Documentation

Archiver des dossiers et des fichiers à l’aide des scripts

BET--TAFFARD Océane

25/11/24

Version 2

TABLE DES MATIERES

Introduction 2

Explications 3

Fonctionnement du script d’archivage 3

Fonctionnement du script de planification de tâche 4

INTRODUCTION

Afin de pouvoir archiver des dossiers et fichiers qui n'ont pas été modifiés depuis un certain temps, deux scripts ont été créés.

Dans le premier script a été défini le seuil au bout duquel on souhaite que les fichiers non modifiés soient archivés. Lorsqu'il s'exécute, le script regarde la date de la dernière modification des dossiers/fichiers et archive ceux dépassant le seuil en conservant les droits d'accès (les administrateurs gardent le contrôle total mais les utilisateurs passent en lecture et exécution seulement).

Le deuxième, quant à lui, sert à créer une tâche afin d'automatiser l'archivage qui s'activera chaque fois qu'il lui a été demandé (par exemple, tous les premiers dimanches du mois à 15h). Elle exécutera alors le premier script.

EXPLICATIONS

Fonctionnement du script d'archivage

Le script d'archivage a plusieurs fonctions : le déplacement des fichiers dépassant le seuil défini dans un serveur d'archive, la conservation des droits d'accès et la restriction des droits des utilisateurs à lecture et exécution seulement.

Étant donné que ce script peut être emmené à être utilisé dans plusieurs environnements différents, il y a quelques parties de son code qui peuvent nécessiter d'être modifiées :

```
param(
    [string]$sourceFolder = "$env:SystemDrive\Donnees", # Chemin vers le dossier source
    [string]$archiveFolder = "\\192.168.1.202\archives", # Chemin vers le dossier d'archivage
    [string]$logFile = "C:\Logs\archive_log.txt", # Chemin vers le fichier de log
    [string]$errorLogFile = "C:\Logs\archive_error_log.txt", # Chemin vers le fichier de log des erreurs
    [int]$daysThreshold = 700 # Seuil en jours
)
```

Le premier paramètre « $env:SystemDrive\Donnees » correspond au chemin vers le dossier source, c'est-à-dire le dossier à partir duquel les dossiers sont récupérés pour être archivés. Il faudra donc remplacer ce chemin par le vôtre (en le mettant bien entre guillemets).

Le deuxième paramètre « \\192.168.1.202\archives » correspond au chemin du dossier d'archivage, donc le dossier dans lequel seront archivés les dossiers récupérés dans le dossier source. De même, il faudra remplacer ce chemin par le vôtre.

Le troisième paramètre « C:\Logs\archive_log.txt » correspond au chemin du fichier de logs (fichier dans lequel est écrit tout ce qui passe lors de l'exécution du script). Il ne nécessite pas obligatoirement d'être modifié étant donné que le script se charge lui-même de créer le fichier mais vous pouvez quand même le faire si vous souhaitez qu'il s'enregistre ailleurs.

Le quatrième paramètre « C:\Logs\archive_error_log.txt» correspond au chemin du fichier de log d'erreurs (fichier dans lequel sont recensées toutes les erreurs qu'il a pu y avoir lors de l'exécution du script). De même que pour le précédent, il ne nécessite pas obligatoirement d'être changé mais vous pouvez quand même le faire si vous souhaitez qu'il s'enregistre ailleurs.

Le dernier paramètre « $daysThreshold » correspond au seuil au bout duquel les dossiers et fichiers non modifiés doivent être archivés. Pour fixer un autre seuil il suffit simplement de changer le « 700 ».

```
# Créer le dossier de logs s'il n'existe pas
$logPath = Split-Path -Parent $logFile
if (-not (Test-Path $logPath)) {
    New-Item -ItemType Directory -Path $logPath -Force
}
```

Ce morceau de code vérifie si le dossier de Logs existe et, si tel n'est pas le cas, le crée.

```
# Fichier de log pour cette exécution
$timestamp = Get-Date -Format "dd-MM-yyyy-HH-mm"
$executionLogFile = "C:\Logs\archive_log_$timestamp.txt"
$executionErrorLogFile = "C:\Logs\archive_error_log_$timestamp.txt"
```

