Introduction to PS

PowerShell is a task automation and configuration management framework developed by Microsoft.
It consists of a command-line shell and an associated scripting language, designed primarily for system administrators and power users.
PowerShell is built on the .NET framework and offers deep integration with the Windows operating system, although it has since evolved into **PowerShell Core**, which is cross-platform (available on Windows, macOS, and Linux).

History of PowerShell
PowerShell was first introduced in 2006 as a replacement for the legacy Windows Command Prompt and Windows Script Host (VBScript, JScript). It was designed to automate tasks and manage system configurations, particularly in enterprise environments.

PowerShell's development follows these major versions:

- **PowerShell 1.0** (2006): Introduced core cmdlets and basic scripting capabilities.
- **PowerShell 2.0** (2009): Added remoting and more cmdlets, introduced the Integrated Scripting Environment (ISE).
- **PowerShell 3.0 & 4.0** (2012, 2013): Added workflows, enhanced usability, and more cmdlets for managing Windows environments.
- **PowerShell 5.0 & 5.1** (2016): Enhanced security features, support for classes, and expanded module support. PowerShell 5.1 was the last version tied to Windows.
- **PowerShell Core 6.0** (2018): Became cross-platform, running on .NET Core.
- **PowerShell 7.0** (2020): Built on .NET Core 3.1 and added many modern features, further extending its cross-platform capabilities.

PowerShell Features:

1. **Cmdlets** (Command-lets): These are specialized .NET classes that perform a specific operation. PowerShell has hundreds of built-in cmdlets for managing various system tasks, such as:
    - Get-Process: Retrieves processes running on a system.
    - Get-Service: Lists all services.
    - New-Item: Creates a new item (file, folder, etc.).

2. **Object-Based Pipeline**: One of PowerShell's key strengths is its pipeline, which passes objects between commands, not just text. This allows more complex data manipulation, as the output from one cmdlet can be easily piped as input to another.
   Ex: `Get-Process | Where-Object { $_.CPU -gt 100 }`

3. **Scripting Language**: PowerShell has a rich scripting language that supports variables, loops, conditional statements, and complex logic. This allows users to write scripts to automate repetitive tasks.

4. **Modules**: PowerShell allows for extensibility through modules, which are packages that can contain additional cmdlets, functions, and workflows. You can load these modules dynamically to access new features:
    - Import-Module ActiveDirectory (AD management tools)
    - Import-Module Azure (Azure cloud tools)

5. **Remoting**: PowerShell supports remoting, which allows you to run commands on remote machines via WinRM (Windows Remote Management). This is essential for managing large server environments.

6. **Jobs**: PowerShell can perform asynchronous tasks through jobs, enabling background execution.

7. **Desired State Configuration (DSC)**: PowerShell DSC allows you to define the desired configuration of a system (like a web server or database server), and PowerShell ensures that the system is in this state.

Common PowerShell Cmdlets:
PowerShell cmdlets generally follow a verb-noun naming convention, which makes them intuitive and easy to understand. Some commonly used cmdlets include:

System Management:
- **Get-Process**: Retrieves information about processes.
- **Stop-Process**: Stops a running process.
- **Get-Service**: Lists all services.
- **Start-Service**: Starts a service.
- **Restart-Computer**: Restarts the machine.

File Management:
- **Get-ChildItem**: Lists the files and directories.
- **Copy-Item**: Copies files or folders.
- **New-Item**: Creates new files or directories.
- **Remove-Item**: Deletes files or directories.

User and Security Management:
- **Get-LocalUser**: Retrieves local users on a machine.
- **Set-LocalUser**: Changes properties of a local user.
- **Add-LocalGroupMember**: Adds a user to a local group.

Networking:
- **Test-Connection**: Pings a machine (like ping in cmd).
- **Get-NetIPAddress**: Lists all IP addresses on the system.

Scripting in PowerShell
- PowerShell scripts are text files with a .ps1 extension that contain one or more PowerShell commands.
- A script can be as simple as a series of cmdlets, or it can include loops, conditionals, functions, and other programming logic.

```
# This script gets a list of services and stops services that are running
$services = Get-Service
foreach ($service in $services) {
    if ($service.Status -eq 'Running') {
        Stop-Service -Name $service.Name
    }
}
```

Execution Policy: To run a script, the execution policy must allow script execution. PowerShell has several execution policies for security purposes:

- **Restricted**: No scripts are allowed to run.
- **RemoteSigned**: Scripts must be signed by a trusted publisher.
- **Unrestricted**: All scripts can run, but warnings are shown for scripts downloaded from the internet.

PowerShell ISE and VS Code

- *PowerShell ISE (Integrated Scripting Environment)*: It is a graphical interface where you can write, run, and debug scripts. It supports IntelliSense, code highlighting, and debugging features.
- *Visual Studio Code (VS Code):* PowerShell development is shifting towards VS Code, where the PowerShell extension provides similar or better features than the ISE, along with cross-platform support.

Use Cases

- System Administration: Automating system configurations, monitoring, and maintenance.
- Cloud Management: Managing Azure resources using the Azure module.
- DevOps: Building scripts to automate deployment pipelines and configuration management.

