

Antony Kervazo-Canut

Powershell for Teenagers

**LE TERMINAL FAVORIS DE
WINDOWS**



SOMMAIRE

Introduction	3
Installation de Powershell	4
Les Cmdlets	5
Scripts et exécutions	6
Signature de scripts	7
Gestion des erreurs	8
Gestion des données	9
Modules et Extensions	10
Création d'une Cmdlet	11
Création d'un module	12
Automatisation - Cron	14
Triggers	15
Surveillance d'une application	16
Surveillance d'un fichier	17
Remoting	18
Sessions	19
DSC	20

Introduction



PowerShell est un langage de script et un shell de ligne de commande développé par Microsoft. Principalement conçu pour l'administration système, il permet l'automatisation des tâches administratives, la gestion des configurations et l'interrogation de données. PowerShell étend les capacités des scripts batch traditionnels en permettant l'exécution de commandes complexes et la manipulation de l'objet système.

PowerShell offre plusieurs avantages pour les professionnels IT et les développeurs :

- Automatisation : Réduit les tâches répétitives en permettant l'automatisation des processus administratifs.
- Accès à une vaste gamme de commandes (Cmdlets) : PowerShell dispose d'une bibliothèque riche en cmdlets permettant de gérer presque tous les aspects des systèmes Windows.
- Scripting puissant : Au-delà des commandes simples, PowerShell permet de créer des scripts complexes pour automatiser des séquences d'opérations.
- Gestion des objets : PowerShell traite les données en tant qu'objets, ce qui permet une manipulation plus riche et plus flexible des données.
- Interopérabilité : Interagit avec différents systèmes et technologies, facilitant ainsi l'intégration avec les services cloud, les bases de données, et d'autres applications.

Installation de Powershell



L'installation de PowerShell varie selon le système d'exploitation. PowerShell est préinstallé sur les versions récentes de Windows.

```
● ● ●

# Installation de PowerShell sur Ubuntu
# Met à jour la liste des paquets et installe PowerShell.
sudo apt-get update
sudo apt-get install -y powershell

# Installation de PowerShell sur macOS
# Met à jour Homebrew et installe PowerShell via cask.
brew update
brew install --cask powershell
```

Une fois installé, vous pouvez lancer PowerShell en tapant pwsh dans votre terminal sur Linux ou macOS, et en cherchant PowerShell dans le menu Démarrer sur Windows.

PowerShell est préinstallé sur les versions récentes de Windows. Cependant, pour obtenir la dernière version ou pour installer PowerShell sur des versions plus anciennes de Windows visitez le [site officiel de GitHub pour PowerShell](#) et installez le MSI de votre version de Windows.



Les Cmdlets

PowerShell utilise une architecture basée sur des cmdlets pour exécuter des commandes. Les cmdlets sont des commandes légères écrites en .NET pour accomplir des tâches spécifiques. Les cmdlets suivent une convention de nommage Verbe-Nom, ce qui rend les commandes intuitives et faciles à comprendre.

```
● ● ●

# Affichage de l'aide : Get-Help <CmdletName>
Get-Help Get-Process

# Listing des fichiers et dossiers
Get-ChildItem

# Recherche de fichiers spécifiques : Get-ChildItem -Path
# <Path> -Include <Pattern>
Get-ChildItem -Path C:\Users\ -Include *.txt

# Listing des aliases
Get-Alias

# Exemple d'un alias de Get-ChildItem
ls
```

Scripts et exécutions



Les scripts PowerShell permettent d'automatiser des tâches en regroupant plusieurs commandes dans un fichier.

Pour créer un script PowerShell, écrivez vos commandes dans un fichier texte et sauvegardez-le avec l'extension .ps1. Par défaut, PowerShell limite l'exécution des scripts pour protéger contre l'exécution de scripts malveillants. Utilisez Get-ExecutionPolicy pour voir la politique actuelle et Set-ExecutionPolicy pour la changer.

```
# Exécution d'un script PowerShell
.\monScript.ps1

# Vérification de la politique d'exécution actuelle
Get-ExecutionPolicy

# Vérification de la politique d'exécution actuelle
Get-ExecutionPolicy -List

# Permet l'exécution de scripts Powershell signé par
l'entreprise (ADCS)
Set-ExecutionPolicy AllSigned
# Permet l'exécution de scripts PowerShell signé un éditeur
de confiance.
Set-ExecutionPolicy RemoteSigned
# Permet l'exécution de scripts PowerShell de toute
provenance (Si le script provient d'internet, un message
d'avertissement s'affiche)
Set-ExecutionPolicy Unrestricted
# Permet l'exécution de scripts PowerShell sans aucun
contrôle
Set-ExecutionPolicy Bypass
# /!\ Attention : Modifier la politique d'exécution peut
affecter la sécurité de votre système. /!\  
6
```

Signature de scripts



La signature de scripts PowerShell est une mesure de sécurité importante qui assure l'intégrité et l'authenticité du script. Elle permet aux utilisateurs de vérifier que le script n'a pas été modifié depuis sa signature et que la signature provient d'un développeur de confiance.

