GUIDE DES COMMANDES POWERSHELL — DEPLOY-GLPIAGENT.PS1

Chaque commande est présentée avec son rôle, sa syntaxe de principe et les pièges à éviter. L'ordre suit la logique du script.

23 min de lecture · 4 août 2026 · Vincent

SOMMAIRE

POWERSHELL · AVANCÉ

GUIDE DES COMMANDES POWERSHELL — DEPLOY-GLPIAGENT.PS1

Chaque commande est présentée avec son rôle, sa syntaxe de principe et les pièges à éviter. L'ordre suit la logique du script.

23 min · 4 août 2026

Ce guide explique **chaque commande** dont tu as besoin, comme si tu n'avais jamais fait de PowerShell. Pour chacune : à quoi elle sert, comment sa syntaxe se lit mot par mot, comment la tester **seule** dans une console avant de la mettre dans le script, et les pièges classiques.

Les exemples de syntaxe sont **génériques** (c'est du PowerShell de base, documenté partout) : c'est à toi de les adapter et de les assembler pour construire ton script.

0. LES BASES À CONNAÎTRE AVANT DE COMMENCER

Cinq notions qui reviennent partout dans ce guide :

Une cmdlet : c'est le nom des commandes PowerShell. Elles suivent toujours le format `Verbe-Nom` : `Get-Service` (« obtenir - service »), `New-Item` (« créer - élément »), `Remove-Item` (« supprimer - élément »). Le verbe te dit ce que la commande fait.

Un paramètre : une option qu'on donne à une cmdlet, toujours précédée d'un tiret : `Get-Service -Name glpi-agent`. Ici `-Name` est le paramètre, et `glpi-agent` la valeur qu'on lui donne.

Une variable : une boîte pour ranger une valeur, toujours préfixée par `$` :

```powershell
$monDossier = "C:\Temp\GLPI"
```

Ensuite, écrire `$monDossier` n'importe où équivaut à écrire son contenu. Important : dans une chaîne entre **guillemets doubles** `"..."`, les variables sont remplacées par leur valeur ; entre **guillemets simples** `'...'`, non.

Un objet : en PowerShell, les commandes ne renvoient pas du texte mais des **objets** avec des **propriétés**. Exemple : `Get-Service` renvoie un objet service, qui a une propriété `Status`, une propriété `Name`, etc. On lit une propriété avec un point :

```powershell
$service = Get-Service -Name glpi-agent
$service.Status
```

Pour découvrir toutes les propriétés d'un objet : `$service | Get-Member`.

Le pipeline `|` : la barre verticale envoie le résultat d'une commande à la commande suivante, comme une chaîne de montage :

```powershell
commande1 | commande2 | commande3
```

Chaque commande travaille sur ce que la précédente lui a passé.

---

1. `PARAM(...)` — RECEVOIR LES PARAMÈTRES DU SCRIPT

Rôle : permettre à celui qui lance le script de lui donner des informations (l'URL GLPI, le tag) sans modifier le code.

Le bloc `param(...)` se place **tout en haut du fichier**, avant toute autre instruction (seuls les commentaires peuvent le précéder). Comme c'est une syntaxe générique de PowerShell et pas la solution de l'exercice, voici le principe complet :

```powershell
param(
    [Parameter(Mandatory = $true)]
    [string]$ServerUrl,

    [string]$Tag,

    [switch]$ForceReinstall
)
```

Ce qu'il faut comprendre, ligne par ligne :

- `[Parameter(Mandatory = $true)]` rend le paramètre **obligatoire**. Si l'utilisateur lance le script sans `-ServerUrl`, PowerShell le lui demande interactivement au lieu de planter. L'attribut s'applique uniquement au paramètre qui le suit immédiatement.
- Le **type entre crochets** (`[string]`, `[switch]`) force le type de la valeur reçue. `[string]` = une chaîne de caractères.
- `Tag` n'a pas d'attribut Mandatory : il est **optionnel**, et vaudra une chaîne vide si absent.
- `[switch]` est un type spécial pour les options oui/non : on ne lui passe **pas de valeur**. Présent sur la ligne de commande = `$true`, absent = `$false`. Dans le script, tu le testes avec `if ($ForceReinstall) { ... }`.
- La **virgule** sépare chaque paramètre — c'est l'oubli classique qui fait planter le bloc.

