**Batch information:**

* **Batch Start Date: 2025-08-04**
* **Batch Name: WiproNGA\_DWS\_B5\_25VID2550**
* **First Name: Seetal**
* **Last Name: Biswal**
* **User Id: 34933**
* **Batch ID: 25VID2550**
* **Topics:** Process check and Delete

Loop based scipting operations

Set-Ini Value

Toolkit Logging location and how its configured

MSI/MSP Logging

individual Command logging within Script

How to catch return Codes with in PowerShell

MSIX Technology Fundamentals with Architecture

**Process check and Delete :** To check if a process is running and delete a file or folder, you can use the following PowerShell commands:

**Check if a process is running:**

Get-Process -Name "ProcessName" -ErrorAction SilentlyContinue

**Delete a file:**

Remove-Item -Path "C:\Path\To\File.txt" -Force

**Delete a folder:**

Remove-Item -Path "C:\Path\To\Folder" -Recurse -Force

**Loop based scipting operations:** Loop-based scripting operations are essential in PowerShell, allowing you to repeat tasks and process data efficiently. Here are some common loop constructs:

**1. Foreach Loop:**

foreach ($item in $collection) { ... }

Used to iterate over a collection of objects, such as an array or a list.

**Example:**

$fruits = @("Apple", "Banana", "Cherry")

foreach ($fruit in $fruits) {

Write-Host "Fruit: $fruit"

}

**2. For Loop:**

for ($i = 0; $i -lt $count; $i++) { ... }

Used to iterate a specified number of times.

**Example:**

for ($i = 0; $i -lt 5; $i++) {

Write-Host "Iteration $i"

}

**3. While Loop:**

while ($condition) { ... }

Used to repeat a block of code while a condition is true.

**Example:**

$i = 0

while ($i -lt 5) {

Write-Host "Iteration $i"

$i++

}

**4. Do-While Loop:**

do { ... } while ($condition)

Used to repeat a block of code while a condition is true, with the condition checked after the code block.

**Example:**

$i = 0

do {

Write-Host "Iteration $i"

$i++

} while ($i -lt 5)

**5. Do-Until Loop:**

do { ... } until ($condition)

Used to repeat a block of code until a condition is true.

**Example:**

$i = 0

do {

Write-Host "Iteration $i"

$i++

} until ($i -ge 5)

**Set-Ini Value:** Set-IniValue is a PowerShell function used to modify values in an INI file.

Set-IniValue allows you to set or update values in an INI file.

**Common Parameters:**

1. Path: Specifies the path to the INI file.

2. Section: Specifies the section in the INI file where the value will be set.

3. Key: Specifies the key for the value to be set.

4. Value: Specifies the value to be set.

**Example:**

Set-IniValue -Path "C:\Path\To\file.ini" -Section "Settings" -Key "Timeout" -Value "30

**Toolkit Logging location and how its configured:** The logging location and configuration for a toolkit can vary depending on the specific toolkit and its settings.

**Logging Location:**

The logging location is typically specified in the toolkit's configuration settings. Common locations for logs include:

1. Log files: A dedicated log file or directory where the toolkit writes log messages.

2. Event logs: The toolkit may write log messages to the Windows Event Log or other system event logs.

3. Console output: Log messages may be displayed in the console or terminal where the toolkit is running.

**Logging Configuration:**

The logging configuration typically includes settings such as:

1. Log level: The level of detail included in the logs, such as Debug, Info, Warning, Error, or Fatal.

2. Log format: The format of the log messages, including the information included in each log entry.

3. Log rotation: Settings for rotating log files, such as the maximum size or age of log files.

4. Log output: Settings for where log messages are written, such as a file, console, or event log.

**MSI/MSP Logging:**

**MSI Logging:**

MSI logging allows you to capture detailed information about the installation process, including:

1. Installation steps: Logs each step of the installation process.

2. Error messages: Captures error messages and codes.

3. Property values: Logs property values used during installation.

**MSP Logging:**

MSP logging is similar to MSI logging, but it's specific to patch installations. It captures information about the patch installation process, including:

1. Patch application: Logs the application of the patch.

2. Error messages: Captures error messages and codes.

3. Patch properties: Logs properties used during patch installation.

**Individual Command logging within Script:** Individual command logging within a script allows you to track the execution and outcome of specific commands. Here's how you can implement it:

**Logging Commands:**

**You can use various logging techniques, such as:**

1. Write-Host: Outputs log messages to the console.

2. Write-Log: A custom function to write log messages to a file.

3. Start-Transcript: Records all console output to a file.

**Example:**

# Define a logging function

function Write-Log {

param (

[string]$Message,

[string]$LogFilePath

)

$timestamp = Get-Date -Format "yyyy-MM-dd HH:mm:ss"

"$timestamp - $Message" | Add-Content -Path $LogFilePath

}

# Set log file path

$logFilePath = "C:\Logs\script.log"

# Log individual commands

Write-Log -Message "Starting script execution" -LogFilePath $logFilePath

try {

# Execute a command

$result = Get-ChildItem -Path "C:\Temp"

Write-Log -Message "Executed Get-ChildItem command successfully" -LogFilePath $logFilePath

} catch {

Write-Log -Message "Error executing Get-ChildItem command: $($Error[0].Message)" -LogFilePath $logFilePath

}

Write-Log -Message "Script execution completed" -LogFilePath $logFilePath

**How to catch return Codes with in PowerShell:** Catching return codes within PowerShell allows you to handle the outcome of commands or scripts and take appropriate actions. Here's how you can do it:

**Using $LASTEXITCODE:**

The $LASTEXITCODE variable stores the exit code of the last external command or executable.

# Run an external command

& "C:\Path\To\Executable.exe"

# Check the exit code

if ($LASTEXITCODE -eq 0) {

Write-Host "Command executed successfully"

} else {

Write-Host "Command failed with exit code $LASTEXITCODE"

}

**Using Try-Catch Blocks:**

Try-catch blocks allow you to catch and handle exceptions raised by PowerShell commands.

try {

# Run a command

Get-ChildItem -Path "C:\NonExistentPath"

} catch {

# Handle the exception

Write-Host "An error occurred: $($Error[0].Message)"

}

**MSIX Technology Fundamentals with Architecture:** MSIX is a packaging format for Windows applications that combines the best features of MSI, ClickOnce, and App-V. Here's an overview of MSIX technology fundamentals and architecture:

**What is MSIX?**

MSIX is a modern packaging format for Windows applications that provides a secure, reliable, and efficient way to distribute and manage applications.

**Key Benefits:**

1. Improved security: MSIX provides robust security features, such as package signing and encryption.

2. Simplified deployment: MSIX simplifies application deployment and management.

3. Better performance: MSIX optimizes application performance and reduces installation time.