Cette partie du code sert à récupérer la date et l'heure actuelles, à définir les chemins des fichiers et à créer leurs noms en y incluant la date et l'heure.

```
# Fonction de log
function Write-Log {
    param($Message, [switch]$IsError)
    $logMessage = "$(Get-Date -Format 'yyyy-MM-dd HH:mm:ss'): $Message"
    Add-Content -Path $executionLogFile -Value $logMessage
    Write-Host $logMessage
    if ($IsError) {
        Add-Content -Path $executionErrorLogFile -Value $logMessage
    }
}
```

Cette fonction sert à écrire les messages de log, avec la date et l'heure, dans les deux fichiers de logs.

```powershell
# Fonction pour obtenir les permissions d'un chemin
function Get-PathPermissions {
    param ([string]$Path)
    $permissions = @{}
    try {
        $acl = Get-Acl -Path $Path
        if ($null -ne $acl) {
            foreach ($access in $acl.Access) {
                $identity = $access.IdentityReference.Value
                if (!$permissions.ContainsKey($identity)) {
                    $permissions[$identity] = $access.FileSystemRights
                } else {
                    $permissions[$identity] = $permissions[$identity] -bor $access.FileSystemRights
                }
            }
        }
        return $permissions
    } catch {
        Write-Log "Erreur lors de la récupération des permissions pour $Path : $_" -IsError
        throw
    }
}
```

Cette fonction permet de récupérer les permissions assignées aux dossiers et fichiers afin de les préserver lors du déplacement vers le serveur d'archives.

```powershell
# Fonction pour vérifier si un utilisateur est administrateur
function Is-Admin {
    param ([string]$Identity)
    return $Identity -match "Administrateurs|Administrateur|Administrators|Système|Domain Admins"
}
```

Cette fonction sert à vérifier si l'utilisateur est un administrateur afin de, si tel est le cas, ne pas lui enlever le contrôle total sur les dossiers et fichiers archivés. Elle peut nécessiter d'être modifiée ou complétée si vous avez un administrateur spécifique qui n'apparaît pas dans cette liste. Dans ce cas-là il faut séparer chaque nom d'administrateur par « | ».

```powershell
# Fonction pour appliquer les permissions sur un dossier
function Set-FolderPermissions {
    param ([string]$Path, [hashtable]$Permissions)
    try {
        $acl = Get-Acl -Path $Path
        if ($null -ne $acl) {
            $acl.SetAccessRuleProtection($true, $false)
            $acl.Access | ForEach-Object { $acl.RemoveAccessRule($_) | Out-Null }
            foreach ($identity in $Permissions.Keys) {
                $rights = $Permissions[$identity]
                if (-not (Is-Admin -Identity $identity)) {
                    $rights = [System.Security.AccessControl.FileSystemRights]::ReadAndExecute
                }
                $rule = New-Object System.Security.AccessControl.FileSystemAccessRule(
                    $identity,
                    $rights,
                    "ContainerInherit,ObjectInherit",
                    "None",
                    [System.Security.AccessControl.AccessControlType]::Allow
                )
                $acl.AddAccessRule($rule)
            }
            Set-Acl -Path $Path -AclObject $acl
            Write-Log "Permissions appliquées sur le dossier : $Path"
        }
    } catch {
        Write-Log "Erreur lors de l'application des permissions sur le dossier : $_" -IsError
        throw
    }
}
```

