**PRACTICE ACTIVITY 8**

**Assignment: Screenshot of the commands ran.**

**Name:** Sajiya

**Email id:** [sajiyadl888@gmail.com](mailto:sajiyadl888@gmail.com)

**User id:** 34764

**1.Get-Process**

In PSAppDeployToolkit, the **Get-Process** cmdlet is used to check running processes by name or ID, allowing filtering to target specific processes, while the **Stop-Process** cmdlet is used to terminate them. You can check if a process is running by retrieving it with -Name or -Id, and stop it by specifying either parameter, optionally suppressing confirmation with -NoConfirm. For example, you might verify if "myProcess.exe" is active and then stop it if found. Stopping processes requires appropriate permissions and should be done cautiously to avoid disrupting system or application functionality. Additionally, PSAppDeployToolkit offers other helpful functions like Execute-Process for launching programs and Get-PendingReboot for checking reboot status.

**$processname = “notepad”**

**If (get-process –name $processname) {**

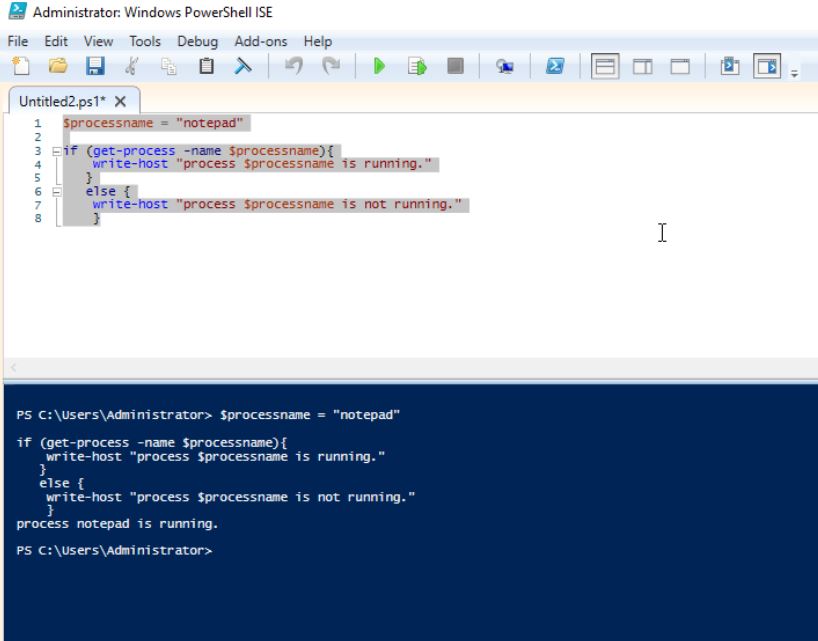
**Write-host “process $processname is running.”**

