PowerShell 6 / 7 - Perfectionnement

m2iformation.fr

Sommaire

- Utilisation des objets avancés
  - Gestion d'un objet de A à Z
  - Ajout de propriétés personnalisées à un objet, gestion des classes
  - Manipuler les objets
  - Gérer les tableaux et les variables avancées

Sommaire

- Construire scripts et fonctions complexes
  - Modulariser son code au maximum
  - Produire des sorties complexes
  - Documentation intégrée
  - Gestion des erreurs avancées

Sommaire

- Concepts avancés sur les fonctions
  - Définir les paramètres
  - Employer des jeux de paramètres
  - Gestion des dépendances et des prérequis
  - Gestion de la sécurité et des erreurs intermodules avancées
  - Exploiter les points d'arrêts en détail
  - Gérer les boucles avancées
  - Gérer les exécutions

Sommaire

- Sécurité et signatures
  - Vue d'ensemble de la sécurité au sein de PowerShell
  - Gestion des accès, des secrets et des certifications
  - Mise en oeuvre au sein d'un parc
- Normaliser un parc avec PowerShell DSC et les workflows
  - Introduction à Desired State Configuration
  - Mise en oeuvre
  - D'une configuration de conformité Des workflows

Gestion d'un objet de A à Z

En PowerShell, **tout est objet**. C'est ce qui le distingue fondamentalement de Bash ou CMD.

Exemple : `Get-Process` ne renvoie pas du texte, mais une **collection d'objets .NET** de type `System.Diagnostics.Process`.

Chaque objet a :

- des propriétés : des valeurs associées à l'objet
- des méthodes : des fonctions associées à cet objet

Gestion d'un objet de A à Z

- Pour traiter les données de manière fiable : pas besoin de parser du texte
- Pour écrire des scripts robustes et réutilisables
- Pour transmettre des données entre fonctions ou scripts

Plusieurs options :

1. Créer un objet de type PSCustomObject
2. Enrichir un objet existant avec Add-Member
3. Créer un type fort avec une classe (PowerShell 5+)

Ajout de propriétés personnalisées à un objet, gestion des classes

Ajout de propriétés personnalisées

Imaginons que l'on récupère une liste de serveurs et que l'on veuille ajouter une information métier :

- Un tag
- Une date de dernier audit
- Un statut de conformité

Ajout de propriétés personnalisées à un objet, gestion des classes

Ajout de propriétés personnalisées

- **Add-Member**

`Add-Member` permet d'ajouter une propriété à un objet existant. Cela fonctionne sur tous les objets, y compris ceux issus d'une commande système (ex: Get-Process).

```
$p = Get-Process -Name notepad
$p | Add-Member -MemberType NoteProperty -Name "Commentaire" -Value "Process critique"
$p.Commentaire
```

- **Limitations de Add-Member :**
  - Propriété ajoutée dynamiquement, Pas de typage fort, Moins lisible si l'on crée des objets complexes

Ajout de propriétés personnalisées à un objet, gestion des classes

Ajout de propriétés personnalisées

- **PSCustomObject**

Plus adapté quand on veut **créer un objet proprement dès le départ** :

```
$serveur = [PSCustomObject]@{
    Nom = "SRV01"
    IP = "192.168.1.1"
    Etat = "OK"
}
```

- **Avantages :**
  - Syntaxe lisible, Export CSV ou JSON facile, Objet cohérent dès la création

Ajout de propriétés personnalisées à un objet, gestion des classes

Gestion des classes

Quand on écrit des scripts **modulaires et réutilisables**, les classes permettent :

- De structurer son code
- De créer des objets fortement typés
- D'ajouter des méthodes métiers

Cela permet de se rapprocher des bonnes pratiques de programmation (orienté objet).