Vous devez obtenir un certificat de signature de code d'une CA (Autorité de Certification) reconnue pour une confiance maximale.

```
# Création d'un certificat auto-signé
$cert = New-SelfSignedCertificate -DnsName "MonNomDeScript" -
CertStoreLocation "Cert:\CurrentUser\My" -Type
CodeSigningCert

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

# Vérification de la signature
Get-AuthenticodeSignature -FilePath
"C:\Path\To\YourScript.ps1"
```

Gestion des erreurs



La gestion des erreurs est cruciale pour écrire des scripts PowerShell robustes et fiables. PowerShell fournit plusieurs mécanismes pour gérer, attraper, et répondre aux erreurs qui peuvent survenir pendant l'exécution d'un script. En structurant votre code avec Try, Catch, et Finally, et en implémentant des stratégies de journalisation, vous pouvez créer des scripts plus robustes et plus faciles à maintenir.

```
# Affichage des erreurs détaillées sur la session en cours
$Error

# Détermine comment PowerShell réagit aux erreurs. Les
valeurs communes incluent : Stop, Continue, SilentlyContinue.
$ErrorActionPreference = "Stop"

# Exécution d'une commande et redirection de la sortie
# d'erreur vers un fichier journal
Get-ChildItem -Path C:\ -Recurse 2>> C:\monScriptErreurs.log

# Exemple d'utilisation de Try, Catch, et Finally
Try {
    # Tentative d'exécution d'une commande qui peut échouer
    Get-ChildItem "C:\Chemin\Inexistant"
} Catch {
    # Ce bloc est exécuté en cas d'erreur
    Write-Error "Une erreur est survenue."
} Finally {
    # Ce bloc s'exécute toujours, idéal pour le nettoyage
    Write-Host "Fin de l'exécution du bloc Try-Catch."
}
```

Gestion des données



La gestion des données est une partie intégrale de l'automatisation des tâches avec PowerShell. Il existe une gamme étendue de cmdlets pour la manipulation de fichiers et de dossiers, facilitant la gestion du système de fichiers.

```
# Création d'un nouveau dossier
New-Item -Path 'C:\MonNouveauDossier' -ItemType Directory

# Copie un fichier ou un dossier vers une nouvelle destination.
Copy-Item -Path 'C:\MonFichier.txt' -Destination
'C:\MonNouveauDossier\MonFichierCopié.txt'

# Supprime le fichier ou dossier spécifié.
Remove-Item -Path 'C:\MonNouveauDossier\MonFichierCopié.txt'

# Lecture du contenu d'un fichier
Get-Content -Path "C:\Temp\NewFolder\file_copy.txt"

# Écriture dans un fichier
$content | Out-File -FilePath
"C:\Temp\NewFolder\file_copy.txt"

# Manipulation avec Regex
if ($fullName -match "\bDoe\b") {
    Write-Host "Le nom de famille Doe a été trouvé."
}

# Conversion d'un texte vers un format Json
$jsonString = '{"name": "John Doe", "age": 32}' |
ConvertFrom-Json
# Conversion d'un texte vers un format texte
$backToJson = $jsonString | ConvertTo-Json
```

Modules et Extensions



Les modules et extensions PowerShell augmentent considérablement la puissance et la flexibilité de PowerShell en ajoutant de nouvelles cmdlets et fonctionnalités.

```
● ● ●

# Rechercher des modules disponibles dans PowerShell Gallery
# (par défaut)
Find-Module -Name Azure* -Repository "PSGallery"

# Installer un module
Install-Module -Name Az

# Charge un module dans la session PowerShell actuelle.
# (Parfois nécessaire)
Import-Module -Name Az

# Mise à jour de modules
Update-Module -Name Az

# Liste les registries sur la machine
Get-PSRepository

# Ajoute une registry
Register-PSRepository -Name "MonEnterpriseRepo" -
SourceLocation "https://monentreprise.com/nuget" -
InstallationPolicy Trusted

# Modifie une registry
Set-PSRepository -Name "MonEnterpriseRepo" -SourceLocation
"https://nouvelleurl.com/nuget"

# Supprime une registry de la liste
Unregister-PSRepository -Name "MonEnterpriseRepo"
```

