

Étape 1 – Collecter avec PowerShell

Premières commandes PowerShell, mise en forme JSON et envoi vers une API HTTP.

5TTR

 Exercice ambitieux

Objectif de l'étape

À la fin, tu dois pouvoir lancer un **seul script PowerShell** qui :

1. interroge ton PC,
2. construit un objet JSON propre,
3. l'envoi en `POST` vers une URL (l'API PHP que tu coderas à l'étape 2),
4. affiche un message clair (succès / erreur).

Pour cette étape, l'API **n'existe pas encore**. Tu vas utiliser un site de test (`https://webhook.site` ou `https://httpbin.org/post`) qui te renvoie exactement ce qu'il a reçu. Comme ça, tu peux développer ton script sans attendre le PHP.

1. Re-découvrir PowerShell – les 4 commandes-clés

PowerShell renvoie des **objets**, pas du texte. C'est très différent du Bash ou du `cmd`. Essaie ces commandes dans l'ordre, dans un terminal :

```
1 | Get-CimInstance Win32_ComputerSystem
2 | Get-CimInstance Win32_OperatingSystem
3 | Get-CimInstance Win32_Processor
4 | Get-Volume
```

Tu remarques que la sortie est tabulaire, lisible, mais surtout chaque objet a des **propriétés** (Name, Manufacturer, FreeSpace, etc.). On peut les piocher avec `Select-Object`.

```
1 | Get-Volume | Select-Object DriveLetter, FileSystemLabel, SizeRemaining, Siz
```

💡 **ASTUCE** – pour découvrir les propriétés d'un objet, ajoute `| Get-Member` ou `| Select-Object *` à la fin d'une commande. C'est ta documentation intégrée.

À toi

Trouve, pour ton propre PC, comment obtenir :

- le nom de l'ordinateur (`hostname` ou `$env:COMPUTERNAME`),
- la quantité totale de RAM en Go,
- le système d'exploitation et sa version,
- l'espace libre du disque `C:` en Go.

Note les commandes dans un fichier `commandes.md` à part. Tu vas en avoir besoin tout le long du projet.

2. Construire un objet JSON

PowerShell sait convertir directement un objet en JSON via `ConvertTo-Json`. Mais attention : si tu passes plusieurs objets bruts, le résultat sera moche. La bonne pratique est de construire **toi-même un objet** avec uniquement les champs que tu veux.

```
1 $rapport = [PSCustomObject]@{
2     hostname     = $env:COMPUTERNAME
3     timestamp    = (Get-Date).ToString("o") # format ISO 8601
4     os           = (Get-CimInstance Win32_OperatingSystem).Caption
5     ramGo       = [math]::Round((Get-CimInstance Win32_ComputerSystem).TotalRAM)
6 }
7
8 $rapport | ConvertTo-Json -Depth 5
```

⚠️ **PIÈGE CLASSIQUE** – par défaut `ConvertTo-Json` ne plonge que de 2 niveaux dans les objets imbriqués. Si tu as des sous-listes (processus, applications...), tu auras `System.Object[]` au lieu du contenu. **Toujours préciser `-Depth 5` ou plus.**

Choix à faire

Tu peux organiser le JSON de plusieurs manières. Décide maintenant laquelle tu adoptes (et tiens-t'y) :

Option A – plat

```
1 { "hostname": "PC-01", "ramGo": 16, "cpu": "Intel i5", "osCaption": "Window
```

```

1  {
2    "hostname" : "PC-01",
3    "timestamp" : "2026-05-13T10:00:00",
4    "system" : { "os" : "Windows 11", "cpu" : "Intel i5" },
5    "memory" : { "totalGo" : 16 },
6    "disks" : [ { "letter" : "C", "totalGo" : 500, "freeGo" : 120 } ]
7  }

```

L'option B est plus verbeuse mais beaucoup plus simple à exploiter côté PHP. **Choisis-la.** On l'imposera de toute manière comme contrat d'API.

3. Le contrat d'API (le plus important du projet)

Avant d'envoyer quoi que ce soit, on définit la **structure attendue** par le serveur. C'est ce qu'on appelle un *contrat*. Si tous les élèves respectent ce contrat, le serveur peut traiter n'importe quel rapport sans bug.