Ajout de propriétés personnalisées à un objet, gestion des classes

Gestion des classes

```
class Serveur {
    [string]$Nom
    [string]$IP
    [string]$Etat
    [datetime]$DateDernierAudit

    Serveur([string]$nom, [string]$ip) {
        $this.Nom = $nom
        $this.IP = $ip
        $this.Etat = "Non vérifié"
        $this.DateDernierAudit = (Get-Date)
    }

    [void]AfficherInfos() {
        Write-Output "Nom=$($this.Nom), IP=$($this.IP), Etat=$($this.Etat), Date=$($this.DateDernierAudit)"
    }
}
```

Ajout de propriétés personnalisées à un objet, gestion des classes

Gestion des classes

```
$srv = [Serveur]::new("SRV02", "192.168.1.2")
$srv.AfficherInfos()
$srv.Etat = "OK"
$srv.AfficherInfos()
```

Avantages des classes :

- Encapsulation
- Typage fort
- Méthodes associées à l'objet
- Recommandé pour les modules réutilisables

Manipuler les objets

- Lire les propriétés d'un objet
- Modifier les propriétés
- Appeler les méthodes
- Filtrer, trier, transformer des objets

Manipuler les objets

- **Lire les propriétés**

```
$p = Get-Process -Name notepad
$p.Name
$p.Id
```

- **Modifier les propriétés (si l'objet le permet)**

```
$monObjet.Etat = "KO"
```

- **Appeler une méthode**

```
$p.Kill() # Méthode native de l'objet Process
```

Manipuler des collections d'objets

Les cmdlets retournent très souvent des **collections** d'objets (tableaux).

On peut utiliser la pipeline pour les manipuler.

- **Parcourir :**

```
$processus = Get-Process
foreach ($p in $processus) {
    "$($p.Name) - $($p.CPU)"
}
```

Manipuler des collections d'objets

Filtrer :

```
$processus | Where-Object { $_.CPU -gt 10 }
```

Trier :

```
$processus | Sort-Object -Property CPU -Descending
```

- **Sélectionner :**

```
$processus | Select-Object Name, CPU
```

Gérer les tableaux et les variables avancées

- **Tableaux simples**

Un tableau est une collection ordonnée.

Création :

```
$tableau = @(1, 2, 3, 4, 5)
```

Accès par index :

```
$tableau[0]
```

Parcours :

```
foreach ($n in $tableau) {
    "Valeur: $n"
}
```

Tableaux d'objets

Utile pour gérer des listes de ressources (serveurs, utilisateurs, processus...).

```
$serveurs = @()
$serveurs += [PSCustomObject]@{ Nom = "SRV01"; IP = "192.168.1.1"; Etat = "OK" }
$serveurs += [PSCustomObject]@{ Nom = "SRV02"; IP = "192.168.1.2"; Etat = "KO" }

foreach ($s in $serveurs) {
    "$($s.Nom) - $($s.Etat)"
}
```

Hashtables (tables de hachage)

- **Création :**

```
$hash = @{
    "SRV01" = "192.168.1.1"
    "SRV02" = "192.168.1.2"
}
```

- **Accès :**

```
$hash["SRV01"]
```

- **Parcours :**

```
foreach ($cle in $hash.Keys) {
    "$cle -> $($hash[$cle])"
}
```

Tableaux imbriqués

Structure plus complexe :

```
$matrice = @(
    @(1, 2, 3),
    @(4, 5, 6)
)

$matrice[0][1] # renvoie 2
```

Construire scripts et fonctions complexes

Quand on débute, on écrit des scripts "monolithiques", linéaires.
En perfectionnement, l’objectif est de :

- Créer des fonctions réutilisables
- Avoir des paramètres
- Retourner des objets structurés
- Écrire du code modulaire et testable

Construire scripts et fonctions complexes

- **Syntaxe d'une fonction PowerShell**

```powershell
function Nom-Fonction {
    param(
        [string]$Param1,
        [int]$Param2
    )

    # Corps de la fonction
    Write-Output "Param1 = $Param1, Param2 = $Param2"
}
```