À l'exécution, ça donne des appels comme :

```powershell
.\Deploy-GLPIAgent.ps1 -ServerUrl "https://helpdesk.example.test/front/inventory.php" -Tag "TSSR-J7"
```

ou avec forçage :

```powershell
.\Deploy-GLPIAgent.ps1 -ServerUrl "..." -Tag "TSSR-J7" -ForceReinstall
```

Ensuite, dans le corps du script, les variables `$ServerUrl`, `$Tag` et `$ForceReinstall` sont **directement utilisables partout** — pas besoin de les redéclarer.

Améliorations pour plus tard (quand la base fonctionne) : `[ValidateNotNullOrEmpty()]` sur ServerUrl pour refuser une chaîne vide, ou `HelpMessage = "..."` dans l'attribut Parameter. Le bloc simple suffit pour le minimum attendu.

---

2. VÉRIFIER LES DROITS ADMINISTRATEUR

Rôle : refuser de continuer si le script n'a pas les privilèges admin (sinon msiexec échouera plus loin, de façon moins claire).

COMPRENDRE D'ABORD LES TROIS NOTIONS

Windows sépare trois choses :

- **L'identité** (identity) : QUI est connecté. Ton compte, son nom, son jeton de sécurité.
- **Le principal** (principal) : l'identité **plus** ses appartenances aux groupes. C'est l'objet capable de répondre à « cet utilisateur fait-il partie de tel groupe ? »
- **Le rôle** (role) : un groupe prédéfini de Windows (Administrators, Users, Guests...), listé dans l'énumération `WindowsBuiltInRole`.

Le test = récupérer l'identité → l'envelopper dans un principal → demander au principal si l'identité appartient au rôle Administrator.

Piège UAC à connaître : être membre du groupe Administrateurs ne suffit pas. Avec l'UAC, un admin qui ouvre une console normale travaille avec un jeton **filtré**, sans les privilèges. Le test renverra `$false` tant que la console n'est pas ouverte via « Exécuter en tant qu'administrateur ». C'est exactement ce qu'on veut détecter.

ÉTAPE A — RÉCUPÉRER L'IDENTITÉ COURANTE

Syntaxe générique .NET/PowerShell :

```powershell
$identite = [Security.Principal.WindowsIdentity]::GetCurrent()
```

Mot par mot :

- Les **crochets** autour du nom disent à PowerShell « ceci est un type (une classe .NET), pas une commande ».

- `::` (deux-points doublés) : la syntaxe pour appeler une **méthode statique** — une méthode qu'on invoque directement sur la classe, sans créer d'objet avant.
- `GetCurrent()` : la méthode en question. Parenthèses vides = aucun argument. Elle renvoie un objet représentant l'utilisateur qui exécute le script.
- `$identite = ...` : on range le résultat pour l'étape suivante.

Mini-test : tape la ligne seule dans une console, puis `$identite.Name`. Tu dois voir ton `MACHINE\utilisateur`.

ÉTAPE B — CONSTRUIRE LE PRINCIPAL AVEC `NEW-OBJECT`

`WindowsPrincipal` n'a pas de méthode statique : il faut **fabriquer un objet**, et sa recette de fabrication (son **constructeur**) exige une identité. Forme générale de `New-Object` :

```powershell
$objet = New-Object NomComplet.De.LaClasse($cequonluidonne)
```

- La classe à utiliser ici : `Security.Principal.WindowsPrincipal`
- L'argument attendu entre parenthèses : ta variable de l'étape A.
- Range le résultat dans une variable, par exemple `$principal`.

À toi d'assembler cette ligne.

ÉTAPE C — POSER LA QUESTION AVEC `.ISINROLE()`

Le principal possède une **méthode d'instance** `.IsInRole()` (« est dans le rôle ? »). Contrairement à l'étape A, elle s'appelle avec un simple **point** sur la variable :

```powershell
$variable.NomDeLaMethode(argument)
```

- L'argument à mettre entre parenthèses est le rôle à vérifier. Le rôle Administrateur est une **valeur d'énumération** (une liste fermée de valeurs nommées), et on y accède avec la même syntaxe `::` :

```powershell
[Security.Principal.WindowsBuiltInRole]::Administrator
```

Ici le `::` désigne une **valeur** de la liste, pas une méthode. - Le retour est un **booléen** : `$true` si les privilèges admin sont actifs, `$false` sinon. Range-le dans une variable du genre `$estAdmin`.

