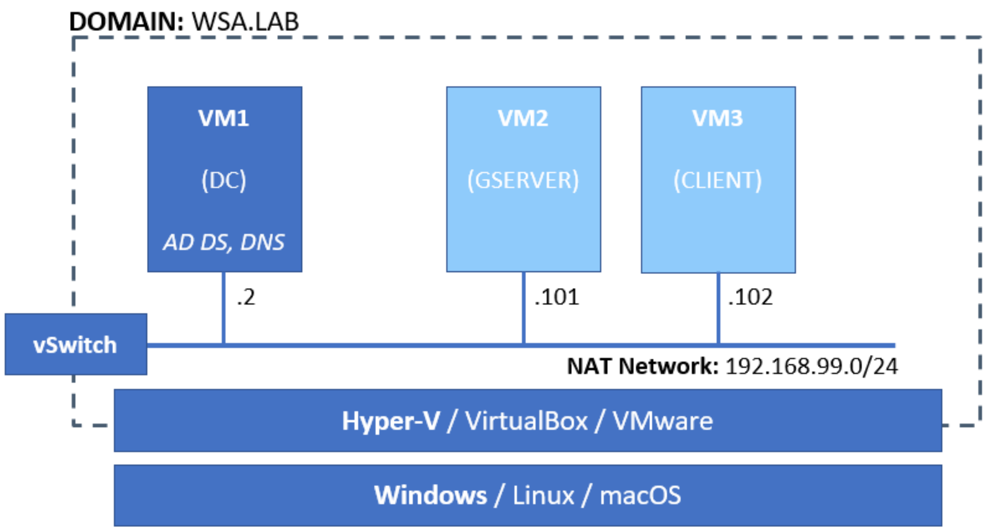
Practice M7: PowerShell

For this lab, we can continue with the infrastructure from the last module – Module 6. In fact, we need just one machine, which can be any of the machines with GUI, used so far.

The following tasks are executed on the **Domain Controller** (**DC**):



Part 1: PowerShell Console

Getting Help

In order to get a list of all possible command actions, we can execute

```
Get-Verb
```

If we are looking for a set of commands related to specific area, we can do

```
Get-Command *User*
```

Of course, we can narrow the search even further with

```
Get-Command -Noun ADUser
```

Now as we have found the command that we are interested in, we can execute

```
Get-Help Get-ADUser
```

If we are not satisfied by the quantity of the information returned, we can update the help. First time we ask for help, and if there is no help installed, we will be asked if we want to download and install it. If we missed somehow this, we could always do it manually.

No matter is it the first time, when we add new modules, or just in any case from time to time, in order to get the latest help, we should execute:

```
Update-Help
```

If the update process stops because of errors, then we can change the command to:

```
Update-Help -Verbose -Force -ErrorAction SilentlyContinue
```

Or we can omit the **-Verbose** switch if we do not want to see detailed information.

To get detailed information for a cmdlet, we must execute:

```
Get-Help Get-ADUser -Full
```

For just the examples, we should execute:

```
Get-Help Get-ADUser -Examples
```

And finally, to get everything in a nice and comfortable way:

```
Get-Help Get-ADUser -ShowWindow
```

Just as a reminder, there is a nicer way to get familiar with a command:

```
Show-Command Get-ADUser
```

Or we can browse all commands with just:

```
Show-Command
```

NOTE: The above is working only on a machine with GUI installed, it won't work on a core installation.

Explore Objects

By now, we should know that when piping commands, the artefacts transferred on the pipe are objects, not text data. There is a way to examine what properties and methods the returned objects possess.

Now, let's execute the following to examine a service:

```
Get-Service MpsSvc
```

Okay, we see only a limited information. We can see more by executing this one:

```
Get-Service MpsSvc | Select *
```

We can go even further with a special command. First, let's see what the command does:

```
Get-Help Get-Member
```

Now, let's pipe our service through it:

```
Get-Service MpsSvc | Get-Member
```

From the output, we can see that two of the properties are aliases to real properties, and that there are some methods. Let's execute one of them:

```
(Get-Service MpsSvc).GetType()
```

Aliases

Now let's see all available aliases with

```
Get-Alias
```

Because the list is long, we can pause it one screen at a time with

```
Get-Alias | More
```

Scroll the results row-by-row by pressing the **Enter** key, or page-by-page with **Space** key, and finally quit with **Q** key.

