Inventaire — Étape 1 : collecter avec PowerShell

Étape 1 du projet inventaire : interroger ton PC, construire un JSON propre et l'envoyer à un endpoint de test.

5TQ | Découverte

Tu vas écrire **un seul script PowerShell** qui interroge ton PC, construit un objet JSON propre et l'envoie en `POST` à une URL distante.

🎯 Objectif de l'étape

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

1. interroge ton PC,
2. construit un objet JSON propre,
3. l'envoie 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.

Partie 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 :

```
Get-CimInstance Win32_ComputerSystem
Get-CimInstance Win32_OperatingSystem
Get-CimInstance Win32_Processor
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`.

```powershell
Get-Volume | Select-Object DriveLetter, FileSystemLabel, SizeRemaining, Size
```

> 💡 **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.

Partie 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.

```powershell
$rapport = [PSCustomObject]@{
    hostname    = $env:COMPUTERNAME
    timestamp   = (Get-Date).ToString("o")  # format ISO 8601
    os          = (Get-CimInstance Win32_OperatingSystem).Caption
    ramGo       = [math]::Round((Get-CimInstance Win32_ComputerSystem).TotalPhysicalMemory / 1
}

$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

```
{ "hostname": "PC-01", "ramGo": 16, "cpu": "Intel i5", "osCaption": "Windows 11" }
```

Option B — structuré

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

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.

Partie 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 :

```
{
  "hostname": "string, obligatoire",
  "timestamp": "string ISO 8601, obligatoire",
  "system": {
    "os": "string",
    "osVersion": "string",
    "cpu": "string",
    "cpuCores": "number"
  },
  "memory": {
    "totalGo": "number",
    "freeGo": "number"
  },
  "disks": [
    { "letter": "C", "totalGo": 500, "freeGo": 120 }
  ],
  "topProcesses": [
    { "name": "chrome", "cpu": 12.5, "memoryMo": 850 }
  ],
  "applications": [
    { "name": "VS Code", "version": "1.95.0", "publisher": "Microsoft" }
  ],
  "recentDownloads": [
    { "name": "setup.exe", "sizeMo": 45.2, "downloadedAt": "2026-05-12T14:30:00" }
```

```
  ],
  "module": {
    "name": "network|security|users|...",
    "data": { /* dépend du module */ }
  }
}
```

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.

Partie 4 — Récupérer processus, applications, 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

```
Get-Process |
    Sort-Object CPU -Descending |
```

```
    Select-Object -First 10 Name, CPU, @{N='MemMo'; E={[math]::Round($_.WorkingSet64/1MB, 1)}}
```

> ⚠️ 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.

```powershell
$keys = @(
    'HKLM:\Software\Microsoft\Windows\CurrentVersion\Uninstall\*',
    'HKLM:\Software\Wow6432Node\Microsoft\Windows\CurrentVersion\Uninstall\*'
)
Get-ItemProperty $keys |
    Where-Object { $_.DisplayName } |
    Select-Object DisplayName, DisplayVersion, Publisher |
    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

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

Partie 5 — Assembler le rapport complet

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

```powershell
$rapport = [PSCustomObject]@{
    hostname  = $env:COMPUTERNAME
    timestamp = (Get-Date).ToString("o")
    system    = @{
        os        = (Get-CimInstance Win32_OperatingSystem).Caption
```

```
        osVersion = [string](Get-CimInstance Win32_OperatingSystem).Version
        cpu       = (Get-CimInstance Win32_Processor).Name
        cpuCores  = (Get-CimInstance Win32_Processor).NumberOfCores
    }
    memory    = @{
        totalGo = [math]::Round((Get-CimInstance Win32_ComputerSystem).TotalPhysicalMemory/1GE
        freeGo  = [math]::Round((Get-CimInstance Win32_OperatingSystem).FreePhysicalMemory*1KE
    }
    disks           = @( <# à toi #> )
    topProcesses    = @( <# à toi #> )
    applications    = @( <# à toi #> )
    recentDownloads = @( <# à toi #> )
}

$json = $rapport | ConvertTo-Json -Depth 6
$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.

Partie 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.**

```
$url = "https://webhook.site/TON-UUID-ICI"

Invoke-RestMethod -Uri $url `
                  -Method Post `
                  -ContentType "application/json; charset=utf-8" `
                  -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.

Partie 7 — Transformer en fonction réutilisable

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

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

Structure suggérée :

```
param(
    [Parameter(Mandatory)]
    [string]$ApiUrl
)

function Get-DiskInfo {
    # …
}

function Get-TopProcesses {
    param([int]$N = 10)
    # …
}

# … autres fonctions

$rapport = [PSCustomObject]@{
    hostname  = $env:COMPUTERNAME
    timestamp = (Get-Date).ToString("o")
    disks     = Get-DiskInfo
    topProcesses = Get-TopProcesses -N 10
    # …
}

$json = $rapport | ConvertTo-Json -Depth 6

try {
    Invoke-RestMethod -Uri $ApiUrl -Method Post -ContentType "application/json; charset=utf-8"
    Write-Host "✅ Rapport envoyé : $($rapport.hostname)" -ForegroundColor Green
}
catch {
```

```
    Write-Host "❌ Échec : $($_.Exception.Message)" -ForegroundColor Red
}
```

✅ Critères de réussite

- [ ] 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 ?

Suite

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