ÉTAPE D — RÉAGIR SI LE TEST ÉCHOUE

Un booléen se teste directement dans un `if`, sans le comparer à quoi que ce soit :

```powershell
if (-not $estAdmin) {
    # erreur claire + sortie (voir section 12)
}
```

- `-not` inverse la condition : « si PAS admin, alors... ».
- Dans le bloc : `Write-Error` avec un message qui dit **quoi faire** (relancer PowerShell en tant qu'administrateur), puis `exit 1`.

Ce test doit être **la première action** après le bloc `param`.

Validation : console normale → refus avec ton message. Console « Exécuter en tant qu'administrateur » → le script continue. Les deux comportements bons = étape validée.

---

3. `JOIN-PATH` — CONSTRUIRE UN CHEMIN PROPREMENT

Rôle : coller deux morceaux de chemin sans se soucier des `\`.

Syntaxe générique :

```powershell
$chemin = Join-Path <cheminParent> <nomEnfant>
```

Ce qu'il faut comprendre :

- Le premier argument est le dossier parent, le second ce qu'on ajoute dedans.
- La cmdlet gère elle-même le séparateur `\` : pas de risque de double `\\` ou de `\` oublié comme avec une concaténation `$a + "\" + $b`.
- Pour le dossier temporaire, le parent tout trouvé est la **variable d'environnement** `$env:TEMP` : c'est le dossier temporaire de Windows, déjà connu du système. Les variables `$env:...` donnent accès aux variables d'environnement Windows (`$env:TEMP`, `$env:COMPUTERNAME`, etc.).

Mini-test : `Join-Path $env:TEMP "Essai"` dans une console → affiche un chemin du genre `C:\Users\toi\AppData\Local\Temp\Essai`. Rien n'est créé : `Join-Path` fabrique juste la **chaîne de texte** du chemin.

---

4. `TEST-PATH` — VÉRIFIER QU'UN CHEMIN EXISTE

Rôle : savoir si un dossier ou fichier existe déjà, avant d'agir.

Syntaxe générique :

```powershell
Test-Path <chemin>
```

- Renvoie un booléen : `$true` si le chemin existe, `$false` sinon.
- Comme c'est un booléen, ça se met directement dans un `if` :

```powershell
if (Test-Path $monChemin) { ... }
if (-not (Test-Path $monChemin)) { ... }
```

Note les **parenthèses supplémentaires** autour de `Test-Path ...` quand on combine avec `-not` : sans elles, PowerShell croit que `-not` est un argument de `Test-Path`.

Mini-test : `Test-Path C:\Windows` → `True`. `Test-Path C:\NExistePas` → `False`.

---

5. `NEW-ITEM` — CRÉER LE DOSSIER TEMPORAIRE

Rôle : créer le dossier de travail où le MSI sera téléchargé.

Syntaxe générique :

```powershell
New-Item -ItemType Directory -Path <chemin> -Force | Out-Null
```

Mot par mot :

- `-ItemType Directory` : on précise qu'on crée un **dossier** (la même cmdlet sait aussi créer des fichiers, d'où la précision).
- `-Path` : où le créer — utilise la variable construite avec `Join-Path`.
- `-Force` : ne pas planter si le dossier existe déjà.
- `| Out-Null` : `New-Item` renvoie un objet décrivant le dossier créé, et PowerShell l'afficherait à l'écran. `Out-Null` jette cet affichage à la poubelle pour garder une sortie propre.

Mini-test : crée un dossier d'essai dans `$env:TEMP`, vérifie avec `Test-Path`, puis supprime-le (section 13).

---

6. `INVOKE-WEBREQUEST` — TÉLÉCHARGER LE MSI

Rôle : télécharger un fichier depuis une URL vers le disque.