**}**

**Else {**

**Write-host “process $processname not running.”**

**}**

****

**get-process notepad**

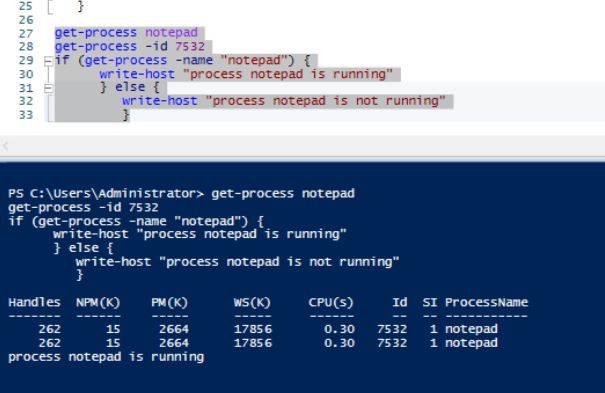
**get-process –id 7532**

**if (get-process –name “notepad”) {**

**write-host “process notepad is running.”**

**} else {**

**Write-host “process notepad is not running.”**

**}**

**2.For-loop**

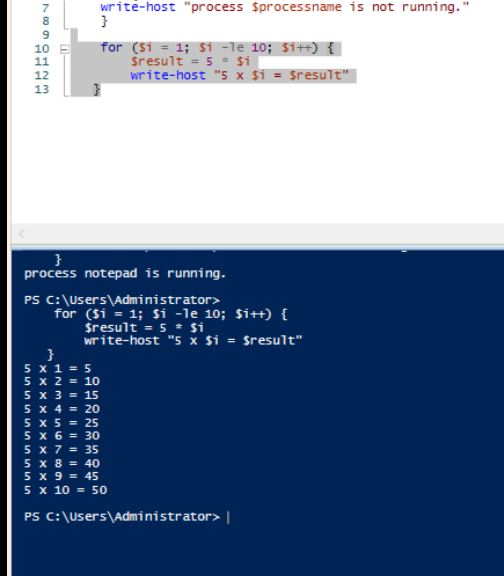
The **for loop** in PowerShell is a control structure used to repeatedly execute a block of code while a specified condition evaluates to $true. It consists of three main parts: **Init** (initialization, often setting a counter variable), **Condition** (the Boolean test that determines whether the loop continues), and **Repeat** (the step executed after each iteration, usually incrementing/decrementing a variable). The loop’s statement block, enclosed in {}, contains the commands to run each cycle. For loops can handle multiple initializations, conditions, and increments, making them more flexible than foreach. They are useful for tasks like iterating through arrays, performing operations on selected elements, renaming files, and generating sequences (e.g., multiplication tables). You can run them indefinitely by omitting the condition or terminate them when the condition becomes $false. Complex scenarios may combine regular expressions, zero-padding, or multiple counters for advanced data processing.

**For ($i = 1 –le 10; $i++) {**

**$result = 5 \* $i**

**Write-host “5 x $i = $result”**

**}**



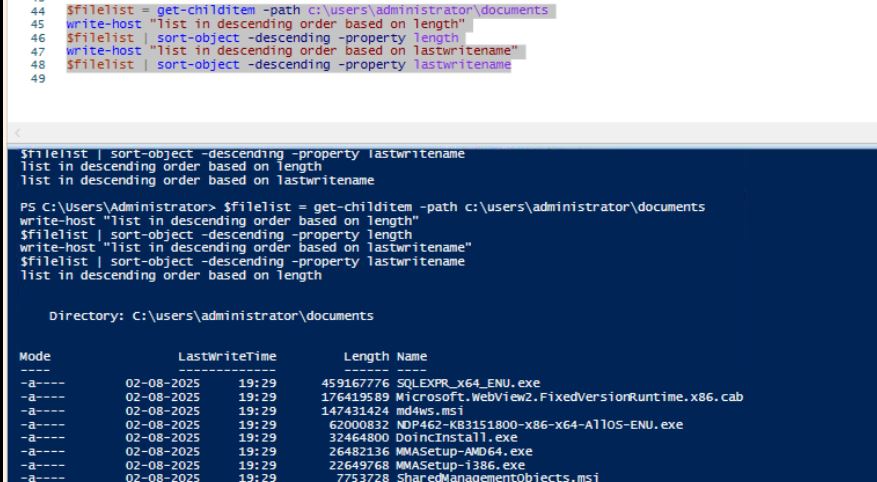
**$filelist = get-childitem –path c:\users\administrator\documents**

**Write-host “list in descending order based on length”**

**$filelist | sor-object –descending –property length**

**Write-host “list in descending order based on writelastname”**

**$filelist | sort-object –descending –property writelastname**

****

**3.Array**

An **array** in PowerShell is a collection that can store multiple values (of the same or different types) in a single variable, making it ideal for situations like a for loop where you want to process a predefined list of items. Arrays are created by enclosing values in @() or simply separating them with commas (e.g., $numbers = 1,2,3). You can access elements by index (starting at 0), loop through them with for or foreach, and even modify them dynamically using operators like + to append values. In the context of loops, arrays provide the data set over which iterations occur, such as looping through filenames, numbers, or configuration values to perform repetitive tasks efficiently.