Let's export the definition of **gcm** alias by executing

```
Export-Alias gcm -As Script -Path C:\Temp\gcm.ps1
```

Now let's see what is in the file

```
Get-Content C:\Temp\gcm.ps1
```

The same way, we can export all aliases if we omit the alias with:

```
Export-Alias -Path C:\Temp\all.ps1 -As Script
```

Now that we know how an alias is defined let's create our own. For example, we can create **np** alias to start notepad

```
New-Alias -Name "np" -Description "Notepad shortcut" -Value "C:\Windows\notepad.exe"
```

Common Commands

Let's see the content of **C:\Windows** with either

```
dir C:\Windows
```

or

```
ls C:\Windows
```

Now let's go to **C:\**

```
cd C:\
```

And create new folder, for example **C:\Temp\Demo** with

```
md C:\Temp\Demo
```

Now go to **C:\Windows\System32\drivers\etc** and examine content of hosts file with either

```
cat hosts
```

Or

```
type hosts
```

Environment Variables

Open classic command shell, type set and then hit Enter. Now you can see all environment variables. Let's pickup one or two and see how we can use them in PowerShell. Return to PowerShell and enter:

```
$env:USERNAME
```

And then

```
$env:COMPUTERNAME
```

Let's try with another one:

```
$env:PATH
```

Okay, let's go one step further, and try few more things:

- In order to get the length of the variable, we can execute:
  `$env:PATH.Length`
- If we want to see the result in a different way, for example each folder on a separate line:
  `$env:PATH.Split(';')`
- And if we want to retrieve the 3-rd component of the **PATH** variable:
  `$env:PATH.split(';')[3]`

PowerShell environment also has its own variables. We can see them by issuing:

```
Get-Variable
```

Or **gv** for short.

We can limit the scope of variables that we are looking for by

```
Get-Variable -Scope Local
```

Of course, we can ask for particular variable with

```
Get-Variable -Name PSHOME
```

Now let's define our own variable, for example

```
$MyVariable = "Demo"
```

And let's get its value either by typing its name

```
$MyVariable
```

Or by executing

```
Get-Variable MyVariable
```

Now let's execute the following, which will execute the block commands in a separate scope, in order to examine how the scope is working:

```
& {Write-Host "Before (existing): $MyVariable"; $MyVariable="Var Demo"; Write-Host "During (local): $MyVariable"; Clear-Variable -Name MyVariable; Write-Host "After (local): $MyVariable";}; Write-Host "After (existing): $MyVariable"
```

Export to Grid and Files

Let's find all users that are part of **Managed Users OU**

```
Get-ADUser -Filter * -SearchBase "OU=Managed Users,DC=WSA,DC=LAB"
```

Now we can modify the output by narrowing the amount of information we are interested in

```
Get-ADUser -Filter * -SearchBase "OU=Managed Users,DC=WSA,DC=LAB" | Select GivenName, Surname, Name, SamAccountName
```

It would be more comfortable the result, especially if it was long enough, to be consumed in a visual way

```
Get-ADUser -Filter * -SearchBase "OU=Managed Users,DC=WSA,DC=LAB" | Select GivenName, Surname, Name, SamAccountName | Out-GridView
```

Here we can sort and filter.

The same result can be stored either in **CSV**

```
Get-ADUser -Filter * -SearchBase "OU=Managed Users,DC=WSA,DC=LAB" | Select GivenName, Surname, Name, SamAccountName | Export-CSV -Path C:\Temp\users.csv
```

Or in **plain-text** file

```
Get-ADUser -Filter * -SearchBase "OU=Managed Users,DC=WSA,DC=LAB" | Select GivenName, Surname, Name, SamAccountName > C:\Temp\users_tee.txt
```

Why not on **screen** and in a **file** at the **same time**

```
Get-ADUser -Filter * -SearchBase "OU=Managed Users,DC=WSA,DC=LAB" | Select GivenName, Surname, Name, SamAccountName | Tee -FilePath C:\Temp\users.txt
```

Files and Folders

We can examine the **C:\Temp** folder with

```
Get-Item C:\Temp
```

Or if we pipe the command through

```
Get-Item C:\Temp | Select *
```

To get extended information.