Voici le contrat **minimal** du tronc commun :

```

1  {
2    "hostname" : "string, obligatoire",
3    "timestamp" : "string ISO 8601, obligatoire",
4    "system" : {
5      "os" : "string",
6      "osVersion" : "string",
7      "cpu" : "string",
8      "cpuCores" : "number"
9    },
10   "memory" : {
11     "totalGo" : "number",
12     "freeGo" : "number"
13   },
14   "disks" : [
15     { "letter" : "C", "totalGo" : 500, "freeGo" : 120 }
16   ],
17   "topProcesses" : [
18     { "name" : "chrome", "cpu" : 12.5, "memoryMo" : 850 }
19   ],
20   "applications" : [

```

```

21     { "name" : "VS Code", "version" : "1.95.0", "publisher" : "Microsoft" }
22   ],
23   "recentDownloads" : [
24     { "name" : "setup.exe", "sizeMo" : 45.2, "downloadedAt" : "2026-05-12T14:3
25   ],
26   "module" : {
27     "name" : "network|security|users|...",
28     "data" : { /* dépend du module */ }
29   }
30 }

```

Le champ `module` est laissé vide pour cette première étape. On le remplira au chapitre 5.

Travailler avec l'IA – Étape 1

Quand l'utiliser ?

- Tu connais la commande `Get-Process` mais tu ne sais pas comment filtrer les 10 plus gros consommateurs CPU.
- Tu n'arrives pas à construire ton `[PSCustomObject]` avec des sous-objets.

Prompt-type efficace :

Je suis débutant en PowerShell. Je veux récupérer les **10 PROCESSUS QUI CONSOMMENT LE PLUS DE CPU** (en %) sur Windows. Donne-moi la commande, explique ce que fait chaque pipeline, et montre comment garder uniquement les colonnes `Name`, `CPU` et la mémoire en Mo. Pas besoin de fonction réutilisable, juste l'expression.

Ce qu'il ne faut pas faire :

- Coller la sortie complète de l'IA dans ton script sans la tester ligne par ligne.
- Demander « écris-moi tout le script qui fait l'inventaire » : tu auras 150 lignes que tu ne maîtrises pas.

Vérification :

Pour chaque commande proposée, exécute-la seule dans un terminal. Tu dois pouvoir expliquer à voix haute ce qu'elle fait avant de la garder.

4. Récupérer les processus, les applications, les téléchargements

Voici les **idées** des commandes — à toi de les compléter, de les filtrer, et de comprendre ce qu'elles renvoient.

Top 10 processus par CPU

```
1 Get-Process |
2     Sort-Object CPU -Descending |
3     Select-Object -First 10 Name, CPU, @{N='MemMo'; E={[math]::Round($_.WorkingSet/1MB, 2)}}
```

⚠ Le CPU renvoyé par `Get-Process` est un **temps cumulé en secondes** depuis le démarrage, pas un pourcentage instantané. Pour avoir un vrai % il faudrait deux mesures espacées dans le temps. **Pour notre TP, le temps cumulé suffit largement.** On le mentionnera dans l'UI.

Applications installées

Les applications Windows sont listées dans le registre. C'est pénible mais c'est la méthode fiable.

```
1 $keys = @(
2     'HKLM:\Software\Microsoft\Windows\CurrentVersion\Uninstall\*',
3     'HKLM:\Software\Wow6432Node\Microsoft\Windows\CurrentVersion\Uninstall\*'
4 )
5 Get-ItemProperty $keys |
6     Where-Object { $_.DisplayName } |
7     Select-Object DisplayName, DisplayVersion, Publisher |
8     Sort-Object DisplayName
```

💡 Pourquoi deux clés ? La deuxième (`Wow6432Node`) contient les apps 32 bits sur un Windows 64 bits. Sans elle, tu rates la moitié des programmes.

