Antony Kervazo-Canut

Powershell for Teenagers

THE FAVORITE TERMINAL OF WINDOWS

SOMMAIRE

Introduction

PowerShell is a scripting language and command-line shell developed by Microsoft. Primarily designed for system administration, it enables the automation of administrative tasks, configuration management, and data querying. PowerShell extends the capabilities of traditional batch scripts by allowing the execution of complex commands and manipulation of system objects.

PowerShell offers several advantages for IT professionals and developers:

- Automation: Reduces repetitive tasks by enabling the automation of administrative processes.
- Access to a wide range of commands (Cmdlets): PowerShell has a rich library of cmdlets that can manage nearly all aspects of Windows systems.
- Powerful scripting: Beyond simple commands, PowerShell allows the creation of complex scripts to automate sequences of operations. Object management: PowerShell handles data as objects, which allows for richer and more flexible data manipulation. Interoperability: Interacts with different systems and technologies, thus facilitating integration with cloud services, databases, and other applications.

Installing PowerShell

The installation of PowerShell varies depending on the operating system. PowerShell is pre-installed on recent versions of Windows.

```
# Installation of PowerShell on Ubuntu
# Updates the package list and installs PowerShell.
sudo apt-get update
sudo apt-get install -y powershell

# Installation of PowerShell on macOS
# Updates Homebrew and installs PowerShell via cask.
brew update
brew install --cask powershell
```

Once installed, you can launch PowerShell by typing `pwsh` in your terminal on Linux or macOS, and by searching for PowerShell in the Start menu on Windows.

PowerShell is pre-installed on recent versions of Windows. However, to obtain the latest version or to install PowerShell on older versions of Windows, visit the official GitHub page for PowerShell (https://github.com/PowerShell/PowerShell) and download the MSI for your version of Windows.

Cmdlets

PowerShell uses an architecture based on cmdlets to execute commands. Cmdlets are lightweight commands written in .NET designed to perform specific tasks. Cmdlets follow a Verb-Noun naming convention, which makes the commands intuitive and easy to understand.

```
# Display help: Get-Help <CmdletName>
Get-Help Get-Process

# Listing files and folders
Get-ChildItem

# Searching for specific files: Get-ChildItem -Path <Path> -
Include <Pattern>
Get-ChildItem -Path C:\Users\ -Include *.txt

# Listing aliases
Get-Alias

# Example of an alias for Get-ChildItem
ls
```

Scripts & Execution

PowerShell scripts allow you to automate tasks by grouping multiple commands into a file.
To create a PowerShell script, write your commands in a text file and save it with the .ps1 extension. By default, PowerShell restricts script execution to protect against the execution of malicious scripts. Use `Get-ExecutionPolicy` to view the current policy and `Set-ExecutionPolicy` to change it.

```powershell
# Running a PowerShell script
.\monScript.ps1

# Checking the current execution policy
Get-ExecutionPolicy

# Checking the current execution policy
Get-ExecutionPolicy -List

# Allows the execution of PowerShell scripts signed by the
enterprise (ADCS)
Set-ExecutionPolicy AllSigned
# Allows the execution of PowerShell scripts signed by a
trusted publisher.
Set-ExecutionPolicy RemoteSigned
# Allows the execution of PowerShell scripts from any source
(If the script is from the internet, a warning message is
displayed)
Set-ExecutionPolicy Unrestricted
# Allows the execution of PowerShell scripts without any
checks
Set-ExecutionPolicy Bypass
# /!\ Caution: Changing the execution policy can affect the
security of your system. /!\
```

Script Signing

PowerShell script signing is an important security measure that ensures the integrity and authenticity of the script. It allows users to verify that the script has not been altered since it was signed and that the signature comes from a trusted developer.