Construire scripts et fonctions complexes

Bonnes pratiques

- Utiliser des **paramètres nommés**
- **Retourner des objets** et non du texte
- Si la fonction est publique, ajouter de la **documentation intégrée**

Construire scripts et fonctions complexes

```powershell
function Get-InfosServeur {
    param(
        [string]$Nom,
        [string]$IP
    )
    $objet = [PSCustomObject]@{
        Nom = $Nom
        IP = $IP
        DateAudit = (Get-Date)
    }
    return $objet
}
# Appel de la fonction
$resultat = Get-InfosServeur -Nom "SRV01" -IP "192.168.1.1"
$resultat
```

Modulariser son code au maximum

Un script "propre" et professionnel doit :

- être découpé en fonctions réutilisables
- utiliser des **modules** (`.psm1`) si nécessaire
- limiter le code dans le corps principal

Avantages :

- Maintenance facilitée
- Réutilisation
- Test unitaire possible

Modulariser son code au maximum

```powershell
function Test-Connectivite {
    param([string]$IP)
    $ping = Test-Connection -ComputerName $IP -Count 1 -Quiet
    return $ping
}
function Get-InfosServeur {
    param([string]$Nom, [string]$IP)
    $etat = if (Test-Connectivite -IP $IP) { "OK" } else { "KO" }
    $objet = [PSCustomObject]@{
        Nom = $Nom
        IP = $IP
        Etat = $etat
    }
    return $objet
}
# Corps du script
$serveur = Get-InfosServeur -Nom "SRV02" -IP "192.168.1.2"
```

Produire des sorties complexes

Un script professionnel doit :

- produire des objets complexes (ex: tableaux d'objets)
- permettre l'export (CSV, JSON, XML)
- être exploitable par d'autres outils (pipeline)

Produire des sorties complexes

- **Tableau d'objets**

```powershell
$listeServeurs = @()
$listeServeurs += Get-InfosServeur -Nom "SRV01" -IP "192.168.1.1"
$listeServeurs += Get-InfosServeur -Nom "SRV02" -IP "192.168.1.2"

# Export CSV
$listeServeurs | Export-Csv -Path "rapport.csv" -NoTypeInformation

# Export JSON
$listeServeurs | ConvertTo-Json | Out-File "rapport.json"
```

Produire des sorties complexes

- **Pipeline-friendly**

On peut aussi concevoir une fonction qui accepte un objet en entrée et renvoie un objet :

```
function Set-TagServeur {
    param(
        [Parameter(ValueFromPipeline = $true)]
        $Serveur
    )
    process {
        $Serveur | Add-Member -MemberType NoteProperty -Name "Tag" -Value "Audit 2025"
        return $Serveur
    }
}
# Utilisation
$listeServeurs | Set-TagServeur
```

Documentation intégrée

- Pour rendre le code lisible et maintenable
- Pour générer une documentation automatique
- Pour être conforme aux standards de développement

Documentation intégrée

```powershell
function Get-InfosServeur {
    <#
    .SYNOPSIS
    Récupère les informations d'un serveur.

    .DESCRIPTION
    Cette fonction retourne un objet contenant le nom, l'adresse IP et la date d'audit.

    .PARAMETER Nom
    Nom du serveur.

    .PARAMETER IP
    Adresse IP du serveur.

    .EXAMPLE
    Get-InfosServeur -Nom "SRV01" -IP "192.168.1.1"

    .OUTPUTS
    PSCustomObject
    #>
    param(
        [string]$Nom,
        [string]$IP
    )

    $objet = [PSCustomObject]@{
        Nom = $Nom
        IP = $IP
        DateAudit = (Get-Date)
    }
```

Gestion des erreurs avancées

- Pour rendre les scripts robustes
- Pour éviter les plantages silencieux
- Pour mieux tracer les erreurs

Gestion des erreurs avancées

- **Erreurs non bloquantes (par défaut) :**

```
Get-Content "fichier-inexistant.txt"
```

PowerShell affiche un message d'erreur mais continue.

