Manipulation de fichiers en PowerShell

Partie 1 : Écrire dans des fichiers avec PowerShell

Écriture dans les fichiers texte

1.1 – Créer et écrire dans un fichier

Pour écrire une chaîne dans un fichier (et l'écraser s'il existe déjà) :

```
"Bonjour le monde" > fichier.txt
```

Remarque : cela crée le fichier `fichier.txt` s'il n'existe pas, ou le remplace complètement.

—

1.2 – Ajouter du texte à un fichier existant

Pour ajouter du contenu sans effacer ce qui existe déjà :

```
"Nouvelle ligne" >> fichier.txt
```

Le symbole `>>` signifie append (ajouter à la fin).

—

1.3 – Utiliser `Out-File`

`Out-File` est une cmdlet plus flexible :

```
"Texte avec Out-File" | Out-File -FilePath fichier.txt
```

`-Append` permet d'ajouter à la fin :

```
"Ajout avec Out-File" | Out-File -FilePath fichier.txt -Append
```

—

1.4 – Écrire plusieurs lignes

```
@(
  "Ligne 1"
  "Ligne 2"
  "Ligne 3"
) | Out-File -FilePath fichier.txt
```

—

1.5 – Exemple pratique

Créer un journal d'activité avec date et heure :

```
$date = Get-Date -Format "yyyy-MM-dd HH:mm:ss"
"[$date] D but du traitement" >> journal.log
```

—

1.6 – Redirection standard avancée

```
Get-Process > processus.txt 2> erreurs.log
```

- `>` : redirige la sortie standard.
- `2>` : redirige les erreurs.
- `*>` : redirige tout (PowerShell 3+).

Partie 2 : Lire des fichiers avec PowerShell

Lecture de fichiers texte

2.1 – Lire tout le contenu d'un fichier

```
Get-Content -Path fichier.txt
```

Remarque : cette commande retourne une collection de lignes (chaque ligne est une chaîne).

—

2.2 – Lire ligne par ligne

Pour parcourir le contenu ligne par ligne dans une boucle :

```
foreach ($ligne in Get-Content fichier.txt) {
    Write-Output "Ligne lue : $ligne"
}
```

—

2.3 – Lire seulement les premières ou dernières lignes

```
Get-Content fichier.txt -TotalCount 5      # Les 5 premi res lignes
Get-Content fichier.txt | Select-Object -Last 3   # Les 3 derni res
```

—

2.4 – Rechercher du contenu dans un fichier

```
Select-String -Path fichier.txt -Pattern "erreur"
```

Select-String fonctionne comme `grep` : il affiche les lignes contenant le motif.

—

2.5 – Lire un fichier et le convertir en une seule chaîne

```
$texte = Get-Content fichier.txt -Raw
```

`-Raw` permet de récupérer tout le contenu du fichier en une seule variable de type chaîne, utile pour traiter des blocs complets.

—

2.6 – Exemple d'analyse de log

```
$erreurs = Select-String -Path journal.log -Pattern "ERREUR"
foreach ($e in $erreurs) {
    Write-Output "[$($e.LineNumber)] $($e.Line)"
}
```

Remarque : chaque objet retourné par `Select-String` contient la ligne, le numéro de ligne, etc.

Partie 3 : Manipuler les fichiers avec .NET dans PowerShell

Accès bas niveau via .NET

PowerShell étant basé sur .NET, on peut accéder directement aux classes de l'espace de noms `System.IO`, ce qui permet une manipulation fine des fichiers : création, lecture, écriture, suppression, etc.

—

3.1 – Écrire dans un fichier avec `System.IO.File`

```
[System.IO.File]::WriteAllText("log.txt", "Contenu  crit  via .NET")
```

Écrase le fichier s'il existe. Utilise `WriteAllLines` pour plusieurs lignes :

```
[System.IO.File]::WriteAllLines("log.txt", @("Ligne 1", "Ligne 2"))
```

—

3.2 – Ajouter à un fichier

```
[System.IO.File]::AppendAllText("log.txt", "Ligne ajout e `n")
```

—

3.3 – Lire un fichier entier

```
$contenu = [System.IO.File]::ReadAllText("log.txt")
Write-Output $contenu
```

Ou ligne par ligne :

```
$lignes = [System.IO.File]::ReadAllLines("log.txt")
foreach ($ligne in $lignes) {
    Write-Output $ligne
}
```

—

3.4 – Utiliser `StreamWriter` pour plus de contrôle

```
$sw = [System.IO.StreamWriter]::new("sortie.txt", $true)
$sw.WriteLine("Texte 1")
$sw.WriteLine("Texte 2")
$sw.Close()
```

`new("chemin", `*true*`)` : *leparamètre*`true` signifie append (ajouter à la fin).

3.5 – Utiliser `StreamReader` pour lecture ligne par ligne

```
$sr = [System.IO.StreamReader]::new("log.txt")
while (-not $sr.EndOfStream) {
    $ligne = $sr.ReadLine()
    Write-Output $ligne
}
$sr.Close()
```

3.6 – Supprimer, copier, déplacer un fichier

```
[System.IO.File]::Delete("ancien.txt")
[System.IO.File]::Copy("source.txt", "copie.txt", $true)
[System.IO.File]::Move("copie.txt", "archives/final.txt")
```

3.7 – Vérifier l'existence d'un fichier

```
if ([System.IO.File]::Exists("fichier.txt")) {
    Write-Output "Le fichier existe."
}
```

Avantages de l'approche .NET

- Plus rapide et plus bas niveau que les cmdlets PowerShell classiques.
- Compatible avec tous les langages .NET (C, F, etc.).
- Permet d'accéder à des fonctions avancées (buffer, async, gestion d'encodage...).