Data Structures in PS

Variables in PowerShell

- A variable is a symbolic name that stores data. In PowerShell, variables are used to hold objects, and their name always begins with a dollar sign ($).
- Variables can hold any type of data, including numbers, strings, arrays, or even complex objects like processes or services.
- Declaring and Assigning a Variable:

```
# Assign a value to a variable
$myVariable = "Hello, PowerShell"
$myNumber = 100
$myBoolean = $true
```

```
# Variables
$a = 10
$a.GetType()

$b = 10.10
$b.GetType()

$c = "Hello"
$c.GetType()

$d = Get-Date
$d.GetType()
$d.Date
$d.Day
$d.DayOfWeek
$d.DayOfYear
$d.Minute
$d.Second

(Get-Date).DayOfWeek
```

Arrays:

- 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"
```

```powershell
# Arrays
$arr1 = (1,2,3,4,5)
$arr1
$arr1.GetType()

# PS way to define an array
$arr2 = @(6,7,8,9,0)
$arr2.GetType()
$arr2.Count
$arr2[0]
$arr2[-1]

#multi-dimentional array
$arr3 = @(
    @("abc","xyz"),
    @(1,2,3),
    @(Get-Process)
)
$arr3.Count
$arr3[0][0]
$arr3[1][2]
$arr3[2][2]
```

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")

```powershell
# Hashtables - key-value pairs
$ht1 = @{}
$ht1.GetType()

#unordered hashtable
$ht2 = @{Name="Jeetu" ; client = "LTIMindtree" ; Batch = "35.2"}
$ht2

#ordered hashtable
$ht3 = [ordered]@{Name="Jeetu" ; client = "LTIMindtree" ; Batch = "35.2"}
$ht3
$ht3.Keys
$ht3.Values
$ht3.Count

# Adding element to HT
$ht3.Add("ClassRoom","Kachnar")
$ht3

# Modifying element from HT
$ht3["Name"] = "Jitendra Singh Tomar"
$ht3

# Deleting element from HT
$ht3.Remove("Batch")
$ht3

cls
Get-Service | Select-Object Name, `
DisplayName, `
@{name="Report"; exp={$_.Status}} `
-First 3
```

Task:

- Create an array that stores integer from 1 to 100 and print them.
- Create a multi-dimensional array with various data types like integer, float and strings. Verify it by printing the array.
- Create an unordered hash table and print it.
- Create an ordered hash table and perform the following operations:
    - Adding new entry in hash table
    - Modify an existing entry in hash table
    - Remove an existing entry.

Objects

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

```
# Objects

Get-Date | Get-Member
Get-Process | Get-Member
Get-Service | gm

# PS Custom Objects
$obj = New-Object -TypeName psobject
$obj.GetType()
$obj | Get-Member

# adding a new member to the object
Add-Member -InputObject $obj -MemberType NoteProperty -Name "LTIM" -Value "Batch35.2"
$obj | Get-Member

# adding multiple members to the object
$newobject = @{
    prop1 = "value1"
    prop2 = "value2"
    prop3 = "value3"
    prop4 = "value4"
}

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

Task:

---

1. List existing members of the following commands:
    a. Get-date
    b. Get-process
    c. Get-service
2. Create a new object and add a new member to its object/member list.
3. Create a new object and add multiple members as property at once.

Conditional Operators

```
PS E:\PowerShell\batch35.2> Get-Help about_*operator*

Name                                Category  Module
----                                --------  ------
about_Arithmetic_Operators          HelpFile
about_Assignment_Operators          HelpFile
about_Comparison_Operators          HelpFile
about_Logical_Operators             HelpFile
about_Operators                     HelpFile
about_Operator_Precedence           HelpFile
about_Type_Operators                HelpFile
```

PowerShell provides a wide range of conditional operators that you can use to compare values, perform logic-based decisions, and manage control flow. These operators are essential for writing conditional statements like if, elseif, and else.

Comparison Operator:

Numeric Comparisons:

Operator	Description	Example
`-eq`	Equal to	`5 -eq 5` (True)
`-ne`	Not equal to	`5 -ne 3` (True)
`-gt`	Greater than	`5 -gt 3` (True)
`-ge`	Greater than or equal to	`5 -ge 5` (True)
`-lt`	Less than	`3 -lt 5` (True)
`-le`	Less than or equal to	`3 -le 5` (True)

String Comparisons:

Operator	Description	Example
`-eq`	Equal to	`"PowerShell" -eq "PowerShell"`
`-ne`	Not equal to	`"PowerShell" -ne "Windows"`
`-like`	Wildcard comparison (supports `*` and `?`)	`"file.txt" -like "*.txt"`
`-notlike`	Wildcard not like	`"file.txt" -notlike "*.pdf"`
`-match`	Regular expression match	`"abc123" -match "\d+"` (True)
`-notmatch`	Regular expression not match	`"abc123" -notmatch "\d+"` (False)
`-replace`	Replace a substring using regex	`"abc123" -replace "\d", "X"` (Outputs "abcXXX")

Logical Operators:

Operator	Description	Example
`-and`	Logical AND	`$true -and $true` (True)
`-or`	Logical OR	`$true -or $false` (True)
`-not`	Logical NOT	`-not $true` (False)
`!`	Shortcut for `-not`	`!($true)` (False)
`-xor`	Logical XOR (exclusive)	`$true -xor $false` (True)

Array and Collection Operators:

Operator	Description	Example
`-contains`	Checks if an array contains an element	`@(1,2,3) -contains 2` (True)
`-notcontains`	Array does not contain the element	`@(1,2,3) -notcontains 4` (True)
`-in`	Checks if value is in an array	`2 -in @(1,2,3)` (True)
`-notin`	Value is not in an array	`4 -notin @(1,2,3)` (True)

Type Operators:

Operator	Description	Example
`-is`	Checks if an object is of a specific type	`5 -is [int]` (True)
`-isnot`	Checks if an object is NOT of a specific type	`"Hello" -isnot [int]` (True)

Redirection and Pipeline Operators:

Operator	Description	Example
`	`	Pipeline operator
`>`	Redirect output to a file	`Get-Process > processes.txt`
`>>`	Append output to a file	`Get-Process >> processes.txt`
`2>`	Redirect error output to a file	`Get-Process 2> errorlog.txt`
`2>&1`	Redirect errors to standard output	`Get-Process 2>&1`

Assignment Operators:

Operator	Description	Example
`=`	Assigns a value	`$var = 10`
`+=`	Adds and assigns	`$var += 5` (same as `$var = $var + 5`)
`-=`	Subtracts and assigns	`$var -= 2`
`*=`	Multiplies and assigns	`$var *= 3`
`/=`	Divides and assigns	`$var /= 2`

```
cls
$a = 21
if ( $a -eq 2 ){
    Write-Host "Values are equal"
}

# ask user for a website and check of its pingable or not.
# show pingable site in green color
# and non-pingable in red color.

cls
$site = Read-Host "Enter website to ping "

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

Task - write powershell program that show menu and does not quits until 5 is pressed.

```
cls
$ans = ""
do{
    $ans = Read-Host "
    Select from the options:
        1. to check internet connection
        2. to display IP address
        3. to display MAC address
        4. Hostname
        5. Exit
    "
    switch($ans){
        1{
            if( Test-Connection google.com -Count 1 -ErrorAction SilentlyContinue ){
                Write-Host "its pinging" -ForegroundColor Green
            }else{
                Write-Host "its NOT pinging" -ForegroundColor Red
            }}
        2{ (Get-NetIPAddress | Where-Object {$_.PrefixOrigin -eq "Dhcp"}).IPAddress }
        3{ (Get-NetAdapter | Where-Object {$_.name -eq "Wi-Fi"}).MacAddress }
        4{ write-host "Your computer name is: " $(HOSTNAME.EXE) }
        5{ break }
        default{ Write-Warning "Invalid selection" }
    }
}while( $ans -ne "5")
```

Code

```
cls
#$ans = ""
do{
  $ans1 = Read-Host "
  Select from the options:
    1. to check internet connection
    2. to display IP address
    3. to display MAC address
    4. Hostname
    5. Exit
  "
  switch($ans1){
    1{
      if( Test-Connection google.com -Count 1 -ErrorAction SilentlyContinue ){
        Write-Host "its pinging" -ForegroundColor Green
      }else{
        Write-Host "its NOT pinging" -ForegroundColor Red
      }}
    2{ (Get-NetIPAddress | Where-Object {$_.PrefixOrigin -eq "Dhcp"}).IPAddress }
    3{ (Get-NetAdapter | Where-Object {$_.name -eq "Wi-Fi"}).MacAddress }
    4{ write-host "Your computer name is: " $(HOSTNAME.EXE) }
    5{ break }
    default{ Write-Warning "Invalid selection" }
  }
}while( $ans1 -ne "5")
```

Task

- Use IF statement and
  - Test if $PROFILE is present or not. Else create it using script.
  - Test if any website (say google.com) is pinging or not. Suppress/hide the errors for any non-pingable site.
- Create a powershell program that asks for your age and prints if you are eligible for voting on appropriate age or still under age.

Loop

Loops:

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

Do..While Loop

Ex1:	Ex2:

$array = @("item1", "item2", "item3")
$counter = 0;
 	
do {
  $array[$counter]
  $counter += 1
} while($counter -lt $array.length)
 	
item1
item2
item3
	#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)

```
# do-while loop
cls
$array = @("item1", "item2", "item3")
$counter = 0;

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

While Loop

#loops - EX1	#loops - EX2

$arr1 = @("a","b","c")
$c = 0
cls
while ($c -lt $arr1.Length){
  $arr1[$c]
  $c += 1
  sleep 1
}
$c = $null
$arr1 = $null
	> $array = @("item1", "item2", "item3")
	$counter = 0;
	
	while($counter -lt $array.length){
	  $array[$counter]
	  $counter += 1
	}
	
	item1
	item2

	item3

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

For loop:

Ex1:-	Ex2:-

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

```
# 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:-	Ex2:-

> $array = @("item1", "item2", "item3")
 	
> foreach ($element in $array) { $element }
item1
item2
item3
 	
> $array | foreach { $_ }
item1
item2
item3
	$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 $_
	}

```
# foreach loop
cls
$arr = @(1..100)
foreach( $a in $arr ){
    Write-Host $a
}
```

Code

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

# do-while loop
cls
$array = @("item1", "item2", "item3")
$counter = 0;

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

# 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
cls
$arr = @(1..100)
foreach( $a in $arr ){
   Write-Host $a
}
```

Functions and Pipelines

Function

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

#EX3:-
 	
#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
	#EX4:-
	
	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

Basic function:

Creating function using Mandate parameter.	**Creating function using validation parameter**

function mandate{
[cmdletbinding()]
param(
[Parameter(Mandatory)]
[string]$a
)
cls
Write-Host "You have sent '$a' using mandate parameters"
 	
}
 	
mandate -a haha
	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:	**To get it right:**
function mandate{ [cmdletbinding()] param(	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.

```
    [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 $_
}
```

```powershell
function add{

    [int]$num1 = Read-Host "Enter 1st number"
    [int]$num2 = Read-Host "ENter 2nd number"

    # $num1.GetType()
    # $num2.GetType()

    $num3 = $num1 + $num2

    Write-Host "Total of $num1 & $num2 is $num3"

    $num1 = $null
    $num2 = $null
    $num3 = $null

}
cls
add
```

Advance function:

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

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

```powershell
function add1{
[cmdletbinding()]
    param(
        [int][parameter(Mandatory=$true,HelpMessage="Enter a number")]$n1,
        [int][parameter(Mandatory=$true,HelpMessage="Enter a number")]$n2
    )

    Write-Host "$n1 + $n2 = $($n1+$n2)"

}
cls

# add1 -n1 20 -n2 40
add1 -n1 20
```

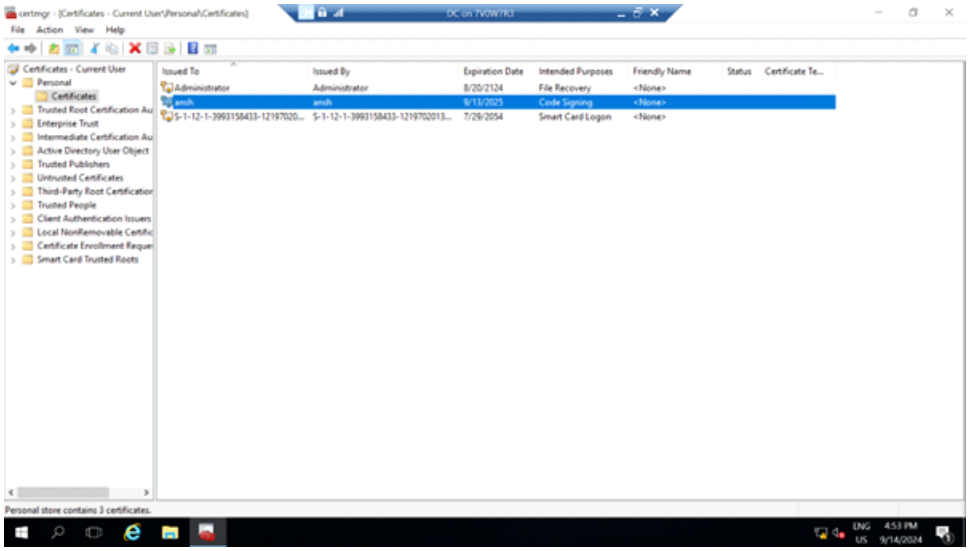
Script Execution

i. Create a new digital certificate by using command

```
New-SelfSignedCertificate -CertStoreLocation Cert:\CurrentUser\My `
-Subject "CN=anyName" `
-KeyAlgorithm RSA `
-KeyLength 1024 `
-Provider "Microsoft Enhanced RSA and AES Cryptographic Provider" `
-KeyExportPolicy Exportable `
-KeyUsage DigitalSignature `
-Type CodeSigningCert
```

Export the certificate

a. To see the certificate use keyboard combination “windows + r”
b. Now type “certmgr.msc”
c. Double click on “Personal”, double click on “certificate”



d. Now you can see your certificate name
e. Now right click on certificate and go to “All task” then click on “Export”
f. A wizard will be opened and click on next
g. Click on “yes, export the private key”
h. Click on next
i. Click on password and give the password
j. Now click on browse to save the file and give the name.
k. Click on next and then finish.

ii. Import/install certificate
a. Double click on exported Certificate clicks on next
b. Give the password and click on next
c. Now click on “Place all certificate on following store”
i. Click on Browse
ii. Click on “Trusted Root Certification Authorities”
iii. Click on next
d. Click on finish

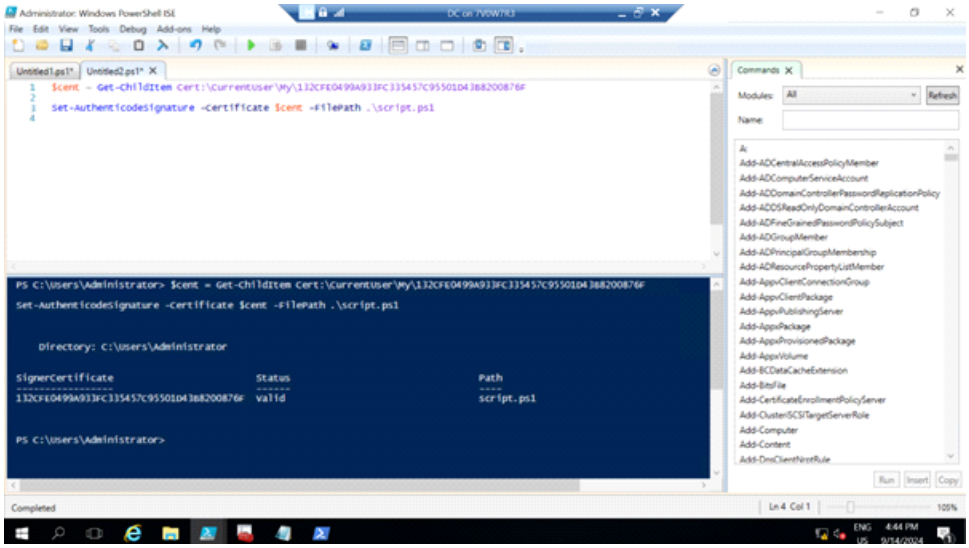
iii. Create a script
a. Open Windows PowerShell ISE
b. Create a new script with command “get-date” and save using name “script”

iv. Bind certificate and script together
a. Use the following commands

v.
```
$a = (Get-ChildItem Cert:\CurrentUser\My | Where-Object {$_.Subject -eq "CN=<certificate
canonical name>"}).Thumbprint
```

b. note down the alphanumeric string
c. use command
d. now the status should be “valid” & you are good to go

```
$cent = Get-ChildItem Cert:\CurrentUser\My\132CFE0499A933FC335457C95501D43B8200876F

Set-AuthenticodeSignature -Certificate $cent -FilePath .\script.ps1
```