- **Forcer une exception :**

```
Get-Content "fichier-inexistant.txt" -ErrorAction Stop
```

Gestion des erreurs avancées

- **Try / Catch**

Le bloc `try / catch` permet de capturer et de traiter proprement les erreurs :

```
try {
    Get-Content "fichier-inexistant.txt" -ErrorAction Stop
}
catch {
    Write-Warning "Erreur lors de la lecture du fichier : $_"
}
```

Gestion des erreurs avancées

Finally

`finally` permet d'exécuter du code dans tous les cas (exemple : nettoyage) :

```
try {
    # Code risqué
}
catch {
    # Traitement de l'erreur
}
finally {
    # Toujours exécuté
}
```

Gestion des erreurs avancées

```powershell
function Test-Fichier {
    param([string]$Chemin)
    try {
        $contenu = Get-Content $Chemin -ErrorAction Stop
        Write-Output "Lecture OK : $Chemin"
    }
    catch {
        Write-Warning "Erreur : Impossible de lire le fichier $Chemin - $_"
    }
    finally {
        Write-Verbose "Fin du traitement du fichier $Chemin"
    }
}
# Appel
Test-Fichier -Chemin "test.txt"
```

Définir les paramètres

- Pour rendre les fonctions **réutilisables** et **flexibles**
- Pour **spécifier les entrées** attendues (avec types, valeurs par défaut, validation, etc.)

Avec le bloc `param()`. On peut définir :

- le type des paramètres
- des valeurs par défaut
- des validations

Définir les paramètres

- **Valeur par défaut**

```
param(
    [string]$Nom = "Inconnu"
)
```

- **Obligatoire**

```
param(
    [Parameter(Mandatory = $true)]
    [string]$Nom
)
```

Définir les paramètres

Validation

```
param(
    [ValidateSet("OK", "KO", "Inconnu")]
    [string]$Etat
)
```

Employer des jeux de paramètres

- Pour gérer plusieurs scénarios d'appel d'une fonction
- Pour proposer différents "modes" d'utilisation

Cela permet de créer des fonctions **polyvalentes**.

Exemple :

```
function Get-Infos {
    param(
        [Parameter(Mandatory, ParameterSetName = "ParNom")]
        [string]$Nom,

        [Parameter(Mandatory, ParameterSetName = "ParID")]
        [int]$ID
    )
    if ($PSCmdlet.ParameterSetName -eq "ParNom") {
        "Recherche par nom : $Nom"
    }
```

Gestion des dépendances et des prérequis

Dans un script complexe, on peut avoir besoin de :

- vérifier que des **modules** ou **binaires** sont présents
- vérifier que l'utilisateur a les **droits nécessaires**
- vérifier que les **paramètres sont compatibles**

Vérifier qu'un module est installé

```
if (-not (Get-Module -ListAvailable -Name "ActiveDirectory")) {
    Write-Error "Le module ActiveDirectory est requis."
    return
}
```

Gestion des dépendances et des prérequis

Vérifier les droits

```
if (-not ([Security.Principal.WindowsPrincipal] [Security.Principal.WindowsIdentity]::GetCurrent()).IsInRole([Security.Principal.WindowsBuiltinRole]::Administrator)) {
    Write-Error "Ce script doit être exécuté en tant qu'administrateur."
    return
}
```

Gestion de la sécurité et des erreurs intermodules avancées

Dans des **modules composés de plusieurs fonctions** (ex: module `.psm1`), il est important de :

- gérer les erreurs proprement entre fonctions
- ne pas faire "remonter" des erreurs imprévues à l'utilisateur final

Techniques :

- Utiliser `throw` pour générer une erreur fatale
- Propager des erreurs propres
- Centraliser la gestion des erreurs

Gestion de la sécurité et des erreurs intermodules avancées

```powershell
function Test-Connexion {
    param([string]$IP)
    if (-not (Test-Connection -ComputerName $IP -Count 1 -Quiet)) {
        throw "Impossible de joindre l'IP $IP."
    }
}
function Audit-Serveur {
    param([string]$IP)

    try {
        Test-Connexion -IP $IP
        "Audit OK pour $IP"
    }
    catch {
        Write-Warning "Erreur lors de l'audit : $_"
    }
}
```

Exploiter les points d'arrêt en détail

- **déboguer** ses scripts
- poser des **points d'arrêt conditionnels**
- examiner les variables à l'exécution
- Avec `Set-PSBreakpoint`.

