegroundColor Cyan

<#
NOT WORKING
#installing ADDS role
Test-WSMan -ComputerName $pip.IpAddress
$cred = Get-Credential
Invoke-Command -ComputerName dcvm01-bd7d4d.eastus.cloudapp.azure.com -Credential $cred -
ScriptBlock{
   New-Item -Path c:\ -Name test.txt -Force
}
#>
```