**$array = @(**

**@(2, 4, 6),**

**@(7, 5, 1),**

**@(3, 8, 9)**

**)**

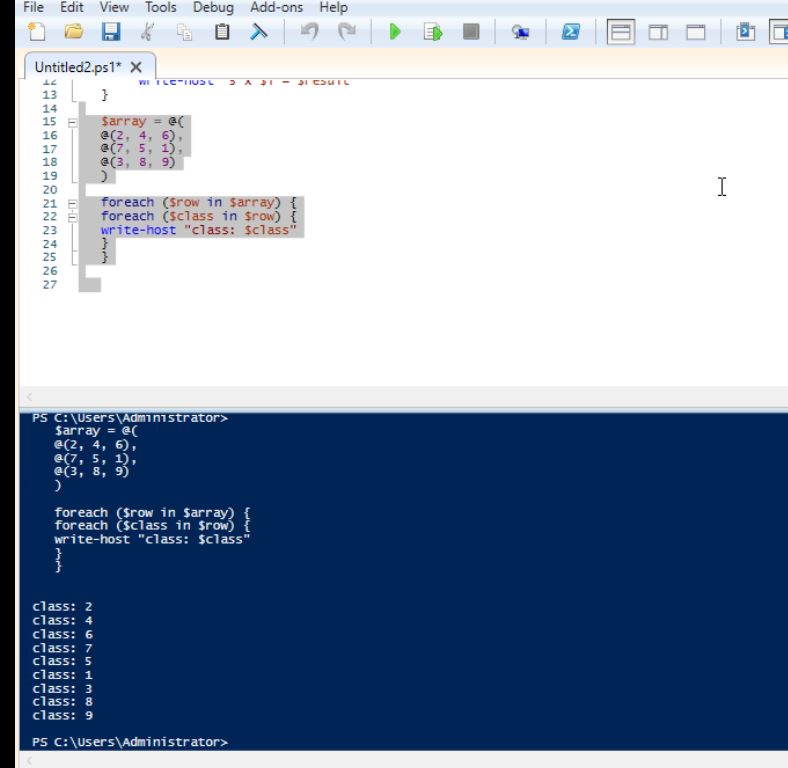
**Foreach ($row in $array) {**

**Foreach ($class in $row) {**

**Write-host “class: $class”**

**}**

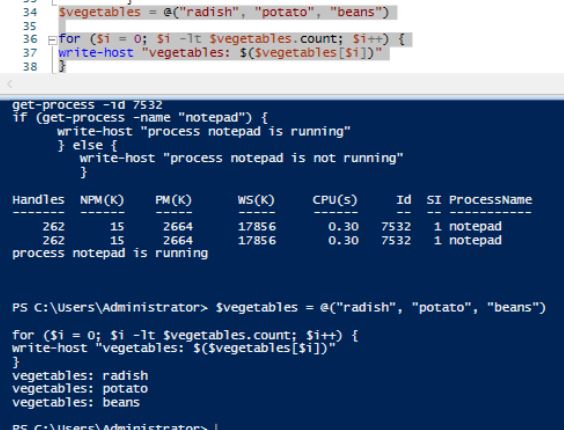
**}**

****

**$vegetables = @(“radish”, “potato”, “beans”)**

**For ($i = 0; -lt $vegetables.count; $i++) {**

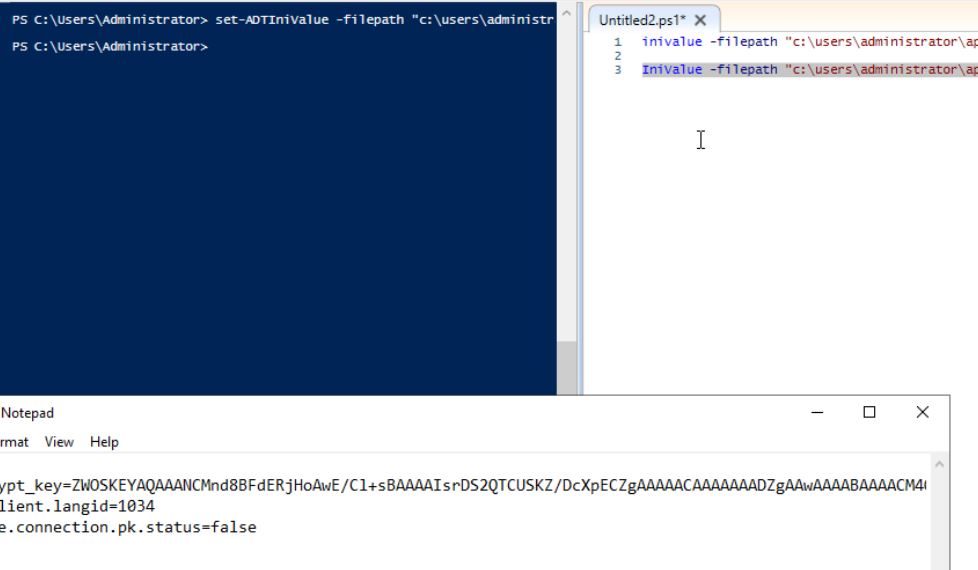
**Write-host “vegetables: $($vegetables[$i])”**

**}**

**4.SetADTInivalue**

The **Set-ADTIniValue** function in PSAppDeployToolkit is used to update or assign a value to a specific key within a given section of an INI file. It requires the file path, section name, key name, and the value to set as parameters. For example, running Set-ADTIniValue -FilePath "C:\Windows\Path\to\my.ini" -Section "MySection" -Key "MyKey" -Value "MyValue" will set MyValue for MyKey in the MySection of the specified INI file.

**Setadtinivalue –filepath “c:\users\administrator\appdata\roaming\zoom\data\zoom.us.ini –-section “zoomchat” -key “com.zoom.client.id –value “1034”**

****

**5. Return codes**

In PowerShell, you can handle errors and check return codes in different ways. For external programs like .exe files, $LASTEXITCODE stores their exit code (0 means success, non-zero means an error). For errors within PowerShell itself, try...catch blocks are used—try holds the risky code, and catch runs if an error happens, often with $ErrorActionPreference = "Stop" so even non-terminating errors can be caught. You can also set custom exit codes using the exit statement to indicate success or failure in your scripts. Overall, $LASTEXITCODE is for external commands, try...catch is for PowerShell error handling, and exit is for returning your own status codes.