Exemple :

```
Set-PSBreakpoint -Script "monScript.ps1" -Line 10
```

Points d'arrêt conditionnels :

```
Set-PSBreakpoint -Script "monScript.ps1" -Line 15 -Condition '$var -gt 100'
```

Gérer les boucles avancées

- **ForEach-Object avec -Parallel (PowerShell 7+)**

```
$IPs = "192.168.1.1", "192.168.1.2", "192.168.1.3"

$IPs | ForEach-Object -Parallel {
    Test-Connection -ComputerName $_ -Count 1
}
```

Gérer les boucles avancées

Do-While et Do-Until

Boucles contrôlées :

```
$count = 0
do {
    Write-Output "Compteur : $count"
    $count++
} while ($count -lt 5)
```

Gérer les exécutions

- le **contrôle de flux** d'un script complexe
- les **étapes d'exécution conditionnelles**
- les **retours d'état**

Gérer les exécutions

Permet de quitter une fonction en retournant un objet ou une valeur.

```
function Test-Valeur {
    param([int]$x)

    if ($x -lt 0) {
        return "Valeur négative"
    }

    return "Valeur positive ou nulle"
}
```

Gérer les exécutions

- **Break et Continue (en boucle)**

```
foreach ($i in 1..10) {
    if ($i -eq 5) { break } # Quitte la boucle
    Write-Output $i
}
```

```
foreach ($i in 1..10) {
    if ($i % 2 -eq 0) { continue } # Ignore les nombres pairs
    Write-Output $i
}
```

Vue d'ensemble de la sécurité au sein de PowerShell

- Powershell permet d'administrer un système, de manipuler l'AD, les certificats, les réseaux…
- Powershell est donc **soumis à des contrôles de sécurité** stricts (pour éviter des usages malveillants).
  - Le **policy model** de PowerShell
  - La **gestion de la signature des scripts**
  - Le **contexte d'exécution**
  - Les protections contre l'injection ou l'exécution non autorisée

Les composants de la sécurité PowerShell

- **Execution Policy**

L'Execution Policy détermine si un script peut être exécuté ou non.
Ce n'est pas un vrai "contrôle d'accès", mais une **protection contre l'exécution involontaire de scripts non signés**.

Niveau	Description
Restricted	Aucune exécution de scripts autorisée
AllSigned	Seuls les scripts signés peuvent s'exécuter
RemoteSigned	Les scripts distants doivent être signés
Unrestricted	Tous les scripts peuvent s'exécuter avec avertissement
Bypass	Aucun contrôle

Les composants de la sécurité PowerShell

- **Voir la politique en place**

```
Get-ExecutionPolicy -List
```

- **Modifier temporairement**

```
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope Process
```

Signature de scripts

Quand on est dans un environnement contrôlé (AD, entreprise), il est recommandé de **signer les scripts**.
Pourquoi ?

- Cela garantit l'**intégrité** du script
- Cela permet d'identifier l'auteur
- Cela permet aux politiques de sécurité d'autoriser le script

Signer un script

1. Il faut un certificat de signature de code
2. On signe avec `Set-AuthenticodeSignature`

Exemple :

```
$cert = Get-ChildItem -Path Cert:\CurrentUser\My -CodeSigningCert
Set-AuthenticodeSignature -FilePath "MonScript.ps1" -Certificate $cert
```

Le script portera ensuite une signature vérifiable.