Syntaxe générique :

```powershell
Invoke-WebRequest -Uri <URL> -OutFile <cheminDestination> -UseBasicParsing -ErrorAction Stop
```

Mot par mot :

- `-Uri` : l'adresse à télécharger (l'URL du MSI x64 — celle trouvée via l'API GitHub dans la version bonus, ou l'URL de la release fournie sinon).
- `-OutFile` : le chemin complet du fichier de destination, **dans ton dossier temporaire** (pas dans Downloads — c'est une contrainte de l'énoncé). Construis-le avec `Join-Path` : ton dossier temp + le nom du fichier MSI.
- `-UseBasicParsing` : option de compatibilité (évite une dépendance à Internet Explorer sur les vieux PowerShell). Ne coûte rien, mets-la.
- `-ErrorAction Stop` : voir section 11 — transforme un échec de téléchargement en erreur **bloquante**, pour que ton `try/catch` la voie.

Pièges :

- Sans `-OutFile`, la cmdlet ne sauvegarde rien sur le disque : elle renvoie la réponse en mémoire.
- Un téléchargement qui « réussit » mais renvoie une page d'erreur HTML au lieu d'un MSI, ça arrive : c'est justement le contrôle SHA256 (section suivante) qui te protège de ça.

Mini-test : télécharge n'importe quel petit fichier public vers `$env:TEMP`, puis vérifie son existence avec `Test-Path`.

---

7. `Get-FileHash` — CONTRÔLER LE SHA256

Rôle : calculer l'empreinte du fichier téléchargé et la comparer à l'empreinte officielle, pour garantir que le MSI est intact et authentique.

CE QU'EST UN HASH, EN DEUX MOTS

Un hash SHA256 est une **empreinte digitale** du fichier : une longue chaîne hexadécimale calculée à partir de son contenu. Si un seul octet du fichier change, l'empreinte change du tout au tout. Comparer ton empreinte calculée à l'empreinte publiée par l'éditeur prouve que le fichier est exactement celui attendu.

CALCULER LE HASH

Syntaxe générique :

```powershell
$resultat = Get-FileHash <cheminDuFichier> -Algorithm SHA256
```

- `-Algorithm SHA256` : l'algorithme demandé par l'énoncé.

- **Attention :** `$resultat` n'est PAS le hash. C'est un **objet** avec plusieurs propriétés (`Algorithm`, `Hash`, `Path`). Le hash lui-même est dans la propriété `.Hash` :

```powershell
$resultat.Hash
```

C'est cette propriété (une chaîne en MAJUSCULES) qu'il faut comparer.

COMPARER AVEC LE HASH ATTENDU

L'opérateur de comparaison d'égalité en PowerShell est `-eq` (pas `==` comme dans d'autres langages) :

```powershell
if ($valeurA -eq $valeurB) { ... }
```

- Sur des chaînes, `-eq` est **insensible à la casse** par défaut : pratique, car les hash publiés sur GitHub sont souvent en minuscules et ceux de `Get-FileHash` en majuscules. La comparaison directe fonctionne.
- Le hash attendu vient soit du champ `digest` de l'asset GitHub (bonus, voir section 15 — attention, il peut être préfixé par `sha256:`, à retirer avant de comparer), soit du fichier `.sha256` publié avec la release.
- **Si les deux hash diffèrent : erreur claire et ARRÊT du script, sans installer.** C'est tout l'intérêt du contrôle.

Mini-test : calcule le hash d'un fichier quelconque deux fois → même résultat. Modifie une lettre dans un fichier texte, recalcule → résultat totalement différent.

---

8. `START-PROCESS` + MSIEXEC — INSTALLER EN SILENCIEUX

Rôle : lancer l'installateur Windows (msiexec.exe) sans interface, avec les options GLPI, et savoir si l'installation a réussi.

POURQUOI `START-PROCESS` ET PAS JUSTE TAPER LA COMMANDE ?

On pourrait écrire `msiexec /i ...` directement, mais PowerShell rendrait la main **immédiatement**, sans attendre la fin de l'installation, et sans te donner le résultat. `Start-Process` règle les deux problèmes avec `-Wait` et `-PassThru`.