Partie 4 : Manipuler des fichiers CSV, JSON et XML en PowerShell

Fichiers structurés avec PowerShell

PowerShell offre des cmdlets natives pour manipuler des fichiers structurés comme CSV, JSON et XML. Ces formats sont très utilisés pour les échanges de données, les rapports et la configuration d'applications.

—

4.1 – Fichiers CSV

Lire un fichier CSV

```
$utilisateurs = Import-Csv -Path "utilisateurs.csv"
foreach ($u in $utilisateurs) {
    Write-Output "$($u.Nom) : $($u.Email)"
}
```

Exemple de CSV :

```
Nom,Prenom,Email
Durand,Claire,claire@example.com
Martin,Luc,luc@example.com
```

—

Créer un fichier CSV

```
$data = @(
    [pscustomobject]@{ Nom="Dupont"; Email="dupont@example.com" }
    [pscustomobject]@{ Nom="Legrand"; Email="legrand@example.com" }
)
$data | Export-Csv -Path "sortie.csv" -NoTypeInformation
```

—

4.2 – Fichiers JSON

Lire un fichier JSON

```
$json = Get-Content -Path "config.json" -Raw | ConvertFrom-Json
Write-Output $json.serveur
```

Exemple JSON :

```
{
  "serveur": "srv01",
  "port": 8080
}
```

—

Créer et écrire un JSON

```
$config = @{
    serveur = "srv01"
    port = 8080
}
$json = $config | ConvertTo-Json -Depth 2
$json | Set-Content -Path "config.json"
```

4.3 – Fichiers XML

Lire un fichier XML

```
[xml]$xml = Get-Content -Path "livres.xml"
$xml.catalogue.livre | ForEach-Object {
    $_.titre
}
```

Exemple XML :

```
<catalogue>
  <livre>
    <titre>Apprendre PowerShell</titre>
  </livre>
</catalogue>
```

Créer un fichier XML

```
$xml = New-Object System.Xml.XmlDocument
$racine = $xml.CreateElement("catalogue")
$livre = $xml.CreateElement("livre")
$titre = $xml.CreateElement("titre")
$titre.InnerText = "Programmation .NET"
$livre.AppendChild($titre)
$racine.AppendChild($livre)
$xml.AppendChild($racine)
$xml.Save("livres.xml")
```

Résumé des cmdlets utiles

- `Import-Csv`, `Export-Csv`
- `ConvertTo-Json`, `ConvertFrom-Json`
- `[xml]::new()`, `XmlDocument.Save()`, `XmlDocument.SelectNodes()`

Ces outils permettent de manipuler efficacement des structures de données complexes avec très peu de code.

Partie 5 : Introduction à .NET depuis PowerShell

Comprendre .NET dans PowerShell

PowerShell repose sur le framework .NET (ou .NET Core / .NET 5+), ce qui lui donne un accès direct à l'ensemble de ses types, objets, méthodes et classes.

—

5.1 – Qu'est-ce que .NET ?

- **.NET** est une plateforme de développement multi-langages (C#, F#, VB.NET...).
- Elle fournit un environnement d'exécution appelé **CLR** (Common Language Runtime).
- Tout repose sur des **types**, **objets**, **classes**, et **méthodes**.

PowerShell exploite tout cela pour accéder à des fonctionnalités avancées via :

- des **objets .NET préexistants**,
- des **classes .NET créées à la volée**,
- des appels à des **assemblies externes**.

—

5.2 – Accéder à une classe .NET existante

Utilisation directe de classes .NET via l'opérateur '[Namespace.Class]::Méthode()' :

```
[System.Math]::Sqrt(25)             # => 5
[System.Guid]::NewGuid()            # => identifiant unique
[System.DateTime]::Now              # => date et heure actuelles
```

—

5.3 – Créer un objet .NET (instanciation)

```
$objet = New-Object System.Text.StringBuilder
$objet.Append("Hello") | Out-Null
$objet.Append(" World") | Out-Null
$objet.ToString()                        # => "Hello World"
```

Autre syntaxe possible avec '[Class]::new()' :

```
$sb = [System.Text.StringBuilder]::new()
$sb.Append("Texte") | Out-Null
$sb.ToString()
```

—

5.4 – Propriétés et méthodes d'un objet

On peut accéder aux propriétés avec le point :

```
$now = [System.DateTime]::Now
$now.Year           # => 2025
$now.DayOfWeek      # => Lundi, Mardi, etc.
```

Lister toutes les méthodes et propriétés :

```
$now | Get-Member
```

—

5.5 – Utiliser les types de base .NET

PowerShell reconnaît directement les types .NET suivants :

- `[string]`, `[int]`, `[bool]`, `[double]`
- `[datetime]`, `[array]`, `[hashtable]`

Exemple :

```
[int]$x = 42
[string]$nom = "In s"
[bool]$estValide = $true
[datetime]$date = Get-Date
```

—

5.6 – Importer une bibliothèque externe (.dll)

```
Add-Type -Path "chemin\vers\librairie.dll"
[Namespace.Classe]::M thode()
```

Ou compiler une classe C à la volée :

```
Add-Type -TypeDefinition @"
public class Bonjour {
  public static string Dire() {
    return "Bonjour depuis C#!";
  }
}
"@
[Bonjour]::Dire()
```

—

Résumé de la syntaxe .NET en PowerShell

- `[Type]` : accéder à un type .NET
- `::` : appeler une méthode ou propriété statique
- `new()` ou `New-Object` : créer une instance
- `| Get-Member` : explorer l'objet
- `Add-Type` : importer ou définir du code C