 # for creating certificate

```
New-SelfSignedCertificate -CertStoreLocation Cert:\currentuser\My `
-Subject "CN=testcert" `
-KeyAlgorithm RSA `
-KeyLength 1024 `
-Provider "Microsoft Enhanced RSA and AES Cryptographic Provider" `
-KeyExportPolicy Exportable `
-KeyUsage DigitalSignature `
-Type CodeSigningCert
```

 # to list/view certificate

```
Get-ChildItem Cert:\currentuser\My                                  # show all certs
Get-ChildItem Cert:\currentuser\My\A8ACB103D2F126A58F39E75FFB9A36AC824029C3     # show ur cert
```

#to export the cert

```
Get-Help Export-PfxCertificate -Online
$cert = Get-ChildItem Cert:\CurrentUser\My\A8ACB103D2F126A58F39E75FFB9A36AC824029C3
Get-Help Export-PfxCertificate -Examples
```

 # exporting the cert using cmd

```
$cert = Get-ChildItem Cert:\CurrentUser\My\A8ACB103D2F126A58F39E75FFB9A36AC824029C3
$mypwd = ConvertTo-SecureString -String "pass@word1" -Force –AsPlainText
$cert | Export-PfxCertificate -FilePath testCert.pfx `
-Password $mypwd `
-CryptoAlgorithmOption AES256_SHA256 `
-Force
```

 # to import/install the exported certificate

```
Get-Help Import-PfxCertificate -ShowWindow
```

 # to bind the script & the cert

```
$cert = Get-ChildItem Cert:\CurrentUser\My\A8ACB103D2F126A58F39E75FFB9A36AC824029C3
Set-AuthenticodeSignature -Certificate $cert -FilePath .\test.ps1
Get-AuthenticodeSignature .\test.ps1
```

```powershell
# for creating certificate
New-SelfSignedCertificate -CertStoreLocation Cert:\currentuser\My `
-Subject "CN=testcert" `
-KeyAlgorithm RSA `
-KeyLength 1024 `
-Provider "Microsoft Enhanced RSA and AES Cryptographic Provider" `
-KeyExportPolicy Exportable `
-KeyUsage DigitalSignature `
-Type CodeSigningCert

# to list/view certificate
Get-ChildItem Cert:\currentuser\My                                          # show all certs
Get-ChildItem Cert:\currentuser\My\A8ACB103D2F126A58F39E75FFB9A36AC824029C3      # show ur cert

#to export the cert
Get-Help Export-PfxCertificate -Online
$cert = Get-ChildItem Cert:\CurrentUser\My\A8ACB103D2F126A58F39E75FFB9A36AC824029C3
Get-Help Export-PfxCertificate -Examples

# exporting the cert using cmd
$cert = Get-ChildItem Cert:\CurrentUser\My\A8ACB103D2F126A58F39E75FFB9A36AC824029C3
$mypwd = ConvertTo-SecureString -String "pass@word1" -Force -AsPlainText
$cert | Export-PfxCertificate -FilePath testCert.pfx `
-Password $mypwd `
-CryptoAlgorithmOption AES256_SHA256 `
-Force


# to import/install the exported certificate
Get-Help Import-PfxCertificate -ShowWindow


# to bind the script & the cert
$cert = Get-ChildItem Cert:\CurrentUser\My\A8ACB103D2F126A58F39E75FFB9A36AC824029C3
Set-AuthenticodeSignature -Certificate $cert -FilePath .\test.ps1
Get-AuthenticodeSignature .\test.ps1
```

Error Handling

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/

Input / Output

In PowerShell, input/output (I/O) cmdlets are essential for interacting with the user, files, the console, and other forms of input and output.
They allow you to display data, receive input, and redirect or store output for further use.

Output Cmdlets
These cmdlets display or redirect output, either to the console, a file, or another cmdlet.

a) Write-Host
- Used to send output directly to the console (stdout).
- It outputs text, which is not captured in the pipeline (i.e., you cannot pipe Write-Host output to another cmdlet).
- Ex:
    - Write-Host "Hello, World!"
    - Write-Host "This is important!" -ForegroundColor Red

b) Write-Output
- Sends objects to the output stream, which can be passed along the pipeline to other cmdlets.
- This is the default output cmdlet in PowerShell, even if you don't explicitly call it.
- Ex:
    - Write-Output "This is output text"
    - Write-Output "Hello" | Out-File "output.txt"

c) Out-File
- Redirects output to a file.
- Ex:
    - Get-Process | Out-File "processes.txt"
    - Get-Date | Out-File -FilePath "log.txt" -Append

d) Out-Host
- Sends the output directly to the console (standard output) and is used when you want to explicitly control the display of output.
- Ex:
    - Get-Process | Out-Host

e) Out-Null
- Suppresses the output of a command, effectively discarding it.
- Ex:
    - Get-Process | Out-Null

f) Export-Csv
- Exports objects (like results from a cmdlet) to a CSV file.
- Ex:
    - Get-Process | Export-Csv -Path "processes.csv" -NoTypeInformation

g) ConvertTo-Json / ConvertTo-Xml
- Converts objects to JSON or XML formats for output or data exchange.