You must obtain a code signing certificate from a recognized Certificate Authority (CA) for maximum trust.

```powershell
# Creating a self-signed certificate
$cert = New-SelfSignedCertificate -DnsName "MonNomDeScript" -
CertStoreLocation "Cert:\CurrentUser\My" -Type
CodeSigningCert

# Signing the script
Set-AuthenticodeSignature -FilePath
"C:\Path\To\YourScript.ps1" -Certificate $cert

# Checking the signature
Get-AuthenticodeSignature -FilePath
"C:\Path\To\YourScript.ps1"
```

Error Handling

Error management is crucial for writing robust and reliable PowerShell scripts. PowerShell provides several mechanisms for handling, catching, and responding to errors that may occur during the execution of a script. By structuring your code with Try, Catch, and Finally blocks, and implementing logging strategies, you can create more robust and maintainable scripts.

```powershell
# Displaying detailed errors about the current session
$Error

# Determines how PowerShell reacts to errors. Common values include: Stop, Continue, SilentlyContinue.
$ErrorActionPreference = "Stop"

# Running a command and redirecting the error output to a log file
Get-ChildItem -Path C:\ -Recurse 2>> C:\monScriptErreurs.log

# Example of using Try, Catch, and Finally
Try {
    # Attempt to execute a command that may fail
    Get-ChildItem "C:\NonexistentPath"
} Catch {
    # This block is executed in case of an error
    Write-Error "An error occurred."
} Finally {
    # This block always executes, ideal for cleanup
    Write-Host "End of Try-Catch block execution."
}
```

Data Management

Data management is an integral part of task automation with PowerShell. There is a wide range of cmdlets available for manipulating files and folders, facilitating the management of the file system.

```powershell
# Creating a new folder
New-Item -Path 'C:\MyNewFolder' -ItemType Directory

# Copying a file or folder to a new destination.
Copy-Item -Path 'C:\MyFile.txt' -Destination
'C:\MyNewFolder\MyCopiedFile.txt'

# Deleting the specified file or folder.
Remove-Item -Path 'C:\MyNewFolder\MyCopiedFile.txt'

# Reading the contents of a file
Get-Content -Path "C:\Temp\NewFolder\file_copy.txt"

# Writing to a file
$content | Out-File -FilePath
"C:\Temp\NewFolder\file_copy.txt"

# Manipulation with Regex
if ($fullName -match "\bDoe\b") {
    Write-Host "The last name Doe was found."
}

# Creating a custom object
$myObject = [PSCustomObject]@{
    LastName = "Doe"
    FirstName = "John"
    Age = 30
}

# Converting text to Json format
$jsonString = '{"name": "John Doe", "age": 32}' |
ConvertFrom-Json
# Converting text back to Json format
$backToJson = $jsonString | ConvertTo-Json
```

Modules & Extensions

PowerShell modules and extensions significantly enhance the power and flexibility of PowerShell by adding new cmdlets and functionalities.

```powershell
# Search for available modules in the PowerShell Gallery (default)
Find-Module -Name Azure* -Repository "PSGallery"

# Install a module
Install-Module -Name Az

# Load a module into the current PowerShell session. (Sometimes necessary)
Import-Module -Name Az

# Update modules
Update-Module -Name Az

# List the registries on the machine
Get-PSRepository

# Add a registry
Register-PSRepository -Name "MyEnterpriseRepo" -
SourceLocation "https://myenterprise.com/nuget" -
InstallationPolicy Trusted

# Modify a registry
Set-PSRepository -Name "MyEnterpriseRepo" -SourceLocation
"https://newurl.com/nuget"

# Remove a registry from the list
Unregister-PSRepository -Name "MyEnterpriseRepo"
```

Creating a Cmdlet

The simplest way to create a cmdlet is to define an advanced function. These functions use attributes and parameters that mimic the behavior of PowerShell's native cmdlets. This structure provides a framework for creating custom cmdlets, with Begin, Process, and End blocks to manage initialization, main processing, and cleanup.

```powershell
function Get-MyCustomCmdlet {
    [CmdletBinding()]
    Param (
        [Parameter(Mandatory=$true)]
        [string]$Param1,

        [Parameter(Mandatory=$false)]
        [int]$Param2
    )

    Begin {
        Write-Host "Initializing my custom cmdlet"
    }

    Process {
        Write-Host "Processing with Param1: $Param1 and
Param2: $Param2"
        # Your cmdlet logic here
    }

    End {
        Write-Host "Cleanup after cmdlet execution"
    }
}
```

Creating a Module

To make your cmdlets easily reusable and shareable, you can encapsulate them in a PowerShell module. A module can contain multiple cmdlets, variables, and other functions that you wish to group together.