Syntaxe générique :

```powershell
$processus = Start-Process -FilePath <programme> -ArgumentList <tableauDArguments> -Wait -PassThru
```

Mot par mot :

- `-FilePath` : le programme à lancer, ici `"msiexec.exe"`.
- `-ArgumentList` : les arguments à lui passer, sous forme de **tableau de chaînes** séparées par des virgules :

```powershell
-ArgumentList "/i", $cheminMsi, "/quiet", "/norestart", "OPTION1=valeur", "OPTION2=valeur"
```

- `-Wait` : attendre que msiexec ait terminé avant de continuer le script.
- `-PassThru` : renvoyer un objet représentant le processus. Sans lui, `Start-Process` ne renvoie rien.
- `$processus.ExitCode` : le **code de sortie** de msiexec, lisible après coup grâce à `-PassThru` + `-Wait`.

LES ARGUMENTS MSIEXEC À CONNAÎTRE

- `/i` suivi du chemin du MSI : « installer ce paquet ».
- `/quiet` : aucune interface, aucune question.
- `/norestart` : ne pas redémarrer le poste tout seul.
- Les options GLPI Agent au format `OPTION=valeur` — **sans espace autour du** `=` (contrainte de l'énoncé). Celles qui t'intéressent : `SERVER=` (avec ta variable `$ServerUrl`), `TAG=` (avec `$Tag`), et l'option qui déclenche un inventaire immédiat — cherche `RUNNOW` dans la doc officielle « Windows installer » de GLPI Agent. Le mode **service Windows** est le comportement par défaut de l'installateur.

INTERPRÉTER LE CODE DE SORTIE

Ton script doit tester `ExitCode` et considérer 0 **et** 3010 comme des succès.

Pièges :

- Si le chemin du MSI contient des espaces, entoure-le de guillemets supplémentaires dans l'argument (guillemets dans la chaîne).
- Grâce aux guillemets **doubles**, `"SERVER=$ServerUrl"` insère bien la valeur de la variable. Avec des guillemets simples, non.
- La piste 5 de l'énoncé propose une variante : préparer la ligne complète puis l'exécuter via `cmd.exe /c msiexec.exe ...`. Les deux approches sont acceptables ; `Start-Process` te donne le code de sortie plus proprement.

---

9. `GET-SERVICE` — VÉRIFIER L'ÉTAT DU SERVICE

Rôle : contrôler qu'après installation, le service `glpi-agent` existe et tourne.

Syntaxe générique :

```powershell
$service = Get-Service -Name <nomDuService>
```

- Le nom du service à vérifier ici : `glpi-agent`.
- Le résultat est un objet avec notamment la propriété `.Status` : valeurs possibles `Running` (démarré), `Stopped` (arrêté), etc.
- **Comportement en cas d'absence :** si le service n'existe pas, la cmdlet lève une erreur. Deux stratégies au choix :
- `-ErrorAction Stop` → l'absence part dans ton `catch` (service manquant = installation ratée = échec clair) ;
- `-ErrorAction SilentlyContinue` → la variable reste vide, et tu testes `if ($null -eq $service) { ... }` pour gérer le cas toi-même.

Pour l'affichage final demandé par l'énoncé, montrer `.Status` (et éventuellement `.Name`) suffit.

Mini-test : `Get-Service -Name Spooler` (le spouleur d'impression, présent sur tout Windows) → observe l'objet renvoyé et sa propriété Status.

---

10. `START-SERVICE` — DÉMARRER LE SERVICE SI BESOIN

Rôle : si le service est installé mais arrêté, le démarrer.

Syntaxe générique :

```powershell
Start-Service -Name <nomDuService>
```

- À placer dans un `if` : seulement si `.Status` n'est **pas** `Running`.
- La comparaison « différent de » en PowerShell : `-ne` (not equal), le cousin de `-eq`.
- Après un `Start-Service`, relis l'état avec `Get-Service` pour l'afficher dans ton résumé.

---

11. `TRY / CATCH / FINALLY` + `-ERRORACTION` — GÉRER LES ERREURS

Rôle : intercepter les échecs (téléchargement, service absent…) pour afficher un message clair au lieu d'un pavé rouge illisible, et garantir le nettoyage final.