Now we can get **C:\Temp** content by either

```
Get-Item C:\Temp\*
```

Or by

```
Get-ChildItem C:\Temp
```

Let's create an empty folder called **Test** under **C:\Temp** and then an empty file **Readme.txt** there

```
New-Item -Path C:\Temp\Test -Type Directory
New-Item -Path C:\Temp\Test\Readme.txt -Type File
```

The **-Type File** option can be omitted, and the result will be the same.

Now let's copy a file in **C:\Temp\Test**

```
Copy-Item -Path C:\Windows\System32\drivers\etc\hosts -Destination C:\Temp\Test\hosts.txt
```

Take a moment to examine the file

```
Get-Content C:\Temp\Test\hosts.txt
```

Now empty the file with

```
Clear-Content C:\Temp\Test\hosts.txt
```

And finally remove **C:\Temp\Test** folder and its content with

```
Remove-Item C:\Temp\Test -Recurse
```

Now remove all, but **ps1** files from **C:\Temp**

```
Remove-Item -Path C:\Temp\*.* -Exclude *.ps1 -Confirm
```

PowerShell Profile

Detailed information can be found here: https://docs.microsoft.com/en-us/previous-versions//bb613488(v=vs.85)

And here: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_profiles?view=powershell-6

In general, if we want to initialize our environment in a certain way, for example to have some variables set, or some aliases there, or something else, we must set it in our profile.

Where is our profile on first place? Good question. The path to it is held in the **$PROFILE** variable. We can check if our profile is initialized and what it contains with:

```
Get-Content $PROFILE
```

Remember that this is our profile and it is host (PowerShell, PowerShell ISE, VSCode, …) dependent.

If the profile file does not exist, we can create it, and put something simple there. The easiest way to do it is to execute:

```
notepad $PROFILE
```

And enter some greeting message, for example:

```
Write-Host "Hello, PowerShell Master!"
```

Save and close the file. Now if we close current PowerShell session and re-open it again, we will see the greeting.

Part 2: Script Building Blocks and Tools

Main Development Tool

Start **PowerShell ISE** and examine its interface. Execute a simple command.

Building Blocks

As first step we can test how we can work with variables. Enter the following:

```
$MyFavoriteColor = "Blue"
Write-Host "My favorite color is $MyFavoriteColor"
```

Now let's add if statement:

```
$YourFavoriteColor = "Red"
Write-Host "Your favorite color is $YourFavoriteColor"
If ($MyFavoriteColor -EQ $YourFavoriteColor) {
    Write-Host "Wow, we like one and the same color!"
}
Else {
    Write-Host "No, we like different colors"
}
```

Let's test how the for loop works. Enter and execute the following code:

```
For ($Counter=1; $Counter -LE 5; $Counter++) {
    Write-Host "Iterration no.$Counter"
}
```

While loop would look like:

```
$Counter = 1
Do {
    Write-Host "Iterration no.$Counter"
    $Counter++
} While ($Counter -LE 5)
```

The same block very easily can be turned in until loop:

```
$Counter = 1
Do {
    Write-Host "Iterration no.$Counter"
    $Counter++
} Until ($Counter -GT 5)
```

Simple Interactive Script

Let's create a very simple interactive greeting script:

```
$CharacterName = (Get-Random -InputObject "John", "Jane", "Jim")
Write-Host "Hello there! My name is $CharacterName."
$PlayerName = (Read-Host -Prompt "And what's yours")
Write-Host "Oh, $PlayerName! What a nice name! Nice to meet you."
```