- Ex:
    - Get-Process | ConvertTo-Json
    - Get-Process | ConvertTo-Xml

Input Cmdlets

Input cmdlets allow you to receive data from users, files, or other sources.

a) Read-Host

- Prompts the user for input and returns the data typed by the user as a string.
- Ex:
    - $input = Read-Host "Enter your name"
    - Write-Host "You entered: $input"

b) Get-Content

- Retrieves the content of a file and outputs it line by line as strings. This is often used to read data from text files.
- Ex:
    - Get-Content "file.txt"

c) Import-Csv

- Reads a CSV file and converts each row into an object, allowing easy manipulation of structured data.
- Ex:
    - Import-Csv -Path "data.csv"

```
# Reading a CSV file and processing data
$users = Import-Csv -Path "users.csv"
foreach ($user in $users) {
    Write-Host "User: $($user.Name), Age: $($user.Age)"
}
```

d) Select-String

- Searches through strings and files using regular expressions. It is similar to the grep command in Unix/Linux systems.
- Ex:
    - Select-String -Pattern "Error" -Path "log.txt"

Redirection Operators

PowerShell also supports redirection operators for input and output management. These operators allow you to send or suppress output, including errors.

- >: Redirects the output to a file, overwriting it.
    - Get-Process > processes.txt

- >>: Appends the output to a file.
    - Get-Process >> processes.txt

- 2>: Redirects error output to a file.
    - Get-Process 2> errorlog.txt

- 2>&1: Redirects both output and error to the same stream (standard output).
    - Get-Process 2>&1 | Out-File output.txt

Working with Pipelines

- PowerShell uses pipelines to pass output from one cmdlet as input to another cmdlet. The pipeline operator (|) makes this possible.
- # Get processes and filter out those using more than 100 MB of memory
    - Get-Process | Where-Object { $_.WorkingSet -gt 100MB } | Out-File "high-memory-processes.txt"

Text Processing and Regular Expressions

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
```

```
<#
#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
#>
```

Configuration using XML

#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"}

#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
```

Windows Registry

```
#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
```

Processes, Services and Event Log Management

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
```

WMI Management

```
#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
```

Remote Execution

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

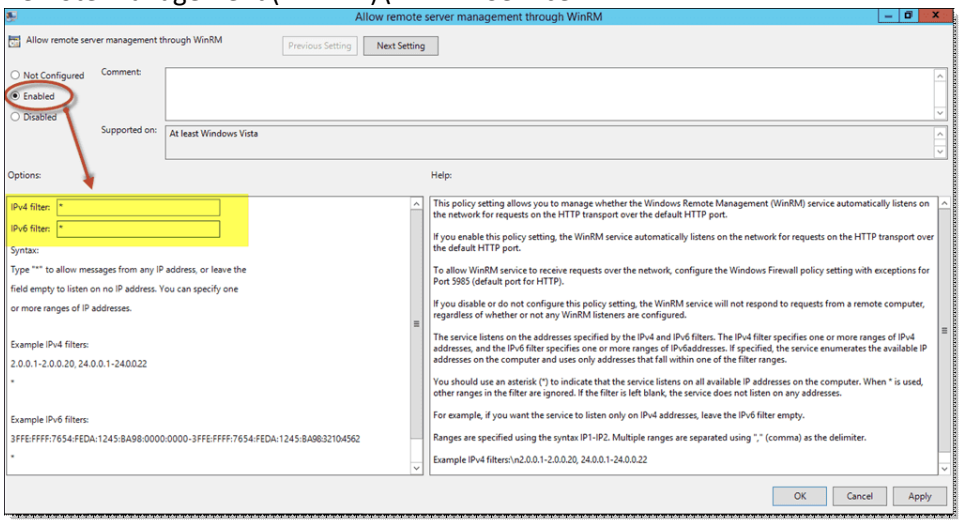
```
Enable-PSRemoting –Force
```

listing all remoting commands

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

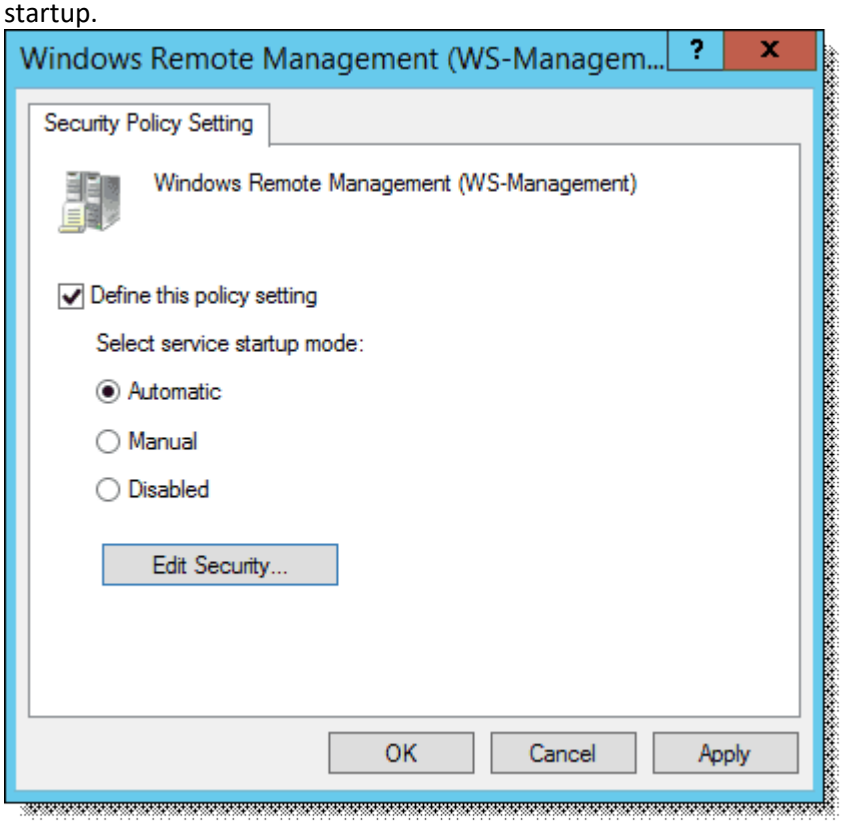
Remote server management through WinRM:

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



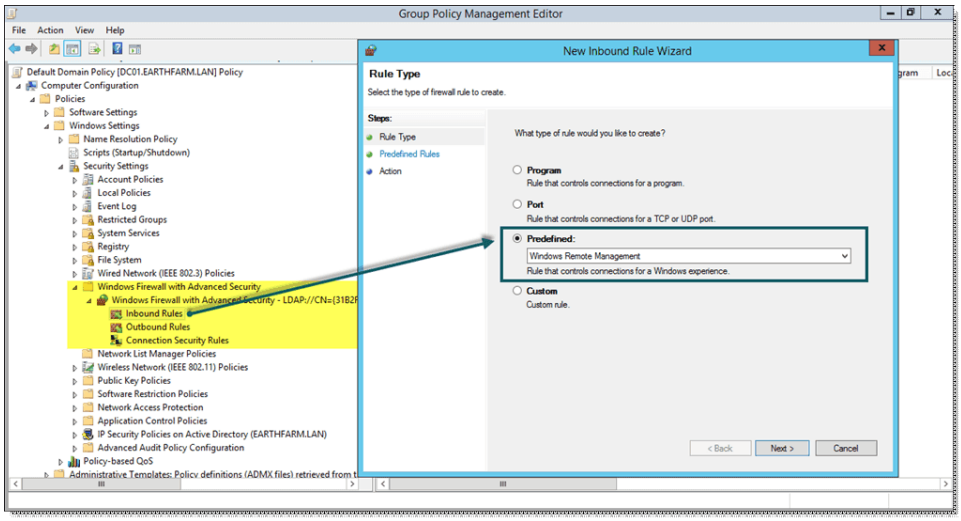
Set WinRM service to automatic startup:
Computer Configuration\Policies\Windows Settings\Security Settings\System Services

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
```