**$errorActionPreference = “stop”**

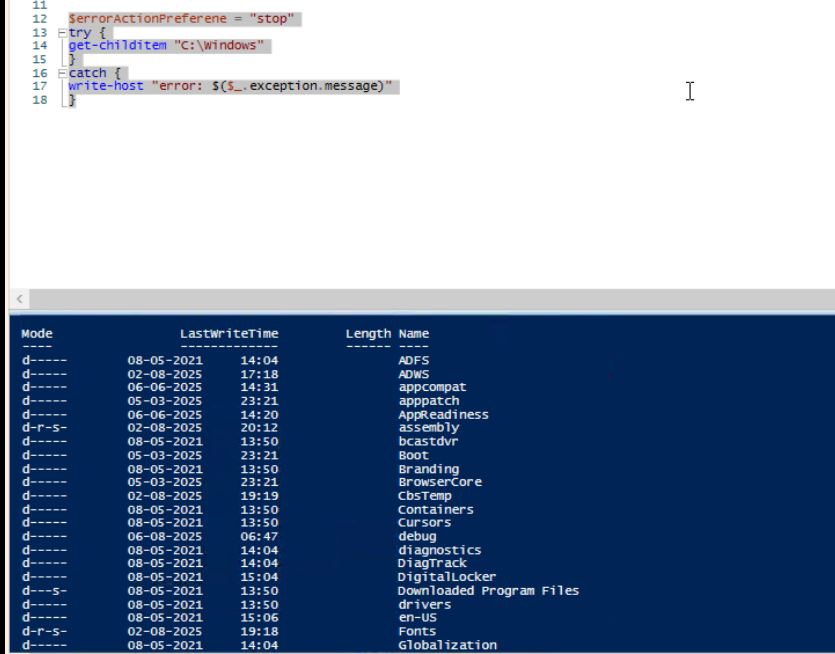
**Try {**

**Get-childitem “c:\windows”**

**}**

**Catch {**

**Write-host “error: $(\_.exception.message)”**

**}**

**Try {**

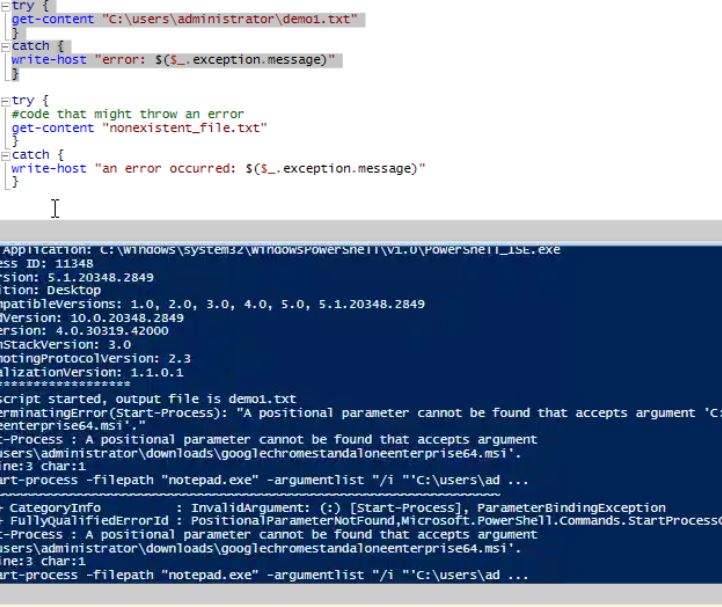
**Get-content “c:\users\administrator\demo1.txt”**