Gestion des accès, des secrets et des certifications

Quand on automatise :

- des connexions à des services
- des accès à des bases de données
- des appels API

Il est essentiel de ne **pas stocker les secrets en clair** dans les scripts.

Options disponibles en PowerShell

- **SecureString**

Ancienne méthode, un peu dépassée mais encore utilisée.

```
$secret = Read-Host -AsSecureString
```

Problème : dépend du poste local.

Options disponibles en PowerShell

- **PSCredential**

Objet contenant login + SecureString.

```
$cred = Get-Credential
```

Utilisé pour passer des informations de connexion aux cmdlets :

```
Invoke-Command -ComputerName "SRV01" -Credential $cred -ScriptBlock { hostname }
```

Options disponibles en PowerShell

- **SecretManagement Module (moderne, recommandé)**

Depuis PowerShell 7, Microsoft a introduit un **module officiel SecretManagement** pour gérer les secrets de façon centralisée.

Installation :

```
Install-Module -Name Microsoft.PowerShell.SecretManagement -Repository PSGallery
```

Utilisation :

```
# Enregistrer un coffre (ex : Windows Credential Manager, Azure Key Vault)
Register-SecretVault -Name "MonVault" -ModuleName Microsoft.PowerShell.SecretStore
# Stocker un secret
Set-Secret -Name "MonMDP" -Secret "MonSuperMotDePasse"
# Récupérer un secret
$mdp = Get-Secret -Name "MonMDP"
```

Avantages :

- Centralisation
- Intégration avec Key Vault, KeePass, Vault de Hashicorp…
- Pas de dépendance locale à un poste spécifique

Gestion des certificats

PowerShell permet aussi de gérer les certificats stockés :

```
Get-ChildItem -Path Cert:\CurrentUser\My
```

On peut utiliser ces certificats pour :

- signer les scripts
- établir des connexions TLS sécurisées
- authentifier des API

Mise en œuvre au sein d'un parc

Dans un parc d'entreprise :

- On doit s'assurer que les postes clients respectent les politiques de sécurité.
- On veut que les scripts PowerShell ne soient pas une **source de vulnérabilité**.

Mise en œuvre au sein d'un parc

Stratégies possibles

1. **Définir l'ExecutionPolicy par GPO**

On peut **déployer l'ExecutionPolicy** au niveau d'une OU Active Directory via GPO.

Mise en œuvre au sein d'un parc

2. **Imposer la signature des scripts**

Mettre `AllSigned` ou `RemoteSigned` :

- Cela force l'usage de certificats valides
- Les scripts doivent être validés et maîtrisés

Mise en œuvre au sein d'un parc

3. **Déployer un coffre de secrets d'entreprise**

Par exemple :

- Azure Key Vault
- Hashicorp Vault
- Windows Credential Manager (poste à poste, pas recommandé en centralisé)

Et intégrer les scripts avec le **SecretManagement module**.

Mise en œuvre au sein d'un parc

4. **Automatiser la rotation des secrets**

Les scripts ne doivent **jamais contenir de mots de passe en dur**.
Utiliser des stratégies :

- rotation automatique des clés
- récupération des secrets au runtime via API sécurisée

Mise en œuvre au sein d'un parc

Dans un vrai environnement pro :

1. Politique GPO → ExecutionPolicy `RemoteSigned`
2. Scripts signés avec certificat interne
3. Stockage des secrets dans Azure Key Vault
4. Modules PowerShell utilisés :

```
Import-Module Microsoft.PowerShell.SecretManagement
Import-Module Az.KeyVault
```

Mise en œuvre au sein d'un parc

5. Récupération des secrets dans le script :

```
Connect-AzAccount
$secret = Get-AzKeyVaultSecret -VaultName "MonVault" -Name "SQLPassword"
```

6. Les scripts exécutés avec compte de service restreint et journalisation :

```
Start-Transcript -Path "audit.log"
# Exécution des actions
Stop-Transcript
```

Desired State Configuration (DSC)

DSC est un **mécanisme de gestion de configuration** intégré dans PowerShell.