Cette fonction attribue les droits d'accès à chaque dossier lorsqu'ils sont déplacés en vérifiant si l'utilisateur à qui il faut mettre les permissions est un administrateur ou un simple utilisateur.

```powershell
# Fonction pour appliquer les permissions sur un fichier
function Set-FilePermissions {
    param ([string]$Path, [hashtable]$Permissions)
    try {
        $acl = Get-Acl -Path $Path
        if ($null -ne $acl) {
            $acl.SetAccessRuleProtection($true, $false)
            $acl.Access | ForEach-Object { $acl.RemoveAccessRule($_) | Out-Null }
            foreach ($identity in $Permissions.Keys) {
                $rights = $Permissions[$identity]
                if (-not (Is-Admin -Identity $identity)) {
                    $rights = [System.Security.AccessControl.FileSystemRights]::ReadAndExecute
                }
                $rule = New-Object System.Security.AccessControl.FileSystemAccessRule(
                    $identity,
                    $rights,
                    "None",
                    "None",
                    [System.Security.AccessControl.AccessControlType]::Allow
                )
                $acl.AddAccessRule($rule)
            }
            Set-Acl -Path $Path -AclObject $acl
            try {
                Set-ItemProperty -Path $Path -Name IsReadOnly -Value $true
                Write-Log "Propriété IsReadOnly appliquée : $Path"
            } catch {
                Write-Log "Erreur lors de la modification de la propriété IsReadOnly pour le fichier : $Path - $_" -IsError
            }
            Write-Log "Permissions appliquées sur le fichier : $Path"
        }
    } catch {
        Write-Log "Erreur lors de l'application des permissions sur le fichier : $_" -IsError
        throw
    }
}
```

Cette fonction fait la même chose que la première mais cette fois sur les fichiers. Elle vérifie si l'identité à qui il faut assigner les droits est un utilisateur ou un administrateur et les applique en fonction.

```powershell
# Fonction pour fusionner les permissions de deux hashtables
function Merge-Permissions {
    param ([hashtable]$Permissions1, [hashtable]$Permissions2)
    $mergedPermissions = @{}
    $allIdentities = @($Permissions1.Keys) + @($Permissions2.Keys) | Select-Object -Unique
    foreach ($identity in $allIdentities) {
        if ($Permissions1.ContainsKey($identity) -and $Permissions2.ContainsKey($identity)) {
            $mergedPermissions[$identity] = $Permissions1[$identity] -bor $Permissions2[$identity]
        } elseif ($Permissions1.ContainsKey($identity)) {
            $mergedPermissions[$identity] = $Permissions1[$identity]
        } else {
            $mergedPermissions[$identity] = $Permissions2[$identity]
        }
    }
    return $mergedPermissions
}
```

Cette fonction est utilisée pour combiner les permissions du fichier source et du fichier de destination lors de la copie, assurant que tous les droits originaux sont préservés.

```powershell
# Fonction pour l'archivage des fichiers
function Start-FileArchiving {
    Write-Log "Début du processus d'archivage"

    # Vérification du dossier source
    if (-not (Test-Path $sourceFolder)) {
        Write-Log "Erreur: Dossier source inaccessible ou inexistant : $sourceFolder" -IsError
        return
    }
    Write-Log "Dossier source trouvé : $sourceFolder"

    # Vérification/création du dossier d'archive
    if (-not (Test-Path $archiveFolder)) {
        try {
            New-Item -ItemType Directory -Path $archiveFolder -Force | Out-Null
            Write-Log "Dossier d'archive créé : $archiveFolder"
        } catch {
            Write-Log "Erreur lors de la création du dossier d'archive : $archiveFolder - $_" -IsError
            return
        }
    }

    try {
        # Récupération de TOUS les fichiers dans le dossier source
        $filesToMove = Get-ChildItem -Path $sourceFolder -Recurse -File -Force |
            Where-Object { $_.LastWriteTime -lt (Get-Date).AddDays(-$daysThreshold) }

        $totalFiles = $filesToMove.Count
        Write-Log "Nombre total de fichiers à traiter : $totalFiles"

        $processedFiles = 0
        foreach ($file in $filesToMove) {
            $processedFiles++
            Write-Log "[$processedFiles/$totalFiles] Traitement du fichier : $($file.FullName)"

            # Calcul du chemin relatif pour préserver la structure
            $relativePath = $file.FullName.Substring($sourceFolder.Length)
            $destinationPath = Join-Path $archiveFolder $relativePath
            $destinationDir = Split-Path -Parent $destinationPath
```

```
            try {
                $sourcePermissions = Get-PathPermissions -Path $file.FullName
                if ($null -ne $sourcePermissions) {
                    if (-not (Test-Path $destinationDir)) {
                        New-Item -ItemType Directory -Path $destinationDir -Force | Out-Null
                        Write-Log "Dossier de destination créé : $destinationDir"
                    }

                    $existingPermissions = @{}
                    if (Test-Path $destinationPath) {
                        $existingPermissions = Get-PathPermissions -Path $destinationPath
                    }

                    $mergedPermissions = Merge-Permissions -Permissions1 $sourcePermissions -Permissions2 $existingPermissions

                    # Application des permissions sur le dossier de destination
                    Set-FolderPermissions -Path $destinationDir -Permissions $mergedPermissions

                    # Copie du fichier
                    Copy-Item -Path $file.FullName -Destination $destinationPath -Force
                    Write-Log "Fichier copié : $destinationPath"

                    # Application des permissions sur le fichier
                    Set-FilePermissions -Path $destinationPath -Permissions $mergedPermissions

                    # Suppression du fichier source
                    Remove-Item -Path $file.FullName -Force
                    Write-Log "Fichier source supprimé : $($file.FullName)"
                } else {
                    Write-Log "Aucune permission trouvée pour le fichier : $($file.FullName)" -IsError
                }
            } catch {
                Write-Log "Erreur lors du traitement du fichier : $_" -IsError
            }
        }

        Write-Log "Traitement terminé. $processedFiles fichiers traités sur $totalFiles"
    } catch {
        Write-Log "Erreur lors du processus d'archivage : $_" -IsError
    }
}
```