**}**

**Catch {**

**Write-host “error: $($\_.exception message)”**

**}**

****

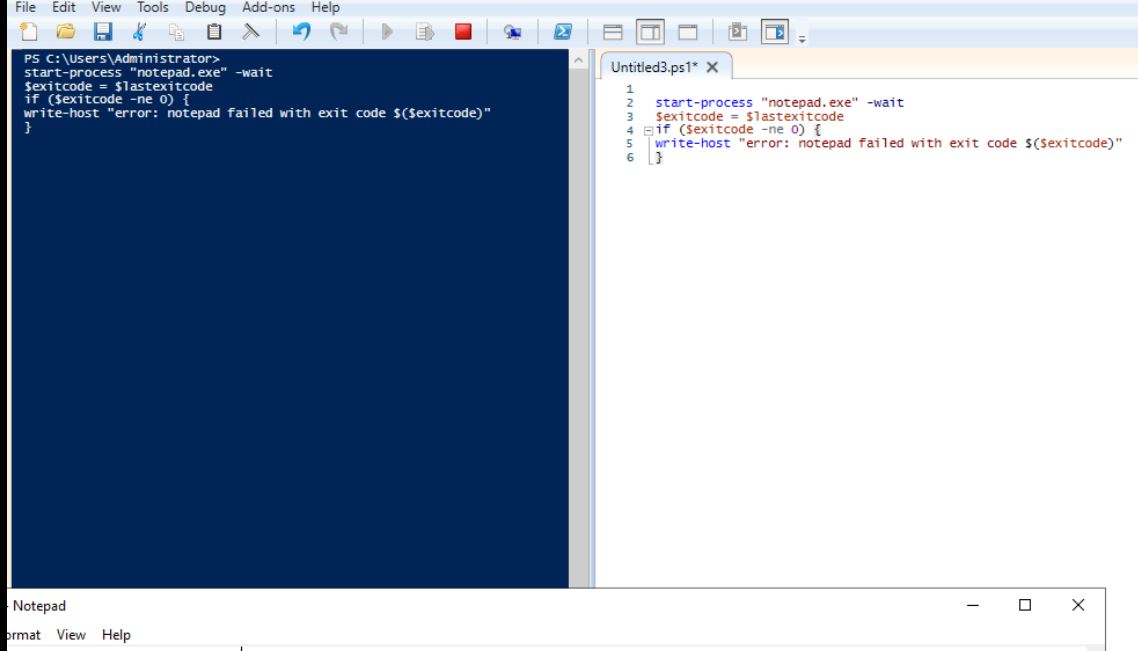
**Start-process “notepad.exe” –wait**

**$exitcode = $lastexitcode**

**If ($lastexitcode –ne 0) {**

**Write-host “error: notepad failed with exit code $lastexitcode”**

**}**

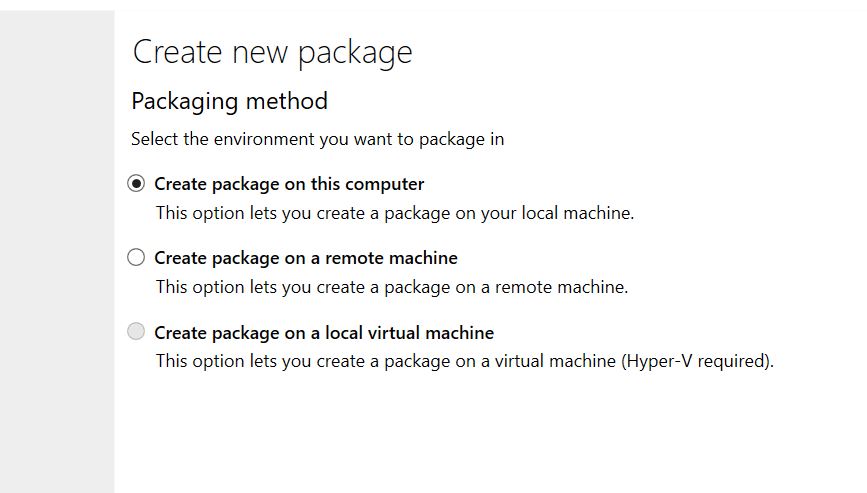
****

**6. MSI/MSP Logging**

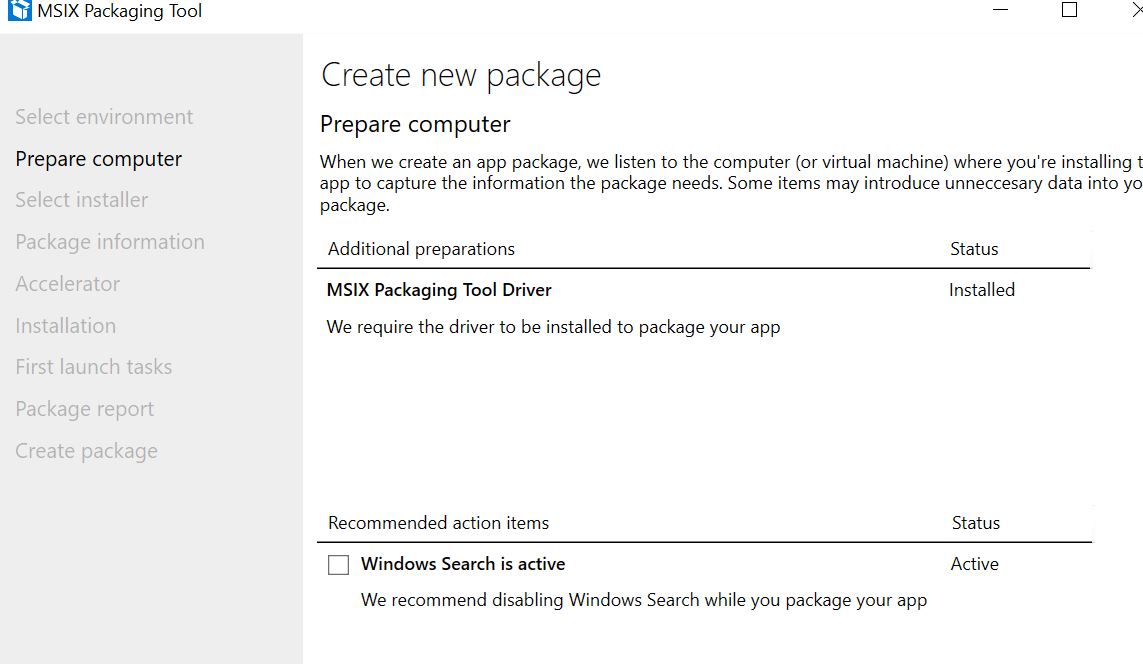
Within the PowerShell App Deployment Toolkit (PSADT), robust logging is a critical feature for managing software installations, and the **Execute-MSI** command is central to this process. This command is not merely an installer; it is a professional tool that ensures deployments are both auditable and troubleshootable. By using the -**LogName parameter**, a system administrator can create a dedicated log file for each deployment, which meticulously records every action, parameter, and outcome, including any errors encountered during the silent installation. This practice is essential for professional IT operations, as it moves deployment from a "black box" process to a transparent and documented event, allowing for swift problem diagnosis and providing a clear, auditable trail for compliance and quality assurance.