LA STRUCTURE

```powershell
try {
    # les étapes qui peuvent échouer
}
catch {
    # ce qu'on fait si une erreur bloquante survient dans le try
}
finally {
    # exécuté DANS TOUS LES CAS, succès ou échec
}
```

Ce qu'il faut comprendre :

- Dès qu'une erreur **bloquante** survient dans le `try`, PowerShell saute immédiatement dans le `catch` (le reste du `try` n'est pas exécuté).
- Dans le `catch`, la variable automatique `$_` contient l'erreur. `$_.Exception.Message` en donne le message lisible, à intégrer dans ton propre message d'erreur.
- Le `finally` est optionnel mais précieux : c'est l'endroit idéal pour le nettoyage du dossier temporaire (section 13), car il s'exécute même si tout a explosé avant.

LE PIÈGE N°1 DU DÉBUTANT : `-ERRORACTION STOP`

Beaucoup de cmdlets lèvent par défaut des erreurs **non bloquantes** : elles affichent du rouge mais le script **continue**, et le `catch` ne se déclenche pas. Pour forcer une cmdlet à produire une erreur bloquante (donc « catchable ») :

```powershell
UneCmdlet -SesParametres -ErrorAction Stop
```

Mets `-ErrorAction Stop` sur toutes les étapes critiques : téléchargement, calcul de hash, vérification du service. Sans ça, ton `try/catch` semble ne « pas marcher ».

Mini-test : dans une console, essaie `Get-Service -Name nexistepas` seul, puis la même chose dans un `try/catch` **sans** `-ErrorAction Stop` (le catch ne se déclenche pas), puis **avec** (le catch se déclenche). La différence saute aux yeux.

---

12. `WRITE-HOST`, `WRITE-ERROR`, `EXIT` — AFFICHER ET SIGNALER

Rôle : informer l'utilisateur de l'avancement, et signaler un échec de façon exploitable.

`WRITE-HOST` — MESSAGES D'AVANCEMENT

```powershell
Write-Host "Etape X : ce que je suis en train de faire..."
Write-Host "OK" -ForegroundColor Green
Write-Host "ECHEC" -ForegroundColor Red
```

- Affiche du texte à l'écran, c'est tout.
- `-ForegroundColor` colore le texte : vert pour les succès, rouge pour les échecs — très lisible pour ton compte rendu.
- Bonne pratique : un message **avant** chaque grande étape, et un message de résultat après. C'est ce qui rend le script « relisible » et son exécution compréhensible.

`WRITE-ERROR` — SIGNALER UNE ERREUR

```powershell
Write-Error "Description claire du probleme et quoi faire"
```

- Affiche le message sur le **flux d'erreur** (en rouge), le canal prévu pour ça — contrairement à un simple `Write-Host` rouge.
- À utiliser dans tes `catch` et tes tests qui échouent (hash invalide, pas admin...).

`EXIT` — SORTIR AVEC UN CODE

```powershell
exit 1
```

- Termine le script immédiatement, avec un **code de sortie** : `0` = tout s'est bien passé (c'est le code par défaut si le script va au bout), toute autre valeur = échec.
- Ce code est le même mécanisme que l'`ExitCode` de msiexec : il permet à un autre outil (ou au formateur) de savoir si ton script a réussi sans lire les messages.
- Le duo classique en cas de pépin : `Write-Error "..."` puis `exit 1`.

---

13. `REMOVE-ITEM` — NETTOYER À LA FIN

Rôle : supprimer le dossier temporaire et le MSI téléchargé une fois l'installation terminée.

Syntaxe générique :

```powershell
Remove-Item -Path <chemin> -Recurse -Force
```

- `-Recurse` : supprime aussi tout le contenu du dossier (sans lui, un dossier non vide fait échouer la suppression, avec une question interactive).
- `-Force` : supprime aussi les fichiers cachés ou en lecture seule.
- **Où le placer :** dans le bloc `finally` (section 11), pour que le nettoyage ait lieu même si le script a échoué en route.
- **Prudence de débutant :** vérifie que la variable de chemin n'est pas vide avant de supprimer (un `Test-Path` dans un `if` fait l'affaire). `Remove-Item -Recurse -Force` sur un mauvais chemin ne pardonne pas.