Cette fonction sert à archiver les fichiers, pour cela elle procède en plusieurs étapes :

- Elle déplace automatiquement les fichiers ayant dépassés le seuil ;
- Copie les fichiers dans un dossier d'archive en préservant leur structure originale ;
- Gère et conserve les permissions des fichiers ;
- Supprime les fichiers originaux après archivage ;
- Enregistre et trace toutes les étapes du processus.

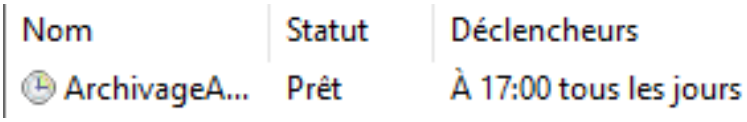
```
# Exécution du script
Write-Log "=== Démarrage du script ==="
Write-Log "Version PowerShell : $($PSVersionTable.PSVersion)"
Write-Log "Utilisateur actuel : $([System.Security.Principal.WindowsIdentity]::GetCurrent().Name)"
Write-Log "Dossier source configuré : $sourceFolder"
Write-Log "Dossier d'archive configuré : $archiveFolder"
Write-Log "Seuil d'archivage : $daysThreshold jours"
Start-FileArchiving
Write-Log "=== Fin du script ==="
```

Ce dernier morceau de code sert à afficher certaines informations avant l'exécution du script tel que la version PowerShell utilisée, l'utilisateur qui exécute le script ainsi que les configurations du dossier source, du dossier d'archives et du seuil d'archivage. Il lance ensuite le processus d'archivage tout en enregistrant les détails de l'exécution.

Fonctionnement du script de planification de tâche

Le script de planification de tâche permet de créer une tâche qui exécutera le script d'archivage automatiquement. Cela évite d'avoir à exécuter le script d'archivage manuellement à chaque fois puisque la tâche a été programmée pour le faire de manière régulière. Ce script n'a besoin d'être activé qu'une seule fois à moins qu'il ait était modifié, auquel cas il faudra le réexécuter afin qu'il prenne les modifications en compte.