Create a .psm1 file, which will contain the code for your cmdlet or advanced functions.

MyModule.psm1

```
function Get-MyCustomCmdlet {
    [CmdletBinding()]
    Param (
        [string]$Param1,
        [int]$Param2
    )
    Write-Output "Here are my parameters: $Param1 and
$Param2"
}
```

A module manifest (.psd1) describes your module, its dependencies, version, and other metadata. Although not mandatory for all modules, it is a good practice for modules intended to be shared.

Creating a Module

MyModule.psd1

```
@{
    ModuleVersion = '1.0'
    GUID = 'some-guid-value'
    Author = 'Your Name'
    CompanyName = 'Your Company'
    PowerShellVersion = '5.1'
    RootModule = 'MyModule.psm1'
    FunctionsToExport = '*'
}
```

Place your module in one of the directories listed in `$env:PSModulePath` to make it automatically discoverable by PowerShell, or manually import it using `Import-Module`.

```
Import-Module .\Chemin\vers\MonModule.psm1
```

Name your cmdlets and parameters following PowerShell conventions (Verb-Noun) to maintain consistency with native cmdlets.

Automation - Cron

Scheduled task management in Windows environments via PowerShell is a powerful feature for automating the execution of scripts at specific times or in response to certain events.

```powershell
# Creating a scheduled task with PowerShell to run a script daily
$Action = New-ScheduledTaskAction -Execute 'Powershell.exe' -Argument '-NoProfile -WindowStyle Hidden -File "C:\Path\To\YourScript.ps1"'
$Trigger = New-ScheduledTaskTrigger -Daily -At 3am
Register-ScheduledTask -Action $Action -Trigger $Trigger -TaskName "DailyBackupTask" -Description "Executes a backup script daily at 3 AM"

# Listing all scheduled tasks
Get-ScheduledTask | Select-Object TaskName, State

# Disabling a scheduled task
Disable-ScheduledTask -TaskName "DailyBackupTask"

# Enabling a scheduled task
Enable-ScheduledTask -TaskName "DailyBackupTask"

# Removing a scheduled task
Unregister-ScheduledTask -TaskName "DailyBackupTask" -Confirm:$false
```

Triggers

PowerShell offers a variety of triggers for scheduled tasks, allowing great flexibility in how and when tasks are executed. These triggers can be configured to respond to specific events, precise schedules, or other conditions.

```powershell
# Daily (Daily): Executes a task every day at a specified time.
$Trigger = New-ScheduledTaskTrigger -Daily -At 9am
# You can also use Weekly (Weekly), Once (Once)

# Executes a task each time the computer starts.
$Trigger = New-ScheduledTaskTrigger -AtStartup

# Executes a task each time a user logs on.
$Trigger = New-ScheduledTaskTrigger -AtLogon -User "DOMAIN\User"

# Executes a task when the computer is idle for a specified period.
$Trigger = New-ScheduledTaskTrigger -OnIdle
```

Application Monitoring

In PowerShell and Windows Task Scheduler, there is no built-in functionality to directly trigger tasks in response to opening or closing specific files or the execution and termination of specific applications. However, it is possible to approximate these functionalities by using monitoring strategies or writing scripts that monitor the system's status and trigger actions accordingly.

```
$processName = "notepad"
$process = Get-Process | Where-Object { $_.Name -eq
$processName }

if ($process) {
    # Logic to execute if the application is running
    Write-Host "$processName is running."
} else {
    # Logic to execute if the application is not found
    Write-Host "$processName is not running."
}
```