Workflow

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
```

Desired State Configuration (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.

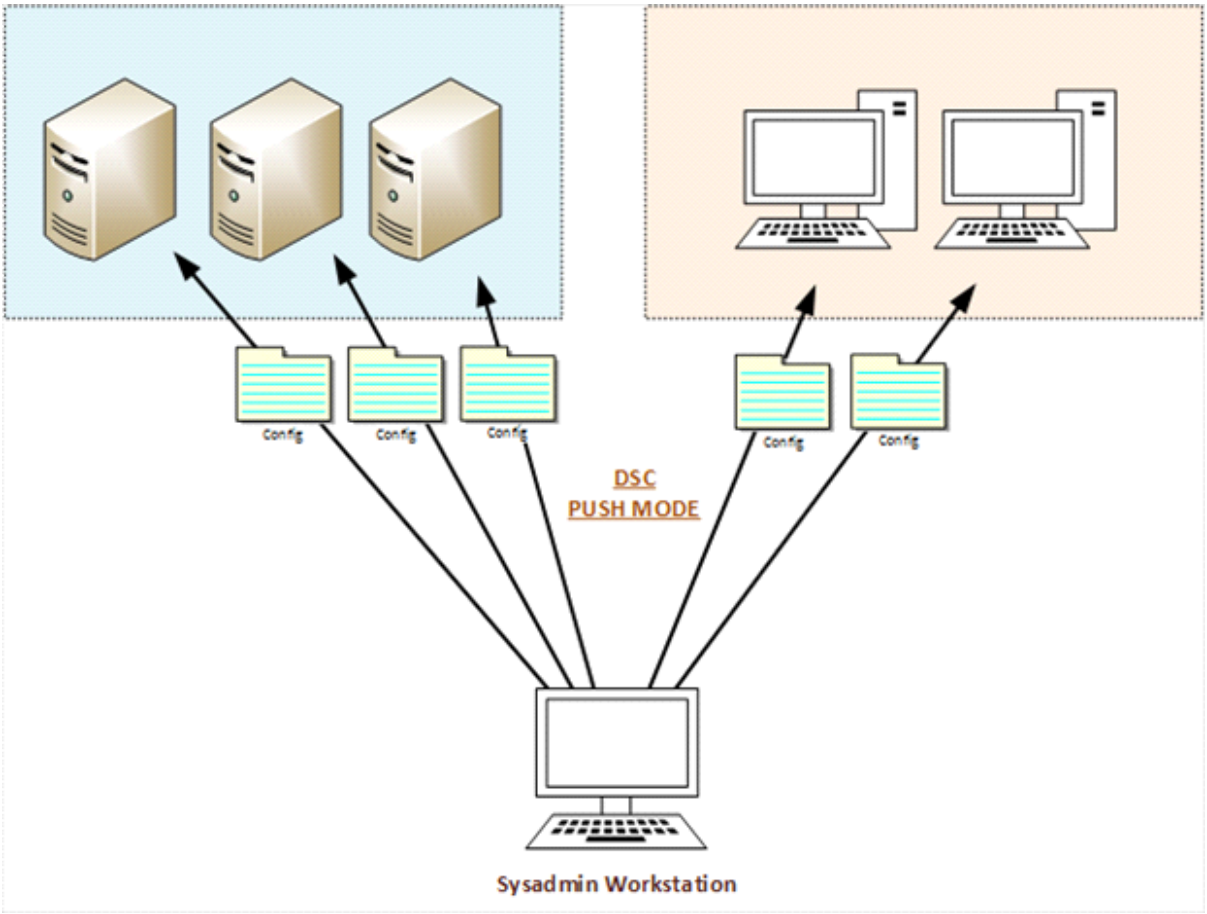
The main advantages of DSC are:

- 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

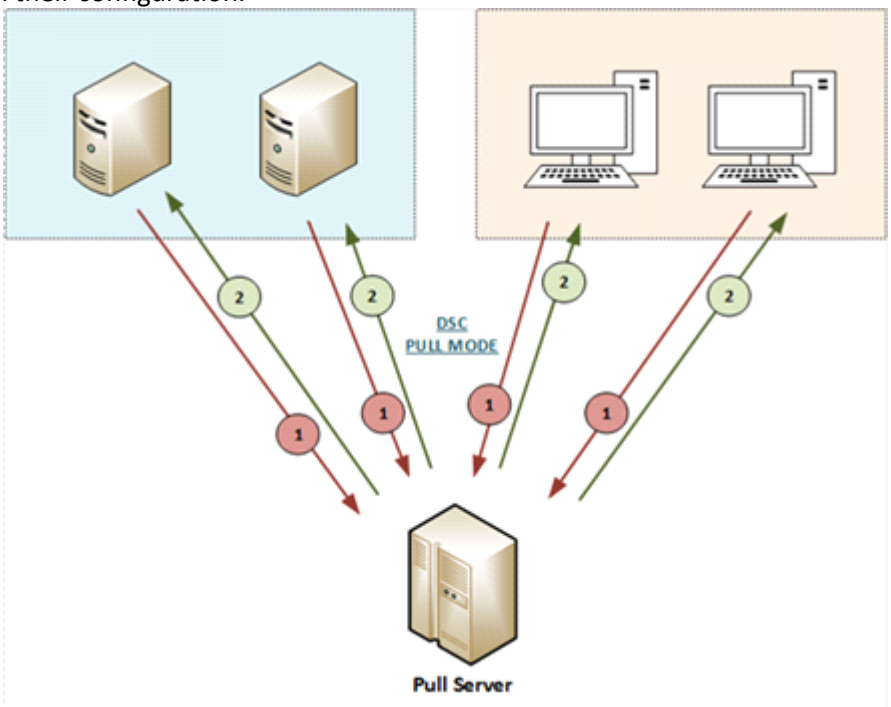
Desired State Configuration is integrated to Framework 4.0, which means that the compatible platforms with DSC are:

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
```

