PowerShell Scripting

What is Windows PowerShell?

Windows PowerShell is a command-line shell and scripting language designed for system administration and automation tasks. It allows users to interact with the operating system, run scripts, and automate complex administrative tasks.

Windows PowerShell architecture

1. PowerShell Console/Host

The PowerShell host is the user interface through which you interact with the PowerShell environment. It can be:

PowerShell Console: The traditional command-line interface.

PowerShell ISE (Integrated Scripting Environment): A more feature-rich environment for writing and testing scripts.

PowerShell Remoting Host: This is used when you need to run PowerShell commands on remote computers.

The host provides an interface where you can type commands (cmdlets) or execute scripts, and receive output from them. It can be customized and extended.

2. Cmdlets

Cmdlets (pronounced command-lets) are lightweight commands used in PowerShell. They are the building blocks of PowerShell's functionality. Cmdlets follow a verb-noun naming pattern (like Get-Process, Set-Date, New-Item), and are written in .NET languages like C#.

Cmdlets are built-in commands that perform specific tasks such as file management, system administration, and process control.

Cmdlets return .NET objects (instead of just text), which gives PowerShell more power and flexibility than traditional command-line interfaces.

3. Pipeline

PowerShell allows for the use of pipelines (|), which are used to pass the output of one cmdlet to the input of another cmdlet. The data passed through the pipeline is not just text but objects, allowing subsequent cmdlets to manipulate those objects directly.

The pipeline is a fundamental part of PowerShell's architecture and enables users to build complex command sequences.

4. PowerShell Runtime

The PowerShell runtime is built on the .NET Framework or .NET Core, depending on the version you're using (Windows PowerShell vs. PowerShell Core).

PowerShell Core (starting with version 6) is cross-platform and runs on .NET Core (which is open-source and can run on Windows, macOS, and Linux).

Windows PowerShell (version 5.1 and below) is built on the full .NET Framework, which is Windows-specific.

The runtime provides access to the Common Language Runtime (CLR) and allows PowerShell to interact with the underlying operating system, manage resources, and perform tasks using .NET libraries.

5. Execution Engine

The PowerShell Execution Engine is responsible for running scripts and commands. It handles parsing and interpreting the commands that are written in PowerShell's scripting language.

The engine works as follows:

- Parser: It reads and parses the commands into a format the engine can understand.
- Compiler: The parsed commands are compiled into intermediate code.
- Executor: The executor executes the compiled code.
- The Execution Engine can execute commands interactively (one at a time) or as part of a script or function.

6. Types and Providers

Types: In PowerShell, types represent .NET classes, and PowerShell can access .NET types, methods, and properties. Users can define custom types (like .NET objects) in scripts or functions.

Providers: Providers allow PowerShell to interact with different data stores and systems (like file systems, registry, Active Directory, etc.). A provider maps a data store (such as the file system or the registry) into a drive, making it accessible using a standard cmdlet syntax. For example:

Get-ChildItem can be used to list files on your file system (C:\), or keys in the registry (HKCU:), if the appropriate provider is in place.

7. Modules

Modules are packages that contain cmdlets, functions, variables, and other resources. They allow users to extend PowerShell's capabilities. For example:

Modules can be used to manage specific services (like Active Directory or SQL Server).

You can import and use third-party or custom modules to add functionality.

Modules help with code organization and reusability.

8. Runspaces

A runspace is an execution environment in PowerShell. It holds the context for executing commands, such as variables, aliases, and functions.

By default, PowerShell uses a single runspace for executing commands in the console.

Multiple runspaces can be used in parallel for executing commands in different threads (e.g., for remoting or background jobs).

9. Remoting

PowerShell remoting allows you to run commands on remote machines. It uses the WS-Management protocol, which is a standard web services protocol for managing devices.

PowerShell remoting enables:

- Running commands and scripts on remote machines.

- Managing multiple systems simultaneously, which is essential for administrators in large environments.

10. Integrated Script Environment (ISE)

PowerShell ISE is an integrated tool for writing, testing, and debugging PowerShell scripts. It has features like:

- Syntax highlighting.
- Intellisense (auto-completion).
- Debugging support for breakpoints and step-by-step execution.

Parameters in PowerShell

In PowerShell, parameters are used to pass values or arguments to scripts, functions, or cmdlets. They allow you to customize behavior and control how the script or function works.

Here's a brief summary of key types of parameters:

- Positional Parameters: Passed in the order defined, without specifying their names.
- Named Parameters: Supplied by specifying the parameter name (e.g., -Name "John").
- Mandatory Parameters: Must be provided when running the script or function.
- Optional Parameters: Have default values if not provided.
- Switch Parameters: Boolean values (either $true or $false), typically used for flags like -Verbose.
- Array Parameters: Allow passing multiple values as an array.
- Parameter Validation: Can include rules like ValidateSet or ValidateRange to control the acceptable values.
- Default Values: Parameters can have default values that are used when none are provided.

Wildcards

Wildcards are something which you can use to match partial values instead of exact values.

The ‘*’ wildcard will match zero or more characters.

The ‘?’ wildcard will match a single character

Pipeline or pipe “|”

Each pipeline operator sends the results of the preceding command to the next command. The

output of the first command can be sent for processing as input to the second command.

If/Then

This the simplest form of decision making in PowerShell. It basically works like this: Something is

compared with something and depending on the comparison do this activity.

The comparison statement must have a logical response of either TRUE or FALSE. Think of it as a

yes or no question.

Loops

Loops are the most important feature that are used to automate/action bulk activities.