PARTIE BONUS

À n'attaquer que quand le minimum fonctionne de bout en bout.

---

14. `INVOKE-RESTMETHOD` — INTERROGER L'API GITHUB

Rôle : demander à GitHub la liste des releases GLPI Agent pour trouver automatiquement la dernière version et son MSI x64.

API ET JSON, EN DEUX MOTS

Une **API web** est une URL qui, au lieu de renvoyer une page à regarder, renvoie des **données structurées** au format JSON (du texte organisé en champs et listes). L'URL à utiliser est donnée dans la piste 3 de l'énoncé.

Syntaxe générique :

```powershell
$reponse = Invoke-RestMethod -Uri <urlDeLApi>
```

Ce qu'il faut comprendre :

- Cousine d'`Invoke-WebRequest`, mais elle **convertit automatiquement le JSON en objets PowerShell**. Pas de texte à découper : tu obtiens directement des objets avec des propriétés.
- Ici, `$reponse` est un **tableau de releases**. Chaque release a notamment :
- `.tag_name` : le nom de version (ex. un tag contenant « 1.18 ») ;
- `.prerelease` : `$true` si c'est une préversion (à écarter) ;
- `.draft` : `$true` si c'est un brouillon (à écarter) ;

- `.assets` : la **liste des fichiers** publiés avec la release. Chaque asset a un `.name` (nom du fichier), une `.browser_download_url` (l'URL de téléchargement à donner ensuite à `Invoke-WebRequest`), et éventuellement un `.digest` (le hash, voir section 7).

Mini-test : appelle l'API dans une console, puis explore : `$reponse.Count` (combien de releases ?), `$reponse[0].tag_name` (la plus récente — les crochets `[0]` prennent le premier élément d'un tableau), `$reponse[0].assets.name` (les fichiers de cette release).

---

15. `WHERE-OBJECT` — FILTRER LES RELEASES ET LES ASSETS

Rôle : garder uniquement les releases stables, puis trouver l'asset `GLPI-Agent-*-x64.msi`.

Syntaxe générique :

```powershell
$resultat = $collection | Where-Object { <condition sur $_> }
```

Ce qu'il faut comprendre :

- `Where-Object` reçoit une collection par le pipeline `|` et ne laisse passer que les éléments pour lesquels la condition entre `{ }` est vraie.
- Dans la condition, `$_` désigne **l'élément en cours d'examen** (chaque release, ou chaque asset, tour à tour).
- Opérateurs utiles dans la condition :
- `-not` : inversion. Pour garder les stables : `-not $_.prerelease` ;
- `-and` : combiner deux conditions (pas prerelease **et** pas draft) ;
- `-like` : comparaison avec **joker** `*` (le `*` remplace n'importe quoi). Pour l'asset : condition sur `$_.name` avec le motif `"GLPI-Agent-*-x64.msi"`.

Tu auras donc **deux filtrages** : un sur `$reponse` (les releases), puis un sur la propriété `.assets` de la release retenue (les fichiers).

Mini-test : `Get-Service | Where-Object { $_.Status -eq "Running" }` → uniquement les services démarrés. Le principe est identique.

---

16. `SELECT-OBJECT -FIRST 1` — GARDER LA PLUS RÉCENTE

Rôle : après filtrage, ne retenir que la première release de la liste.

Syntaxe générique :

```powershell
$premier = $collection | Select-Object -First 1
```

- L'API GitHub renvoie les releases **triées de la plus récente à la plus ancienne** : le premier élément après filtrage est donc la dernière version stable.
- S'enchaîne naturellement après `Where-Object` dans le même pipeline : filtre, puis prends le premier.

---

17. `Get-ItemProperty` SUR LE REGISTRE — VERSION INSTALLÉE

Rôle : détecter si GLPI Agent est déjà présent sur le poste, et en quelle version.

LE REGISTRE, EN DEUX MOTS

Le **registre Windows** est la base de configuration du système. PowerShell y navigue comme dans un disque : `HKLM:` (HKEY_LOCAL_MACHINE, la config machine) s'utilise comme `C:`. Chaque programme installé laisse une **clé de désinstallation** contenant son nom et sa version.