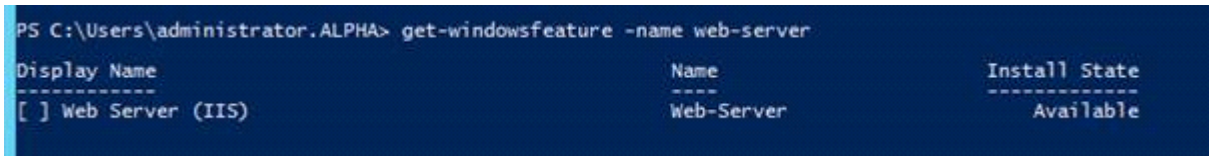
```
Start-DscConfiguration -Path C:\day5\bits-service -Wait -Verbose

#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



Syntax:

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

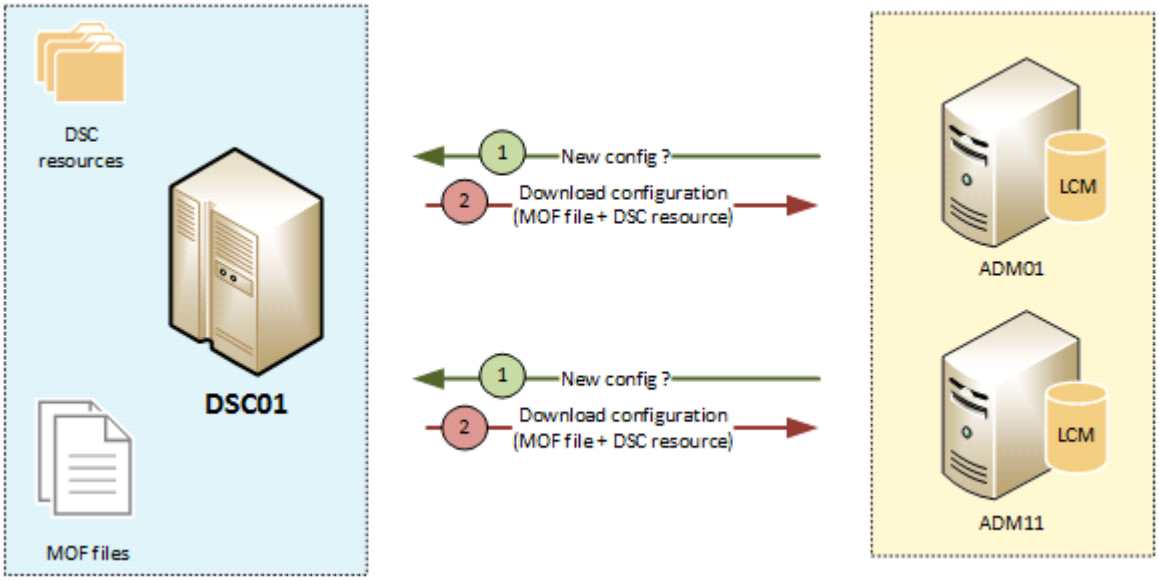
DSC: enabling LCM config on target

```
#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

#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
```