/ ! \ Important / ! \



Lorsque le script a été exécuté il faut aller vérifier dans le planificateur de tâche que la tâche a bien été créée et qu'elle a pour statut « Prêt » ou « En cours », si elle a pour statut « Désactivé » il faut faire un clic droit dessus et cliquer sur « Activer » (il peut être nécessaire de fermer et rouvrir le planificateur de tâche pour qu'il se mette à jour).

```
# Suppression de l'ancienne tâche
Unregister-ScheduledTask -TaskName "ArchivageAutomatique" -Confirm:$false -ErrorAction SilentlyContinue
```

Cette ligne de code située tout en haut du script sert à supprimer la tâche nommée « ArchivageAutomatique » avant d'en créer une nouvelle. Cela permet d'éviter, lorsque l'on réexécute le script parce qu'il a été modifié, d'avoir deux fois la même tâche et de s'assurer que la nouvelle tâche remplace correctement l'ancienne.

```powershell
# Vérification de l'existence du dossier Logs
if (-not (Test-Path "C:\Scripts\Logs")) {
    New-Item -ItemType Directory -Force -Path "C:\Scripts\Logs"
}
```

Ce morceau de code vérifie si le dossier « Logs » existe dans le dossier « Scripts » et, si tel n'est pas le cas, le crée. Si vous souhaitez que le dossier de logs s'enregistre ailleurs, il suffit de changer le chemin « C:\Scripts\Logs ».

```powershell
# Création du script wrapper
$wrapperContent = @'
Start-Transcript -Path "C:\Scripts\Logs\log_archivage_$(Get-Date -Format 'dd-MM-yy
Write-Host "Début de l'exécution : $(Get-Date)"
& "C:\Scripts\ArchivageAutoTest.ps1"
Write-Host "Fin de l'exécution : $(Get-Date)"
Stop-Transcript
'@
$wrapperContent | Out-File "C:\Scripts\wrapper.ps1" -Force
```

Ce morceau de code crée un script « wrapper » qui enregistre toute l'exécution dans un fichier de log, indique à quelle date et à quelle heure le script à démarré et s'est fini et exécute le script principal « ArchivageAutoTest.ps1 ».

```powershell
# Création de la tâche
$action = New-ScheduledTaskAction -Execute "C:\Windows\System32\WindowsPowerShell
    -Argument '-NoProfile -ExecutionPolicy Bypass -File "C:\Scripts\wrapper.ps1"'
$trigger = New-ScheduledTaskTrigger `
    -Daily `
    -At "17:00"
$settings = New-ScheduledTaskSettingsSet `
    -AllowStartIfOnBatteries `
    -DontStopIfGoingOnBatteries `
    -StartWhenAvailable `
    -ExecutionTimeLimit (New-TimeSpan -HOURS 24) `
    -MultipleInstances IgnoreNew
$principal = New-ScheduledTaskPrincipal `
    -UserId "Administrateur" `
    -RunLevel Highest
```

Ce morceau de code crée la tâche à exécuter et configure notamment :

- Quand elle doit s'exécuter (ici elle a été configurée pour s'exécuter tous les jours à 17h00)
- Les paramètres avancés comme l'exécution sur batterie et la continuité de l'exécution si passage sur batterie, l'exécution dès que possible si manqué et la limitation à 24h
- Qui exécute la tâche et avec quels droits (ici c'est l'administrateur qui exécute la tâche avec les droits administrateurs maximum, cela signifie

que seuls ceux ayant les droits administrateurs peuvent modifier, supprimer ou changer la configuration de la tâche).

```
# Création de la tâche avec Force pour écraser si elle existe déjà
$task = Register-ScheduledTask `
    -TaskName "ArchivageAutomatique" `
    -Action $action `
    -Trigger $trigger `
    -Principal $principal `
    -Settings $settings `
    -Force
```

Ce morceau de code permet de créer la tâche dans le planificateur Windows avec toutes les configurations précédentes en utilisant l'option -Force afin d'écraser les tâches du même nom dans le cas où il en existerait.

```
# Désactiver la tâche en modifiant ses propriétés
$task.Settings.Enabled = $true
$task | Set-ScheduledTask
```

Ces lignes de code situées tout à la fin du script servent à activer la tâche et à la mettre à jour chaque fois que le script est exécuté. Si vous souhaitez la mettre à jour sans toutefois l'activer vous pouvez passer la première ligne à « $false », cela mettra la tâche à jour dans le planificateur Windows mais son statut passera à « Désactivé ».