Création d'une Cmdlet



La manière la plus simple de créer une cmdlet est de définir une fonction avancée. Ces fonctions utilisent des attributs et des paramètres qui imitent le comportement des cmdlets natives de PowerShell. Cette structure fournit un squelette pour créer des cmdlets personnalisées, avec des sections Begin, Process, et End pour gérer l'initialisation, le traitement principal, et le nettoyage.

```
function Get-MyCustomCmdlet {
    [CmdletBinding()]
    Param (
        [Parameter(Mandatory=$true)]
        [string]$Param1,
        [Parameter(Mandatory=$false)]
        [int]$Param2
    )

    Begin {
        Write-Host "Initialisation de ma cmdlet personnalisée"
    }

    Process {
        Write-Host "Traitement avec Param1: $Param1 et Param2: $Param2"
        # Logique de votre cmdlet ici
    }

    End {
        Write-Host "Nettoyage après l'exécution de la cmdlet"
    }
}
```

Création d'un module



Pour rendre vos cmdlets facilement réutilisables et partageables, vous pouvez les encapsuler dans un module PowerShell. Un module peut contenir plusieurs cmdlets, des variables, et d'autres fonctions que vous souhaitez regrouper.

Créez un fichier .psm1, qui contiendra le code de votre cmdlet ou de vos fonctions avancées.

```
# MonModule.psm1
function Get-MyCustomCmdlet {
    [CmdletBinding()]
    Param (
        [string]$Param1,
        [int]$Param2
    )
    Write-Output "Voici mon paramètre: $Param1 et $Param2"
}
```

Un manifeste de module .psd1 décrit votre module, ses dépendances, la version, et d'autres métadonnées. Bien que ce ne soit pas obligatoire pour tous les modules, c'est une bonne pratique pour les modules destinés à être partagés.

Création d'un module



```
# MonModule.psd1
@"
    ModuleVersion = '1.0'
    GUID = 'some-guid-value'
    Author = 'Votre Nom'
    CompanyName = 'Votre Entreprise'
    PowerShellVersion = '5.1'
    RootModule = 'MonModule.psm1'
    FunctionsToExport = '*'
"
```

Placez votre module dans l'un des répertoires listés dans \$env:PSModulePath pour le rendre automatiquement découvrable par PowerShell, ou importez-le manuellement avec Import-Module.

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

Nommez vos cmdlets et paramètres en suivant les conventions PowerShell (Verbe-Nom), pour une cohérence avec les cmdlets natives.

Automatisation - Cron



La gestion des tâches planifiées dans les environnements Windows via PowerShell est une fonctionnalité puissante pour automatiser l'exécution de scripts à des moments spécifiques ou en réponse à certains événements.

```
● ● ●

# Création d'une tâche planifiée avec PowerShell pour
# exécuter un script quotidiennement
$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"

# Lister toutes les tâches planifiées
Get-ScheduledTask | Select-Object TaskName, State

# Désactiver une tâche planifiée
Disable-ScheduledTask -TaskName "DailyBackupTask"

# Activer une tâche planifiée
Enable-ScheduledTask -TaskName "DailyBackupTask"

# Supprimer une tâche planifiée
Unregister-ScheduledTask -TaskName "DailyBackupTask" -
    Confirm:$false
```

Triggers



PowerShell offre une variété de triggers pour les tâches planifiées, permettant une grande flexibilité dans la manière et le moment de l'exécution des tâches. Ces triggers peuvent être configurés pour répondre à des événements spécifiques, à des calendriers précis, ou à d'autres conditions.

```
# Daily (Quotidien) : Exécute une tâche tous les jours à une
heure spécifiée.
$Trigger = New-ScheduledTaskTrigger -Daily -At 9am
# Vous pouvez aussi utiliser Weekly (Hebdomadaire), Once (Une
fois)

# Exécute une tâche chaque fois que l'ordinateur démarre.
$Trigger = New-ScheduledTaskTrigger -AtStartup

# Exécute une tâche chaque fois qu'un utilisateur ouvre une
session.
$Trigger = New-ScheduledTaskTrigger -AtLogon -User
"DOMAIN\User"

# Exécute une tâche lorsque l'ordinateur est inactif pendant
une période spécifiée.
$Trigger = New-ScheduledTaskTrigger -OnIdle
```