Number Guessing Game

Now let's create new more sophisticated interactive script. We will create a version of the popular number guessing game. One possible solution is:

```
#
```

```
# Guess Game
#

# Initialize variables
$MinNo = 1
$MaxNo = 100
$GuessNum = (Get-Random -Minimum $MinNo -Maximum $MaxNo)

Write-Host "I have a number between $MinNo and $MaxNo on my mind. Could you guess it?"
Write-Host ""

# Ask the opponent
$MaxTries = (Read-Host -Prompt "How many attempts would you like to have")
Write-Host ""

# Initialize the guess attempt counter
$GuessCnt = 1

# Start the guess loop
Do {
    $Guess = (Read-Host -Prompt "Your guess attempt no.$GuessCnt is")

    Write-Host "You think that my number is $( $Guess )? Let me see ..."

    if ($Guess -EQ $GuessNum) {
        Write-Host "Congratulations! You guessed it!"
        Write-Host "My number is $GuessNum"
        Break
    }

    if ($Guess -LT $GuessNum) {
        Write-Host "No. Try a bigger number."
    }

    if ($Guess -GT $GuessNum) {
        Write-Host "No. Try a smaller number."
    }
    Write-Host ""

    $GuessCnt++;
} While ($GuessCnt -LE $MaxTries)

if ($GuessCnt -GT $MaxTries) {
    Write-Host "My friend, you ran out of attempts ..."
    Write-Host "GAME OVER. My number was: $GuessNum"
}
```

Now save the file in **C:\Temp\Number-Guessing.ps1**

Execute the script either from the **PowerShell ISE** tool by hitting the **F5** key or choosing the **File** and then **Run**. Another option would be to open a PowerShell session, and execute:

```
C:\Temp\Number-Guessing.ps1
```

Part 3: Create Complete Scripts

Ranges and Lists

Create a simple range for the numbers between 1 and 10

```
1..10
```

We can extend the range

```
-10..10
```

Now let's filter only even values

```
-10..10 | Where-Object {$_ % 2 -eq 0}
```

Let's define an array

```
"one", "two", "three"
```

If we surround it in parentheses (to make an array), we can invoke a method, for example

```
("one", "two", "three").ToUpper()
```

ForEach-Object

Let's print all IP addresses in a subnet

```
1..254 | ForEach {"192.168.99.$_"}
```

Now let's try the other approach

```
$nodes = (1..254)
ForEach ($node in $nodes) {"192.168.99.$node"}
```

We could eventually create several folders in an automated fashion

```
1..10 | ForEach {New-Item -Path "C:\Temp\DIR$_" -Type Directory}
```

Combine ForEach-Object and Get-Content

Let's create a simple text file, for example **C:\Temp\folders.txt** and its content to be:

```
folder1
folder2
folder3
```

Now, let's ask for the contents of the file:

```
Get-Content C:\Temp\folders.txt
```

Okay, let's combine it with **ForEach-Object** and print each line:

```
Get-Content C:\Temp\folders.txt | foreach {"Line read from file is: " + $_}
```

It is not very exciting, but it is working. Now, let's extend it to create folder for each line read:

```
Get-Content C:\Temp\folders.txt | foreach {New-Item -Name $_ -Path C:\Temp -ItemType Directory
-Force}
```

We can go even further. Let's create new file **C:\Temp\objects.txt** with the following content:

```
object1,File
object2,Directory
object3,File
```

Now try to do the usual simple parse:

```
Get-Content C:\Temp\objects.txt | foreach {$_}
```

Okay, again nothing exciting. But let's use one of the methods of the returned object:

```
Get-Content C:\Temp\objects.txt | foreach {$_.ToString().Split(',')}
```

It seems that we are getting closer. Let's be clear about the iterations. In order to distinguish the strings in each iteration, let's execute this:

```
Get-Content C:\Temp\objects.txt | foreach {$_.ToString().Split(','); Write-Host "-----"}
```

Now, it seems that we have something like an array of strings on each iteration, so let's capture them in a variable:

```
Get-Content C:\Temp\objects.txt | foreach {$o=$_.ToString().Split(',')}
```

Nothing on the screen. Now, let's print the second part of each line read:

```
Get-Content C:\Temp\objects.txt | foreach {$o=$_.ToString().Split(','); $o[1]}
```

We can go forward and finish our one-liner. The final version that will read the file and will create what it dictates will look like:

```
Get-Content C:\Temp\objects.txt | foreach {$o=$_.ToString().Split(','); New-Item -Name $o[0] -
Path C:\Temp -ItemType $o[1]}
```

Create Get-HostStatus Function

Our aim is to create a ping-like script that we will be used by our colleagues to test if a range of hosts is up and running.

Let's start by finding an existing suitable cmdlet. After some research we came up with Test-Connection, so it will be the basis of our script.