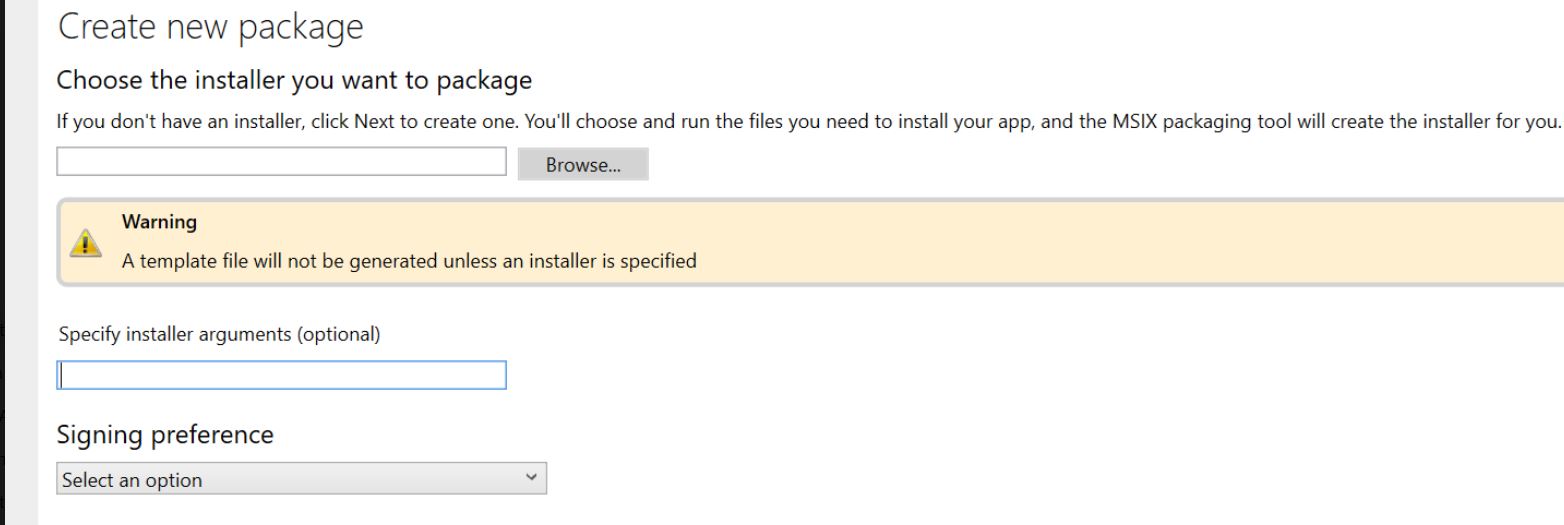
**7. MSIX**

MSIX is a modern Windows app packaging format that improves on older installers like MSI and AppX by supporting both traditional desktop and modern Store apps, offering enhanced security, efficient updates, and cleaner installation/removal. It packages application files, configuration data, and optional resources into a zip-like container, often running apps in isolated environments to improve stability and security. MSIX integrates well with existing deployment tools, supports various distribution methods (like Intune, MECM, or Microsoft Store), and allows advanced features like App Attach for flexible app management. Compared to MSI, MSIX delivers stronger security through signing/validation, more efficient deployment and updates, and better application isolation.

****Step 1

Step 2

****

**Step 3**