ForEach() loop

This is the most used loop in PowerShell and can be used most of the times to serve our

Purpose

For Loop

This loop is used in case you require a counter, ie if you want to run the loop for 'n' number of

times.

```
for($i = 1 ; $i -le 10 ; $i++)
 {
 Write-Host "Current value : "$i -ForegroundColor Green
 }
```

While Loop

This loop also works based on a condition and can be used in scenarios where a counter must be

used or in a situation where a change of state must be monitored.

The while statement runs a statement list zero or more times based on the results of a

conditional test. The syntax of this loop is pretty simple.

```
while(condition)
 {
 Code
 }
```

Do While Loop

A ‘Do While’ will execute at least once before validating the condition, even if the condition is

True in the first iteration, the loop will still execute once.

This loop works based on the positive result of the condition.

The syntax of the loop is as below:

```
Do
 {
 Code
 }while(Condition)
```

Functions

```
function “our choice of name”
 {
 1st line in code
 2nd line in code
 etc
 }
```

Variable

Any character that has a symbol ‘$’ in front of it becomes a variable. A variable comes in play

when you need to store a value so that you can make use of it later in the script.eg

You have 2 values ‘10’ and ‘4’ and you must perform 3 functions on it, like addition, subtraction,

multiplication. So, there are 2 ways to perform this task

(a) 10+4, 10-4, 10*4

(b) $a = 10

$b = 4

$a + $b

$a - $b

$a * $b

Variable Types

PowerShell variables are not strictly typed, meaning they can store any type of data, such as strings, integers, arrays, and objects. However, you can specify types explicitly using [type] casting if needed.

Variable Scope

Variables in PowerShell have different scopes. The most common scopes are:

- Global Scope: Available everywhere.
- Local Scope: Available only within the script or function where it is defined.
- Script Scope: Available throughout the script but not globally.
- Private Scope: Available only in the current session.

Using Variables

Variables are used by referencing them with the $ symbol. When you want to use the value stored in a variable, just refer to it by its name.

What is PowerShell Remoting?

PowerShell Remoting is a feature that allows you to run PowerShell commands and scripts on remote computers. It enables managing multiple machines from a single location, either interactively or in batch, over a network. It uses the WS-Management protocol and works over HTTP/HTTPS.

Key commands:

- Enter-PSSession: For interactive remote sessions.
- Invoke-Command: To run commands/scripts on remote computers.
- New-PSSession: To create persistent remote sessions.

Filtering object out of the pipeline

In PowerShell, filtering objects out of the pipeline means removing specific items from the output stream based on a condition. This is typically done using cmdlets like Where-Object, Select-Object, and other filtering techniques.

Here are common ways to filter objects out of the pipeline in PowerShell:

- Where-Object: Filters objects based on a condition.
- Select-Object: Selects specific properties (useful for "excluding" unnecessary properties).
- -NotLike / -NotMatch: Filters objects based on string matching conditions.

- ForEach-Object with if: Filters out objects manually with conditional logic.
- Out-Null: Discards unwanted objects from the pipeline.

Redirecting Formatted Output to the file

>: Redirect output to a file (overwrites the file).

>>: Redirect output to a file (appends to the file).

Out-File: More flexible redirection with additional formatting options (encoding, append, etc.).

Format-Table / Format-List: Used to format the output before redirecting it to a file.

Tee-Object: Allows output to be displayed on the console and saved to a file simultaneously.

Out-String: Forces the output to be treated as a string before saving.

Creating own Module in PowerShell

What is a PowerShell Module?

A PowerShell module is simply a .psm1 file (PowerShell script module) that contains functions, cmdlets, variables, and other resources. It helps in organizing and reusing code effectively.

2. Steps to Create a PowerShell Module

Step 1: Create the Module Folder

Modules are typically stored in a specific folder structure. To create your own module:

Create a folder for the module. The folder name should be the same as the module name.

Inside that folder, create the module file with a .psm1 extension (PowerShell Module).

Step 2: Write the Module Code

Open the .psm1 file in any text editor (e.g., Visual Studio Code, Notepad++) and write the functions or code you want to include in the module.

Step 3: Optional: Create a Module Manifest

A module manifest is a .psd1 file that provides metadata about the module, such as the module version, dependencies, author, etc. It's not strictly required, but it's a good practice to include one.

Step 4: Test Your Module Locally

Before sharing your module or using it in other sessions, test it locally by importing it into your PowerShell session.

Step 5: Make Your Module Available to Other Sessions

To make the module available to all PowerShell sessions, you can copy the module folder to one of the following locations:

For a single user: $HOME\Documents\WindowsPowerShell\Modules\

For all users: C:\Program Files\WindowsPowerShell\Modules\

Step 6: Distribute Your Module (Optional)

If you want to share your module with others or distribute it, you can publish it to the PowerShell Gallery, which is the official repository for PowerShell modules. You would need to create a free account and follow the steps outlined in the PowerShell Gallery documentation for publishing.

To publish a module to the PowerShell Gallery, you can use the Publish-Module cmdlet.

3. Advanced Features in PowerShell Modules

PowerShell modules can also include additional advanced features:

Cmdlets: You can define custom cmdlets in a module using C# and compile them into a .dll file, then import them into the module.

Module Aliases: You can create aliases for module functions or cmdlets.

Private Functions: Functions not exported outside the module can be made private using the Export-ModuleMember cmdlet with the -Function flag.

Example Directory Structure

Your module folder structure might look like this:

```
MyModule\
  ├── MyModule.psm1
  ├── MyModule.psd1   (optional manifest file)
  └── README.md       (optional documentation)
```

5. Importing the Module in PowerShell

Once the module is created and placed in the appropriate directory, you can load and use the module in any PowerShell session by running:

```powershell
Import-Module MyModule
```

To ensure the module is loaded automatically in every session, you can add Import-Module MyModule to your $PROFILE.