File Monitoring

For files, the task is more complex. PowerShell does not directly provide a way to trigger scripts or tasks in response to events such as opening or modifying files. However, you can use the .NET `FileSystemWatcher` component to monitor file changes in a specific directory. This component can be set up to trigger actions in PowerShell scripts when specified file events occur, such as creations, deletions, modifications, or renamings of files.

```powershell
$watcher = New-Object System.IO.FileSystemWatcher
$watcher.Path = "C:\MyFolder"
$watcher.NotifyFilter = [System.IO.NotifyFilters]::FileName -
or [System.IO.NotifyFilters]::LastWrite
$watcher.Filter = "*.*"

# Defines what should happen when an event is triggered
$action = { param($sender, $e) Write-Host "The file
'$($e.FullPath)' has been modified." }

# Subscribes to events
Register-ObjectEvent -InputObject $watcher -EventName Changed
-Action $action

# Starts monitoring
$watcher.EnableRaisingEvents = $true
```

Remoting

Executing PowerShell scripts across multiple machines is a powerful capability for managing distributed systems, updating software, or deploying configurations. `Invoke-Command` is the cmdlet of choice for executing commands or scripts on one or multiple remote machines. It requires that PowerShell Remoting be enabled on the target machines.

```powershell
# Enable PowerShell Remoting
Enable-PSRemoting -Force
```

```powershell
# Server01 can represent either a hostname or an IP address
$Computers = @("Server01", "Server02", "Server03")

$ScriptBlock = {
    Get-PSDrive C | Select-Object Used, Free
}

# Retrieves used and free space on the C drive of each listed
server and displays the results.
Invoke-Command -ComputerName $Computers -ScriptBlock
$ScriptBlock
```

Sessions

PSSessions provide a powerful and flexible way to manage remote machines. You can use them to perform remote administration tasks while minimizing risks to your environment.

```powershell
# List of target computers
$Computers = @("Workstation01", "Workstation02",
"Workstation03")

# Script block to execute on the remote computers
$ScriptBlock = {
    # Script to install the update
    Start-Process "C:\Path\To\UpdateInstaller.exe" -
ArgumentList "/silent" -Wait
}

# Requesting credentials once for all connections
$Credential = Get-Credential

# Creating PSSessions for each computer using the provided
credentials
$Sessions = $Computers | ForEach-Object {
    New-PSSession -ComputerName $_ -Credential $Credential
}

# Executing the script on each session
Invoke-Command -Session $Sessions -ScriptBlock $ScriptBlock

# Closing the sessions after script execution
$Sessions | ForEach-Object {
    Remove-PSSession -Session $_
}
```

DSC

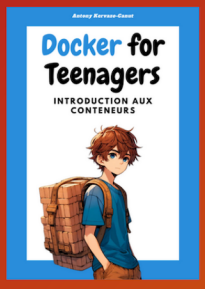
Desired State Configuration (DSC) is a configuration management platform integrated with PowerShell that allows you to define the desired state of your software and hardware resources and ensures that these resources are configured correctly.

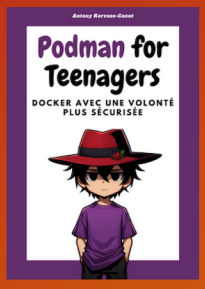
The documentation for DSC is available here:
]https://learn.microsoft.com/en-us/powershell/dsc/overview?view=dsc-3.0

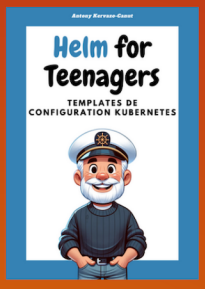
As the tool evolves and changes depending on the version you are using, I am only providing documentation here. It might be an alternative for those who prefer not to use tools like Ansible.

In the same collection

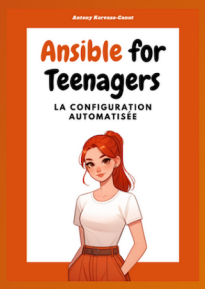
Container Orchestration and Management





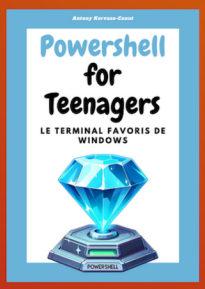


Infrastructure as Code



Security & Secrets Management

Development & CI/CD



↓ FOLLOW ME ↓

ANTONYCANUT

ANTONY KERVAZO-CANUT