Téléchargements récents

```
1 $downloads = Join-Path $env:USERPROFILE 'Downloads'
2 Get-ChildItem $downloads -File |
3     Sort-Object LastWriteTime -Descending |
4     Select-Object -First 20 Name,
5         @{N='SizeMo'; E={[math]::Round($_.Length/1MB, 2)}},
6         @{N='Date'; E={$_.LastWriteTime.ToString("o")}}
```

5. Assembler le rapport complet

Une fois chaque morceau testé séparément, tu assembles **dans une variable** :

```
1 $rapport = [PSCustomObject]@{
2     hostname = $env:COMPUTERNAME
3     timestamp = (Get-Date).ToString("o")
4     system = @{
5         os = (Get-CimInstance Win32_OperatingSystem).Caption
6         osVersion = [string](Get-CimInstance Win32_OperatingSystem).Version
7         cpu = (Get-CimInstance Win32_Processor).Name
8         cpuCores = (Get-CimInstance Win32_Processor).NumberOfCores
9     }
10    memory = @{
11        totalGo = [math]::Round((Get-CimInstance Win32_ComputerSystem).TotalMemory)
12        freeGo = [math]::Round((Get-CimInstance Win32_OperatingSystem).FreeSpace)
13    }
14    disks = @( <# à toi #> )
15    topProcesses = @( <# à toi #> )
16    applications = @( <# à toi #> )
17    recentDownloads = @( <# à toi #> )
18 }
19
20 $json = $rapport | ConvertTo-Json -Depth 6
21 $json | Out-File "rapport.json" -Encoding utf8
```

⚠ **ENCODING** – toujours `-Encoding utf8` quand tu écris un fichier destiné à un serveur web. Sinon, accents cassés garantis.

6. Envoi vers une URL de test

Inscris-toi 30 secondes sur webhook.site (rien à installer). Le site te donne une URL unique. Tout ce qui arrive sur cette URL est affiché en direct dans ton navigateur. **C'est l'outil parfait pour voir ce que tu envoies vraiment.**

```
1 $url = "https://webhook.site/TON-UUID-ICI"
2
3 Invoke-RestMethod -Uri $url `
4                 -Method Post `
5                 -ContentType "application/json; charset=utf-8" `
6                 -Body $json
```

Va voir sur webhook.site. Tu dois voir ton JSON apparaître ligne par ligne. Si tu vois des `??` à la place des accents → relis le piège encodage.

Travailler avec l'IA – Debug

Symptôme typique : ton JSON arrive sur webhook.site mais les sous-objets sont affichés comme `System.Collections.Hashtable`.

Bon prompt :

```
Voici mon code PowerShell : [colle uniquement le $rapport = ... ] Voici ce que je reçois côté serveur : [colle ce que webhook.site affiche] Pourquoi mes sous-objets s'affichent comme System.Collections.Hashtable au lieu d'être sérialisés ?
```

L'IA va probablement t'expliquer le `-Depth`. Tu vérifies en lisant la doc Microsoft de `ConvertTo-Json`, tu ne te contentes pas de croire.

7. Transformer en fonction réutilisable

Tu finiras avec un seul script `inventaire.ps1` exécutable comme ça :

```
1 | .\inventaire.ps1 -ApiUrl "http://localhost/inventaire/api/report"
```

Structure suggérée :

```
1 param(  
2     [Parameter(Mandatory)]  
3     [string]$ApiUrl  
4 )  
5  
6 function Get-DiskInfo {  
7     # ...  
8 }  
9  
10 function Get-TopProcesses {  
11     param([int]$N = 10)  
12     # ...  
13 }  
14  
15 # ... autres fonctions  
16  
17 $rapport = [PSCustomObject]@{
```

```

18     hostname = $env:COMPUTERNAME
19     timestamp = (Get-Date).ToString("o")
20     disks     = Get-DiskInfo
21     topProcesses = Get-TopProcesses -N 10
22     # ...
23 }
24
25 $json = $rapport | ConvertTo-Json -Depth 6
26
27 try {
28     Invoke-RestMethod -Uri $ApiUrl -Method Post -ContentType "application/j
29     Write-Host "✅ Rapport envoyé : $($rapport.hostname)" -ForegroundColor
30 }
31 catch {
32     Write-Host "❌ Échec : $($_.Exception.Message)" -ForegroundColor Red
33 }

```

✅ Critères de réussite de l'étape 1

- [] Le script `inventaire.ps1` se lance avec un paramètre `-ApiUrl`.
- [] Le JSON envoyé respecte **strictement** le contrat (champs, types).
- [] Le résultat est visible sur `webhook.site` (pas de caractère cassé).
- [] Tu peux **expliquer chaque ligne** de ton script.
- [] Tu as testé : qu'est-ce qui se passe si l'URL est mauvaise ?

Pour la suite

À l'étape 2, on remplace `webhook.site` par **ta propre API PHP** qui va recevoir, valider et stocker ces JSON.