- Il permet de **décrire l'état souhaité** d'un système : par exemple "tel service doit être en cours d'exécution", "tel fichier doit exister", "tel rôle Windows doit être installé", etc.

DSC va ensuite :

- **Appliquer cet état** (mettre le système en conformité)
- **Vérifier régulièrement** que cet état est maintenu (conformité dans le temps)

Desired State Configuration (DSC)

- **Les composants de DSC**
  - Une **configuration déclarative** (tu décris ce que tu veux)
  - Une **compilation en fichier MOF** (Managed Object Format)
  - Une **application par un moteur local** (LCM - Local Configuration Manager)
- **Architecture DSC :**

```
[ Configuration ] --> [ Fichier MOF ] --> [ LCM applique ] --> [ Système configuré ]
```

Desired State Configuration (DSC)

- **Modes d'utilisation DSC**
  - **Push mode** : on envoie manuellement la configuration sur la machine
  - **Pull mode** : la machine va chercher automatiquement la config sur un serveur Pull DSC (en entreprise)

Desired State Configuration (DSC)

- **Syntaxe de base d'une configuration DSC**

```
# Exemple simple DSC : s'assurer que le service Print Spooler est démarré

Configuration MonDSC {
    Node "localhost" {
        Service PrintSpooler {
            Name = "Spooler"
            StartupType = "Automatic"
            State = "Running"
        }
    }
}
# On génère le fichier MOF
MonDSC
```

Desired State Configuration (DSC)

- **Application de la configuration**

```
# Application manuelle de la config sur la machine
Start-DscConfiguration -Path ./MonDSC -Wait -Verbose -Force
```

Explications :

- `-Path` = dossier contenant les fichiers MOF
- `-Wait` = attendre la fin de l'application
- `-Verbose` = afficher les détails
- `-Force` = forcer l'application même si déjà existante

Desired State Configuration (DSC)

- **Contrôle de conformité**

```
# Vérifier si la machine est conforme à la config
Test-DscConfiguration -Verbose
```

Retourne `True` ou `False`.

Desired State Configuration (DSC)

- **Mise en œuvre d'une configuration de conformité avancée**

```
Configuration ServeurWeb {
    Node "localhost" {
        WindowsFeature IIS {
            Name = "Web-Server"
            Ensure = "Present"
        }
        Service W3SVC {
            Name = "W3SVC"
            StartupType = "Automatic"
            State = "Running"
            DependsOn = "[WindowsFeature]IIS"
        }
    }
}
# Générer le fichier MOF
ServeurWeb
```

Les workflows dans DSC

Dans PowerShell DSC, un **workflow** permet d'exécuter des étapes de configuration de manière contrôlée :

- en parallèle
- en séquence
- avec gestion de redémarrage si besoin

Mais attention : dans PowerShell 7, les "workflow" PowerShell au sens du mot-clé `workflow` ne sont plus supportés.

- On utilise plutôt les **composants DSC** + gestion de dépendances (`DependsOn`) + orchestration externe.**