Surveillance d'une application



Dans PowerShell et le Planificateur de tâches Windows, il n'existe pas de fonctionnalité intégrée permettant de déclencher directement des tâches en réponse à l'ouverture ou à la fermeture de fichiers spécifiques ou à l'exécution et à l'arrêt d'applications spécifiques. Cependant, il est possible d'approcher ces fonctionnalités en utilisant des stratégies de surveillance ou en écrivant des scripts qui surveillent l'état du système et déclenchent des actions en conséquence.

```
● ● ●

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

if ($process) {
    # Logique à exécuter si l'application est en cours
    # d'exécution
    Write-Host "$processName est en cours d'exécution."
} else {
    # Logique à exécuter si l'application n'est pas trouvée
    Write-Host "$processName n'est pas en cours d'exécution."
}
```

Surveillance d'un fichier



Pour les fichiers, la tâche est plus complexe. PowerShell ne fournit pas directement un moyen de déclencher des scripts ou des tâches en réponse à des événements tels que l'ouverture ou la modification de fichiers. Cependant, vous pouvez utiliser le composant .NET FileSystemWatcher pour surveiller les modifications de fichiers dans un répertoire spécifique.

```
● ● ●

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

# Définit ce qui doit se passer lorsque un événement est déclenché
$action = { param($sender, $e) Write-Host "Le fichier '$($e.FullPath)' a été modifié." }

# S'abonne aux événements
Register-ObjectEvent -InputObject $watcher -EventName Changed -Action $action

# Démarre la surveillance
$watcher.EnableRaisingEvents = $true
```

Remoting



L'exécution de scripts PowerShell sur plusieurs machines est une capacité puissante pour l'administration de systèmes distribués, la mise à jour de logiciels, ou encore pour le déploiement de configurations. Invoke-Command est la cmdlet de prédilection pour exécuter des commandes ou des scripts sur une ou plusieurs machines distantes. Elle nécessite que PowerShell Remoting soit activé sur les machines cibles.

```
# Active Powershell Remoting
Enable-PSRemoting -Force
```

```
# Server01 peuvent représenter soit un nom d'hôte soit une
# adresse IP
$Computers = @("Server01", "Server02", "Server03")

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

# récupère et affiche l'espace utilisé et libre sur le
# lecteur C de chaque serveur listé et affiche les résultats.
Invoke-Command -ComputerName $Computers -ScriptBlock
$ScriptBlock
```

Sessions



Les PSSessions offrent un moyen puissant et flexible de gérer les machines distantes. vous pouvez les utiliser pour effectuer des tâches d'administration distante tout en minimisant les risques pour votre environnement.

```
# Liste des ordinateurs cibles
$Computers = @("Workstation01", "Workstation02",
"Workstation03")

# Bloc de script à exécuter sur les ordinateurs distants
$ScriptBlock = {
    # Script d'installation de la mise à jour
    Start-Process "C:\Path\To\UpdateInstaller.exe" -
    ArgumentList "/silent" -Wait
}

# Demande d'identification une seule fois pour toutes les
# connexions
$Credential = Get-Credential

# Création de sessions PSSession pour chaque ordinateur en
# utilisant les identifiants fournis
$Sessions = $Computers | ForEach-Object {
    New-PSSession -ComputerName $_ -Credential $Credential
}

# Exécution du script sur chaque session
Invoke-Command -Session $Sessions -ScriptBlock $ScriptBlock

# Fermeture des sessions après l'exécution du script
$Sessions | ForEach-Object {
    Remove-PSSession -Session $_
}
```

DSC



Desired State Configuration (DSC) est une plateforme de gestion de configuration intégrée à PowerShell qui vous permet de définir l'état souhaité (ou "desired state") de vos ressources logicielles et matérielles et assure que ces ressources sont configurées correctement.

La documentation sur DSC est disponible ici :
[https://learn.microsoft.com/en-us/powershell/dsc/overview?
view=dsc-3.0](https://learn.microsoft.com/en-us/powershell/dsc/overview?view=dsc-3.0)

Comme l'outil évolue et change en fonction de la version que vous utilisez, je ne place que de la documentation ici. Et c'est peut être une alternative à ceux qui refuse d'utiliser des outils comme Ansible.

Dans la même collection

Orchestration et Gestion de Conteneurs



Infrastructure as Code



Sécurité & Gestion des secrets



Développement & CI/CD



↓ FOLLOW ME ↓



[ANTONYCANUT](#)



[ANTONY KERVAZO-CANUT](#)