Syntaxe générique :

```powershell
$programmes = Get-ItemProperty -Path <cheminRegistre>
```

Ce qu'il faut comprendre :

- Le chemin des programmes installés : `HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\*` — le `*` final signifie « toutes les sous-clés », donc tous les programmes.
- Par acquit de conscience, il existe une seconde branche pour les programmes 32 bits : même chemin avec `WOW6432Node` intercalé après `SOFTWARE`.
- Chaque objet renvoyé a (entre autres) :
- `.DisplayName` : le nom du programme tel qu'affiché ;
- `.DisplayVersion` : sa version, en chaîne de caractères.
- Pour isoler GLPI Agent : un `Where-Object` (section 15) avec une condition `-like` sur `.DisplayName` et le motif `"GLPI Agent*"`.
- Si le filtrage ne renvoie **rien** : l'agent n'est pas installé — c'est une information utile, pas une erreur (le script doit alors installer).

Mini-test : liste tout avec ce chemin, affiche les `DisplayName`, et retrouve à l'œil des programmes que tu connais.

---

18. LE CAST `[VERSION]` — COMPARER DEUX VERSIONS

Rôle : comparer correctement la version locale et la version en ligne pour décider s'il faut installer.

POURQUOI NE PAS COMPARER LES CHAÎNES DIRECTEMENT

En comparaison de **chaînes**, `"1.9"` est supérieur à `"1.18"` (comparaison caractère par caractère : « 9 » > « 1 »). C'est faux au sens des versions. Le type `[version]` de .NET compare **nombre par nombre** : 1.18 > 1.9, comme attendu.

Syntaxe générique du **cast** (conversion de type) :

```powershell
[version]$maChaine
```

- Placer `[version]` devant une chaîne la convertit en objet version comparable.
- Les opérateurs de comparaison PowerShell : `-lt` (less than, inférieur), `-gt` (greater than, supérieur), `-eq` (égal), `-le` / `-ge` (inférieur ou égal / supérieur ou égal). Exemple de principe :

```powershell
if ([version]$versionLocale -lt [version]$versionEnLigne) { ... }
```

Pièges :

- La chaîne doit être **purement numérique à points** (« 1.18 », « 1.18.0 »). Le `.tag_name` GitHub peut contenir un préfixe non numérique : nettoie la chaîne avant le cast (regarde du côté de la méthode `.TrimStart()` ou de l'opérateur `-replace`).
- Prévois le cas « rien d'installé » (section 17) : pas de version locale à comparer → on installe, tout simplement.
- C'est ici que se branche le paramètre `ForceReinstall` : même à jour, si le switch est présent, on installe quand même.

Mini-test : dans une console, compare `"1.9" -lt "1.18"` (résultat surprenant) puis `[version]"1.9" -lt [version]"1.18"` (résultat correct). Tu verras immédiatement pourquoi le cast est indispensable.

Ordre d'assemblage du script

1. `param` — paramètres (section 1)
2. Test admin (section 2)
3. Variables importantes visibles en début de script (chemins, noms...)

4. Dossier temporaire (sections 3, 4, 5)
5. (Bonus) API GitHub → release stable → asset x64 (sections 14, 15, 16)
6. (Bonus) Version locale + comparaison + ForceReinstall (sections 17, 18)
7. Téléchargement (section 6)
8. Contrôle SHA256 — arrêt si différent (section 7)
9. Installation msiexec + code de sortie (section 8)
10. Vérification / démarrage du service (sections 9, 10)
11. Résumé final pour le compte rendu (section 12)
12. Nettoyage dans le `finally` (sections 11, 13)

Le tout enveloppé dans `try / catch / finally`, avec `-ErrorAction Stop` sur les étapes critiques.

Méthode de travail conseillée

1. Teste chaque commande **seule dans une console** avec les mini-tests de ce guide, avant de l'écrire dans le script.
2. Construis le script **étape par étape** : fais fonctionner le test admin seul, puis ajoute le dossier temporaire, puis le téléchargement, etc. Relance le script à chaque ajout.
3. Garde les bonus pour la fin, un par un, sans casser ce qui marche.
4. Note au fur et à mesure les infos demandées pour le compte rendu (version détectée, URL du MSI, résultat du hash, code de sortie, état du service).