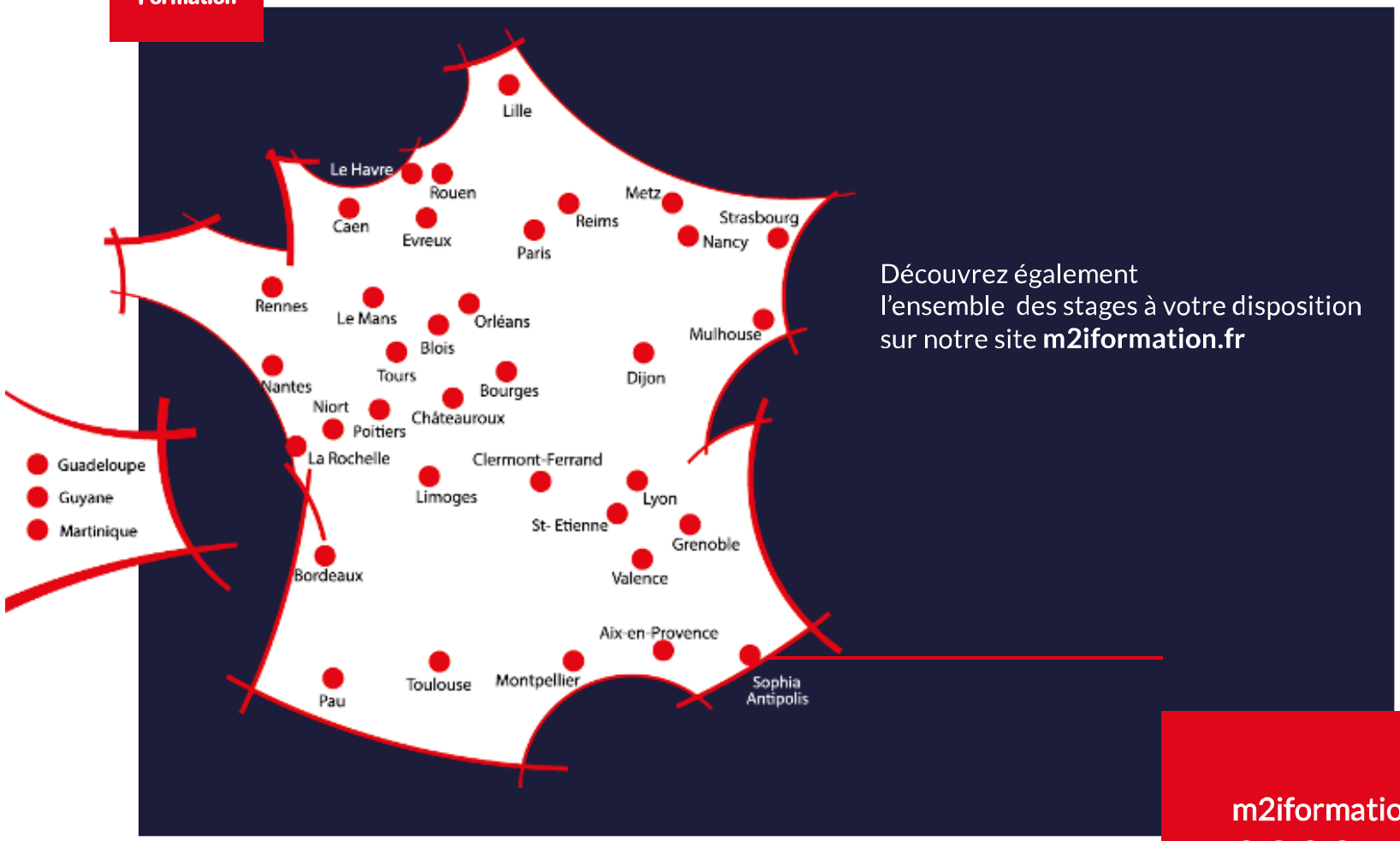
Mise en œuvre d'une configuration enchaînant plusieurs étapes

```
Configuration ExempleMultiEtapes {
    Node "localhost" {
        File FichierTest {
            DestinationPath = "C:\Temp\test.txt"
            Contents = "Hello DSC!"
            Ensure = "Present"
        }

        Service MonService {
            Name = "Spooler"
            State = "Running"
            DependsOn = "[File]FichierTest"
        }
    }
```

Mise en œuvre d'une configuration enchaînant plusieurs étapes

```
# Générer et appliquer
ExempleMultiEtapes
Start-DscConfiguration -Path ./ExempleMultiEtapes -Wait -Verbose -Force
```



Découvrez également
l'ensemble des stages à votre disposition
sur notre site **m2iformation.fr**

m2iformation.fr