First let's see it in action

```
Test-Connection DC
```

Eventually we can ping more than one machine with

```
Test-Connection -ComputerName DC, IIS-CORE, SQL-CORE
```

Let's see if it will work with IP addresses

```
Test-Connection 192.168.99.1
```

Now let's put some automation

```
"192.168.99.1", "192.168.99.2" | ForEach {Test-Connection $_}
```

It will be easier if maintain only the host portion of the address, so

```
1..5 | ForEach {Test-Connection "192.168.99.$_"}
```

We do not need to ping each host multiple times so, let's change the command to

```
1..5 | ForEach {Test-Connection "192.168.99.$_" -Count 1}
```

We are ready to move to PowerShell ISE in order to continue with our script. Create new file and paste the command prepared so far. Save it as **C:\Scripts\HostStatus.ps1** and execute it.

It would be nice to move some parts to variables. Let's create three variables

```
$subnet = "192.168.99."
$start = 1
$end = 5
```

Now let's change the command to

```
$start..$end | ForEach {Test-Connection "$subnet$_" -Count 1}
```

Save it and give it a try.

Okay we do not like to have so many error messages if the hosts are not responding. Modify the command to

```
$start..$end | ForEach {Test-Connection "$subnet$_" -Count 1 -Quiet}
```

Save it and give it a try.

Okay now we would like to have only live hosts. So we change the command to

```
$start..$end | Where {Test-Connection "$subnet$_" -Count 1 -Quiet}
```

Save it and give it a try.

Now we would like to see the successfully pinged IP address. So we will modify it to

```
$start..$end | Where {Test-Connection "$subnet$_" -Count 1 -Quiet} | ForEach {"$subnet$_"}
```

Save it and give it a try.

So far, so good, but it would be nice to have an option not to change every time the file, but to use parameters. So let's change the variables part to

```
Param (
    $SubNet = "192.168.99.",
    $Start = 1,
    $End = 5
)
```

Save it and give it a try.

Now execute it with different end value

```
.\HostStatus.ps1 -End 2
```

Okay, according to good practices we should set variables data type. So the parameters section will become

```
Param (
    [string]$SubNet = "192.168.99.",
    [int]$Start = 1,
    [int]$End = 5
)
```

Save it and give it a try.

Now execute it with different Subnet value, but use for example Sofia

```
.\HostStatus.ps1 -SubNet Sofia
```

We are ready to transform our script to a function. Now the script should look like

```
Function Get-HostStatus {
    Param (
        [string]$SubNet = "192.168.99.",
        [int]$Start = 1,
        [int]$End = 5
    )

    $start..$end | Where {Test-Connection "$subnet$_" -Count 1 -Quiet} | ForEach {"$subnet$_"}
}
```

Save it and source it.

Start typing its name and execute it.

Now try

```
Get-Help Get-HostStatus
```

Let's extend it a little bit. We will add set of so-called common parameters. Add the following just above the parameters section

```
[cmdletbinding()]
```

Save it and source it. Type its name and then put dash and hit several times the Tab key.

Now let's add verbose output. Add the following line just before the actual body

```
Write-Verbose "Pinging $SubNet from $Start to $End"
```

Save it and source it. Then execute it with -Verbose as parameter.

As last step we can add some comments and help related instructions. The final script should look like

```
Function Get-HostStatus {
<#

.SYNOPSIS
Simple host pinging utility
.DESCRIPTION

This function accepts three parameters $SubNet, $Start, and $End


.EXAMPLE

Get-HostStatus -SubNet "192.168.99." -Start 1 -End 100


.NOTES

First source the HostStatus.ps1 file. Keep in mind that if the function is executed without
parameters there are some default values.


.LINK

https://www.google.com


#>


    [cmdletbinding()]

    Param (

        [string]$SubNet = "192.168.99.",

        [int]$Start = 1,

        [int]$End = 5

    )


    Write-Verbose "Pinging $SubNet from $Start to $End"

    $start..$end | Where {Test-Connection "$subnet$_" -Count 1 -Quiet} | ForEach {"$subnet$_"}

}
```

Now we can source it and then get the help to see the final result. If we want to have this function available in every PowerShell session, we must include it or source it in our profile.